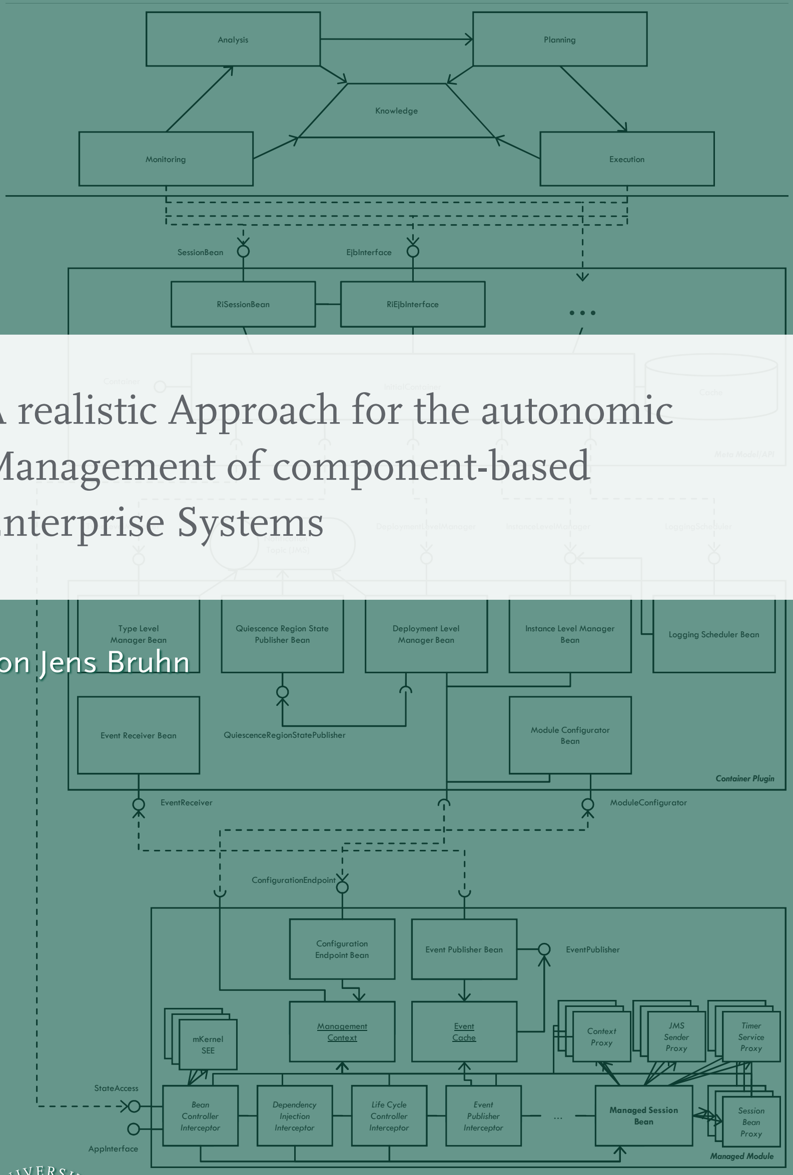
3 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg



A realistic Approach for the autonomic Management of component-based Enterprise Systems

von Jens Bruhn

UNIVERSITY OF BAMBERG PRESS

Schriften aus der Fakultät
Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Band 3

University of Bamberg Press 2009

A realistic Approach for the autonomic Management of component-based Enterprise Systems

von Jens Bruhn

University of Bamberg Press 2009

Bibliographische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Informationen sind im Internet über http://dnb.ddb.de/ abrufbar

Diese Arbeit hat der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität als Dissertation vorgelegen
1. Gutachter: Prof. Dr. Guido Wirtz
2. Gutachter: Prof. Dr. Andreas Henrich
Tag der mündlichen Prüfung: 5. Juni 2009

Herstellung und Druck: Digital Print Group, Erlangen
Umschlaggestaltung: Dezernat Kommunikation und Alumni

ISSN: 1867-7401

ISBN: 978-3-923507-49-8

URN: urn:nbn:de:bvb:473-opus-1986 (Online-Ausgabe)

To my beloved wife Barbara.

Acknowledgments

First of all, I thank my doctoral adviser, Prof. Dr. Guido Wirtz, for supporting my thesis through all the years. The provision of a working environment allowed me to develop my ideas in the first place. Moreover, the opportunities to present my work on international conferences were valuable and helpful contributions to this thesis. I also thank Prof. Dr. Andreas Henrich and Prof. Michael Mendler, PhD, for their support as member of my thesis committee.

Special thanks go to my colleagues and friends Sven Kaffille and Karsten Loesing. Their constructive criticism and their different views on the topic of this thesis during the countless discussions were of tremendous value for me. I am additionally very grateful for their motivational and helpful encouragement during the hard time of writing this thesis.

I thank the participants of the practicals and projects at the university of Bamberg who worked with the presented infrastructure. Their extensive usage of the AC-infrastructure and the corresponding feedback contributed to the refinement of *mKernel*. I specially thank Thomas Vogel for his high interest and his contribution to my thesis in different projects.

I thank my mother and my brother for proofreading the final manuscript of my thesis and for their constructive and motivational feedback. I owe my loving thanks to my wife Barbara. Without her encouragement and understanding it would have been impossible for me to finish this work.

Last, but not least, I want to thank Cornelia Schecher for her kind and helpful support regarding the administrative matters of my thesis. Additionally, Cornelia ensured that I never ran out of fuel through hectoliters of coffee.

Zusammenfassung

Seit einigen Jahrzehnten ist ein stetiges Ansteigen der Leistungsfähigkeit verfügbarer Hardwareressourcen festzustellen. Dieses ermöglicht die Zuweisung immer umfangreicherer Aufgaben an Softwaresysteme, was gleichzeitig ein massives Ansteigen der inhärenten Komplexität der verwendeten Systeme zur Folge hat. Ein für die Zukunft zu erwartender weiterer Komplexitätsanstieg erfordert eine explizite Adressierung. Das Konzept der Komponentenorientierung stellt einen Ansatz zur Komplexitätsreduktion für die Entwicklung und Konfiguration von Software durch funktionale Dekomposition dar. Mit der Vision des Autonomic Computing existiert ein Ansatz zur Komplexitätsbewältigung für Betrieb und Wartung von Softwaresystemen. In diesem Zusammenhang wird der Ansatz verfolgt, Aufgaben zur Feinsteuerung eines verwalteten Systems auf das System selbst zu übertragen. Da die Komponentenorientierung zu klar voneinander abgrenzbaren Elementen innerhalb von Systemarchitekturen führt, erscheint sie als viel versprechende Grundlage zur Realisierung der Vision des Autonomic Computing.

Diese Arbeit stellt eine realistische Infrastruktur für die autonome Verwaltung von komponentenbasierten Geschäftsanwendungen vor. Der Anwendungsbereich solcher Systeme stellt spezielle Anforderungen an verwaltete Systeme und ist besonders von der Komplexitätsproblematik betroffen. Um die praktische Relevanz der verfolgten Konzepte und Verfahren zu gewährleisten, wurde ein etablierten Komponentenstandards als Grundlage des Ansatzes gewählt. Bei diesem handelt es sich um Enterprise JavaBeans, Version 3.0. Die vorgestellte Infrastruktur ist generisch konzipiert und umgesetzt. Sie stellt sie eine Plattform bereit, auf

deren Basis Lösungen für verschiedene Anwendungsfelder des Autonomic Computing realisiert werden können. Zur Unterstützung autonomer Verwaltungseinheiten wird eine Programmierschnittstelle bereitgestellt, welche ein System auf drei Ebenen abbildet und dessen Steuerung ermöglicht: Auf oberster Ebene erfolgt die Betrachtung der einem System zugrunde liegenden Software. Auf mittlerer Ebene wird die Architektur eines verwalteten Systems adressiert. Interaktionen innerhalb eines Systems werden auf der untersten Ebene dargestellt. Auf dieser Grundlage kann ein System ganzheitlich und modellbasiert verwaltet werden. Zur Unterstützung der Laufzeitverwaltung eines Systems dient eine spezielle Komponente, welche in ein betroffenes System integriert werden muss. Sie ist konform zum verwendeten Standard und erfordert keine Anpassung der zugrunde liegenden Komponentenplattform. Für die Herstellung der Verwaltbarkeit von Komponenten wird ein Werkzeug bereitgestellt, welches automatisiert alle nötigen Anpassungen vornimmt. Darüber hinaus ist die Verwaltung eines Systems für dessen Elemente zur Laufzeit transparent. Zusammengenommen bleibt die Entwicklung von Geschäftsanwendungen von einer Verwendung der Infrastruktur unbeeinflusst.

Contents

List of Tables

List of Figures

List of Listings

1. Introduction

During the last decades information technology (IT) is characterized by constantly increasing performance of available hardware resources. The so-called *Moore's Law* [118] is a synonym for this development. Stated in 1965 by *Moore* it forecasts that the number of components per inch on an integrated circuit would double approximately each two years for at least the following ten years. With small deviations this prediction is fulfilled nearly up to now. This development led to the opportunity to assign more and more complex tasks to information systems. Additionally, the rapid increase of network bandwidth in combination with decreasing latency times supported the shift from monolithic applications, operating in isolation, to distributed, collaborating systems (cf. [20]). This evolution of IT led to an increasing infiltration of nearly all aspects of our everyday life. The diversity of applied devices reaches from high performance servers and clusters over desktop computers and notebooks down to mobile devices. In order to address the manifold application areas of IT, a broad range of software systems is applied. At the same time the complexity of software systems increased and still increases.

The term *Software Crisis*, first mentioned by *Dijkstra* in 1972 [63], stands for the problem of programming complexity to make use of the more and more powerful hardware resources. The applied concepts, tools, and programming languages up to that time were supposed of not being able to keep up with this development. As one reaction the discipline of *Software Engineering* was founded to support the development of software. While first intended to facilitate the tasks of programmers, it evolved to

> *"[a]n engineering discipline which is concerned with all aspects of software production from the early stage of system specification through to maintaining the system after it is gone into use."* (cf. [142], p. 6)

In 2001 *Horn* stated that addressing the complexity of computer system administration would be the *"next grand challenge"* (cf. [80], p. 1) of IT. He argued that the further development of IT would exceed the human ability to manage the future computer systems if no new concepts for management automation would be developed. In contrast to the software crisis, *Horn* did not address the software development, but its original usage in a concrete environment. In this context, he established the term *Autonomic Computing* (AC) and suggested that the administrative tasks should be assigned to the managed system itself to disburden human administrators.

In combination, the underlying ideas of software engineering and autonomic computing provide the foundations for addressing software complexity as a whole. There do exist different touch points between these two approaches which might influence the success of each of them mutually. Consequently, an inspection of the aspects influencing complexity and concepts for addressing it are of fundamental interest.

1.1. Enterprise Systems

According to the *IEEE Standard Glossary of Software Engineering Terminology* [87], the term *Software* is defined as

> *"[c]omputer [p]rograms, procedures, and possibly associated documentation and data pertaining to the operation of a computer system."* (cf. [87], p. 66)

Consequently, software does not only consist of one or many programs, but also includes other elements, enabling or supporting its operation. Summarizing, software covers all aspects of making a computer system usable. The definition does not include the actual usage of software itself, but concentrates on the needed artifacts. It subsumes all types of software such as operating systems, word processors, or custom software. All of them might have special characteristics and demands. The further discussion in this section concentrates on a special family of software, namely *Application Software* which is, according to [87],

> *"[s]oftware designed to fulfill specific needs of a user; for example software for navigation, payroll, or process control."* (cf. [87], p. 10)

The definition commits application software to *specific needs of a user*. These needs relate to real world problems for which application software should provide a solution or at least assist its users during problem solving. The definition confines the range of considered software in that it excludes system software such as firmware or operating systems, providing an abstraction from hardware and an infrastructure for higher-level software. Moreover, multi-purpose software such as middleware or database systems is also excluded from this definition, because it does not relate directly to a *specific* problem of its users. Instead of that, it can be used as building block for the development of other software, for example, application software. In the following, the term *Software* is used as synonym for application software.

The above definitions focus on the constituting artifacts of software while excluding its original application. To keep this separation software in use is called a *Software System*, or *System* for short, in this thesis.

Nearly each software exhibits an internal structure which is called *Software Architecture* in literature. There does exist a broad consensus regarding the core aspects of such an architecture. Nevertheless, there do exist manifold definitions which extend these aspects for concrete application contexts. A comprehensive set of definitions can be found at [141]. One generic definition is provided by *Garlan and Perry* [70] who define software architecture as

> *"[t]he structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time."* (cf. [70], p. 269)

This definition concentrates on structural aspects of software which are reflected by the constituting *components* and the *relationships* among them. Consequently, not all parts of a software are addressed, but only those which directly contribute to its functionality. The second part of the definition addresses software engineering aspects of an architecture. The term *Component* suggests an abstraction from fine-grained details like data structures and concrete algorithms while focusing on coarse-grained elements of an architecture. What these fundamental elements are depends on the concrete architecture. *Principles* and *guidelines* relate to the application of methods which are manifested in a concrete architecture, as well as rules for the further development. The *evolution* of software, as mentioned in the definition, indicates that a software might be subject to changes due to various reasons. The definition states nothing directly about the level of granularity of a software architecture. While the definition in general is considered appropriate, the terms *Component* and *System* are mistakable, because they are used with different meanings in this thesis.Therefore, they are replaced with the terms *functional elements* and *software*, resulting in a working definition of *Software Architecture* as

> the structure of the functional elements of a software, their interrelationships, and principles and guidelines governing their design and evolution over time.

Analog to the distinction between software and system, the term *System Architecture* is used in this thesis if the overall structure of an applied system is referred to. It can be interpreted as instantiation of a software architecture, covering all concrete configuration aspects in a concrete environment, for example, physical distribution and relations to the system environment. It is conceivable that only a subset of the elements of the corresponding software architecture is manifested in a system architecture, for example, if a software architecture contains optional elements or alternatives for certain facilities.

Enterprise systems are a family of software systems especially developed for enterprise environments. The specific needs fulfilled by enterprise systems relate to the provision of business logic. In this context, the IT-centric view of *Swarz and DeRosa* [150] on enterprises provides an appropriate foundation for the further discussion. *Swarz and DeRosa* define an enterprise as

> *"[...] a collection of [enterprise] systems whose operational capabilities are inextricably intertwined with considerations of people, processes, and technology, whose boundaries are often imprecise, and which can often be characterized by a set of special, additional properties, such as emergent behavior, non-determinism, and environmental dependencies."* (cf. [150], p. 3)

According to the definition enterprise systems are the constituent elements of an enterprise. These elements, the relationships among them, and technological and organizational dependencies establish the corresponding enterprise architecture. Moreover, there might also exist relationships between enterprise systems and the environment of the enterprise, for example, if an enterprise system provides an access point for customers or suppliers. Therefore, integration is a major aspect of an enterprise architecture (cf. [102]). Emergence might result from the interplay of different enterprise systems and might comprise the potential for new capabilities. At the same time the behavior of an enterprise as a whole might become non-deterministic, that is, there might arise interaction situations with potentially harmful effects. *Swarz and DeRosa* state that

> *"[t]he architecture of the enterprise and both its explicit requirements and implicit potential capabilities will evolve and emerge as trends in technology, scope of the enterprise, the aggregate user base, and other factors evolve over time"* (cf. [150], p. 3)

Due to the fact that an enterprise is embedded in the real world, it is subject to new or changing influences and requirements on different levels over time. On technology level there might, for example, arise the need to provide Web Services for interaction with new suppliers. On organization level the demand for supporting new business areas might also emerge. If these influences and requirements cannot be addressed by the enterprise directly, a need for adjustment of its architecture is given. This might also include the addition, removal, or change of constituting enterprise systems, as well as relationships among them. *Fowler* [67] characterizes enterprise systems as data intensive, potentially large systems which are accessed by a potentially large number of users with different intentions and needs (cf. [67], p. 6 - 8). Therefore, the demand arises to support multiple views while keeping the underlying data consistent. Additionally, an enterprise system might interact with other systems in a heterogeneous environment. Thus, an enterprise system should itself support integration, for example, through the provision of standardized access points. One major source of complexity are organizational changes which must be reflected by changes of the provided business logic. Therefore, enterprise software must be constructed to support changes over time. As enterprise systems are intended to support the operating company doing its business, they are more or less critical success factors. For that reason the corresponding software must be of high quality. *Hasselbring and Reussner* [77] define seven quality attributes in the context of *Trustworthy Computing* which are also of high relevance for enterprise systems:

1. **Correctness**: Correctness relates to the accordance of enterprise software with its specification. Due to the application area of enterprise systems this aspect is of very high importance, especially if an enterprise system has direct legal impacts such as the conclusion of the contracts in an online store.

2. **Safety**: The application of a safe system does not harm its environment. This attribute mainly relates to systems directly or indirectly affecting their physical environment like in automated warehouses.

3. **Availability**: Availability is the degree of reachability for service of a system over time. The more an enterprise system contributes to business activities, the more its availability is critical for the operating company. If, for example, an online store for customers has a low availability, the loss of business opportunities, trust, and reputation could have serious consequences.

4. **Reliability**: A reliable system has a low fraction of incorrect behavior in its overall processing. This aspect directly relates to correctness. While correctness focuses on the general absence of deviations from a specification, reliability relates to the actual occurrence of incorrect behavior.

5. **Performance**: A performance system is characterized by low response times and high throughput. This aspect is of special interest for efficient application of a system inside a company and for satisfactory usage experiences of external users. Moreover, an enterprise system should not only be performance for a given situation or state, but should also be scalable to support company growth.

6. **Security**: A secure system only provides its service to authorized users.

7. **Privacy**: Privacy demands that information is only submitted to those users which have the permission to access it. This is of special importance if the system deals with personal data or information sensible for the operating company.

Hasselbring and Reussner subsume the attributes 3 to 5 under the topic of *Quality of Service* (QoS), because they directly relate to the concrete usage experience of a system.

1.2. Life Cycles

From the initial planning of a successful enterprise software project to the phaseout of support software passes through different phases. These phases in combination are called *Software Life Cycle*. On system level the term *System Life Cycle* comprises all phases from the installation preparation to the deinstallation of a single system. Both of them influence each other mutually, resulting from the relationship between software and system. The outcomes of the software life cycle phases directly affect the corresponding systems. The other way round, observations during system life cycles might directly affect the corresponding software life cycle, for example, through change requests or bug reports. Although the two life cycles are often considered in combination in literature, in the following two sections a clear distinction is kept to reach a separation of concerns for later discussions. Additionally, it is assumed that distinct groups of people are responsible for the different life cycles. This assumption might even hold for subsequent phases of a single life cycle (cf. [21, 127]). It is also conceivable that the life cycles collapse, for example, if an enterprise software is constructed and the corresponding system is administrated by the IT-department of a single company. Nevertheless, the different tasks, as discussed in the following, are considered of being also present for such scenarios. The following sections focus solely on architectural aspects.

1.2.1. The Software Life Cycle

The life cycle of successful software can broadly be divided into the three main phases *Planning*, *Development* and *Maintenance*. *Successful* in this context means that the software passes all phases of the life cycle, and that the process is not aborted. It neither relates to the software quality nor to its adaptation and acceptance.

The *Planning* is the preliminary phase laying the foundation for the original construction of a software. During this phase decisions are made whether and how the software project should be realized. Furthermore, the fundamental requirements on the envisioned software are defined. Beyond the establishment of a general frame, planning has no direct influence on the artifacts of the software itself. Depending on the concrete situation, the decision whether a software project should be started might be based on different aspects such as realizability, alternatives, risks, or economic aspects. A successful planning should result in a *feasibility study*, containing at least a first definition of the software to construct and a plan for its realization (cf. *Balzert* [15], p. 58 - 61). The planning phase can be estimated of being the shortest phase during the software life cycle.

During the *Development Phase* functional and non-functional requirements for the envisioned software are collected and realized. The goal of the development phase is the construction of artifacts for the initial software releases and for supporting the subsequent maintenance phase. There might exist different releases of a software, potentially assembled from different sets of artifacts. Supporting artifacts of the maintenance phase are those artifacts which are not directly incorporated in any release, but intended for internal use such as development documentations. The complexity of development results from the requirements stated for the software to develop. Ideally, they are addressed on top level through the establishment of an appropriate software architecture and are realized by programmers in form of high quality source code. Depending on the observer's viewpoint different aspect might influence the perception of software quality. For developers the source code complexity might be the most important aspect. In relation to the architecture the complexity of a software can arise on intra- and inter-element level (cf. [14]). The intra-element complexity relates to the internal realization of architecture elements. The inter-element complexity emerges directly from the coupling among them. It has been investigated that too small or too large elements result in increasing complexity of software. Nevertheless, there does not exist a general recommendation for number and size of elements in a software architecture (cf. [16]). Moreover, not only the software architecture and the mere size of source code influence software complexity, but also additional artifacts like the corresponding documentation. These artifacts play an important role for enhancing understandability and reducing perceived complexity (cf. [125]). The foundations of the subsequent maintenance phase are also laid during development regarding changes of the software (cf. [40]). If, for example, requirements for changes are foreseen during development they might be considered in a way that facilitates the later change integration. Additionally, a software might be designed open for extensions. Nevertheless, not all potential demands for adjustments or extensions might be foreseen during development. A software also highly influences the complexity of system administration, for example, through the provision of user interfaces for information discovery and reconfiguration (cf. [18]). Additionally, a high degree of automation regarding the execution of reconfigurations might facilitate administrative tasks. Moreover, openness for extensions would allow administrators to integrate enterprise specific enhancements (cf. [40]). For users the observable properties of a software system might be the most important aspects. Besides the core experiences of ease of use, trustworthiness plays an important role, as discussed in section 1.1. In addition to development, software complexity also has major influence on the subsequent maintenance phase. Thus, the development phase lays the foundation for later adjustments of software during maintenance. There are different software development processes proposed in literature, for example, the *Waterfall Model* [130], the *Spiral Model* [25], or the *Rational Unified Process* [88]. These processes subdivide development into phases with different tasks, leading to a structured execution of the development phase. A detailed discussion of concrete software engineering methods and processes is out of scope of this thesis. *Mens et al.* [115] state that the effects on the subsequent maintenance phase and on system life cycles are undervalued in the different software development processes. They demand that post-development changes of software, as well as the integration of those changes during system life cycles must be explicitly addressed during development. In the end of development the software is assembled and packed. Afterwards, it is released and can be transferred to its users. The original transfer is not part of the software life cycle, but belongs to the system life cycle.

The *Maintenance* phase addresses adjustments of a software after the end of its development. According to *Lehman* [103, 104] software used for solving real world problems must continually evolve. He argues that this type of software – *E-type programs* according to his classification – is subject to changing requirements which could not all be foreseen during development. If solutions for these requirements are not integrated into a software during maintenance, it would become progressively less satisfactory and its quality would appear to be declining. Types of adjustments were categorized by *Swanson* [149] into *corrective, adaptive,* and *perfective. Corrective* adjustments address the correction of errors in the software source code. Besides functional and non-functional errors *Swanson* also summarizes implementation adjustments under corrective adjustments, for example, to correct inconsistencies between design and implementation. Changes in the execution environment are the reasons for *adaptive* adjustments. These might, for example, become necessary when a new operating system version should be supported. Finally, *perfective* adjustments address the optimization of software, for example, through the application of better algorithms. Additionally, changes for enhancing maintainability are also covered under this category. The categorization was also adopted by the *Institute of Electrical and Electronics Engineers* (IEEE) for defining maintenance as the

> *"modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."* (cf. [108], p. 74)

The categorization of *Swanson* focuses on but is not limited to the source code of software. Although widely used in literature, the categories are subject to different interpretations. Moreover, there do exist different extensions to the core set of categories. *Chapin et al.* [43] provide a more fine-grained categorization, based on the work of *Swanson*. In this context, they, amongst others, extend the original categories with respect to a more fine-grained consideration about changes of the functionality provided by the software. They distinguish between situations where the users of the software are supported with additional functionalities (*enhansive*), and situations where deprecated functionalities are removed (*reductive*). Over time maintenance results in more and more adjustments of the original software leading to increasing inherent complexity. Additionally, the underlying software architecture itself might be changed in a way that might hinder future modifications (cf. [126]). Therefore, software becomes less maintainable over time unless maintainability is explicitly addressed through perfective adjustments (cf. [104]). Generally, the effects of ongoing maintenance are discussed in literature under the term *Software Aging* (cf. [125]). Taking these considerations as foundation *Bennett and Rajlich* [21, 127] subdivide the maintenance phase into the four stages *Evolution*, *Servicing*, *Phaseout*, and *Closedown*. During the evolution stage all types of adjustments might be performed. In the end of this stage software becomes legacy due to aging effects, and a subsequent *servicing* stage is entered. During this stage only minor adjustments are possible, mainly addressing *corrective* aspects. The *phaseout* stage is characterized by the absence of any further adjustments. Finally, in the *closedown* stage support for the software itself is discontinued. Users are only supported during migration to a replacing software, if any. Although this staged model does not need to be observable for any given enterprise software, and the concrete designs of the stages might vary, it delivers an appropriate blueprint describing the effects of software aging during maintenance. Moreover, it highlights why the maintenance phase should not be equated with continued development. While development addresses the construction of new software for fulfilling known requirements, maintenance is concerned with the integration of new aspects into an existing architecture.

1.2.2. The System Life Cycle

As discussed in section 1.1, enterprise systems are the constituent parts of an enterprise. As the enterprise itself is embedded in the real world, there do exist manifold technological and organization aspects influencing the constituent enterprise systems. The other way round, enterprise systems and relationships among them might affect the overall enterprise behavior. Correctness of a single system might depend on the correct behavior of other systems. Therefore, incorrect behavior might be observed which does not result from the affected system itself, but from an interaction with another, faulty system. Quality of service might degrade, for example, due to crashes, performance bottlenecks, or changing workloads. Security and privacy might not only depend on a system itself, but also on the underlying infrastructure, for example, a web server into which a web shop is installed. If such a server is vulnerable to *SQL Injection* (cf. [72]), the web shop might be attacked based on that threat. Thus, the fundamental complexity of enterprise management arises from the administrated systems and relationships among them, from the underlying infrastructure, from organizational aspects, and from environmental relationships (cf. [18]). *Halprin* [73] differentiates between *reactive* and *proactive* administrative workflows. While reactive workflows mainly address the resolution of problems and failures, proactive workflows are concerned with improvements of the enterprise such as the integration of new enterprise systems or the optimization of processes. As one consequence an enterprise system cannot be administrated in isolation, but is influenced by its environment to a high degree. Nevertheless, the following discussion focuses on the system life cycle of a single enterprise system, because the management of an enterprise as a whole is out of scope of this thesis. Additionally, the life cycle presented here addresses only activities directly related to the system itself. Accompanying activities such as the training of users are not covered. The life cycle of an enterprise system consists of the four main phases *Planning, Deployment, Management*, and *Undeployment*.

Analog to the software life cycle the *Planning* phase can be interpreted as a preparation for the subsequent deployment phase. It mainly addresses aspects of the identification of necessary tasks, resource allocation, and time scheduling. While the subsequent phases depend directly on the corresponding software, this phase might start even before the software is finished, that is, its development phase ended, because the system itself is not subject to any actions of this phase directly. Nevertheless, this might imply several uncertainties, for example, regarding the time when the software will be available, the set of configuration options, or the final environmental requirements.

The establishment of a system is performed during the *Deployment* phase. It covers all activities from obtaining the corresponding software up to the final activation of all parts of the affected system (cf. [57]). *Coupaye and Estublier* [50] also assign the packing and transfer on the software producer side to the deployment phase. This might be meaningful if the software is specially customized to the user needs on producer side. As preparation for the actual installation the target execution environment must be analyzed with respect to all aspects affecting the installation. Such aspects might, for example, be available hardware resources and software systems. If the new system should replace an existing one, the migration of data sources must also be considered, if necessary. Afterwards, the concrete installation process might be prepared. This is especially important if it leads to temporal shutdowns of parts of the enterprise, for example, when an upgrade is performed. Moreover, individual tasks must be coordinated if a distributed installation affects different sites. Besides time schedules and staff allocation, installation preparation also covers configuration aspects and the assembling of installation packages. These might, besides the software to install, also include additional programs on which the target software depends such as libraries or site-specific drivers. Moreover, necessary hardware enhancements must also be considered if a target environment does not fulfill the needs of the software to install. Following the installation plan, the system can be installed, tested, and finally activated (cf. [50]). During deployment the internal system architecture is determined, laying the foundations for later adjustments. Moreover, relationships to other systems might be established which implies a manipulation of the enterprise architecture. In literature the importance of deployment support from producers of software is pointed out, both for effectiveness and efficiency reason (cf. [89, 119]). Especially for large software systems the provision of installation tools, documentations, and support systems is highly recommended. Although the deployment phase of a system life cycle only represents a preparation for system application it might be a very complex and time consuming task. The *SAP Deutschland AG*, for example, highlights in a success story that the upgrade of an *Electronic Resource Planning* system (ERP) with 11,000 users took only four month (cf. [135]).

During the *Management* phase the system is productively used. The main objective of administrators during this phase is to reach a high degree of experienced system quality regarding the characteristics of trustworthy computing systems presented in section 1.1. In this context, there do exist many different sources of information which must be evaluated to identify needs for adjustment. Examples of those are log files, user feedbacks, or notifications of software producers regarding newly available updates. Furthermore, administrators must be supported with facilities to inspect the system state. Due to the different aspects to address there might exist a broad range of tasks to fulfill, for example, user account management, reactions to workload shifts, security management, repair of failures, or the integration of updates (cf. [18, 128]). Not all of these tasks have direct impacts on the actual system behavior. They are more related to system usage than to its management as considered in this section. These tasks are not discussed any further. The tasks addressed in the following demand for system reconfiguration with respect to structural and behavioral manipulation. In this context, no distinction is made regarding the reasons for applying concrete changes. *McKinley et al.* [111] distinguish between *parameter adaptation* and *compositional adaptation*. Parameter adaptation addresses reconfigurations which can be performed based on changes of system variables. In contrast, compositional adaptation refers to changes of the system architecture. In detail, this covers the addition, removal, or exchange of elements, as well as manipulations of connections among them (cf. [123]). Parameter adaptation is limited to changes, foreseen during software development. In contrast, compositional adaptation allows the integration of unforeseen changes like new functionalities. The time a compositional adaptation might be performed delivers a further distinctive feature. *McKinley et al.* [111] distinguish between *development time*, *compile time*, *load time*, and *runtime composition*. If composition can only be performed during development time or compile time the system behavior cannot be adjusted at all, but is completely determined by the corresponding software. While development time composition is limited to source code manipulations, compile time composition enables the customization for different target environments. Load time composition allows the configuration of systems, because it assigns the determination of the system structure and behavior to the startup of a system and the loading of its constituent elements. Nevertheless, the three previous composition types are summarized under the term *static composition*, because after determination of the system behavior it cannot be changed without halting and restarting the system. In contrast, *dynamic composition* supports the integration of changes into a system while it is running. *McKinley et al.* further subdivide systems allowing dynamic composition into *tunable* and *mutable* systems. While tunable systems prohibit manipulations of the provided business logic, mutable systems do not comprise this restriction.

As stated in section 1.1, availability is of very high importance for enterprise systems. It relates to the time when a considered system should be usable. There might exist systems which do not need to be available permanently, but, for example, only during the business hours of a certain office. For these systems the opportunity to perform static composition might be sufficient, because the execution of composition tasks might be scheduled to those times the system does not need to be available and thus allows a temporal shutdown. Nevertheless, there might also exist systems which need to be available permanently such as a web shop and the connected warehouse system. For that reason the opportunity to perform mutable composition would be of very high value (cf. [40]).

Depending on the concrete mechanisms for the execution of dynamic composition affected parts of a system might need to be isolated and be brought to a *quiescent state* for ensuring consistencies during reconfiguration. Ongoing interactions must be finished and newly initiated interactions must be avoided or blocked (cf. [99, 117]). For those approaches the disruption of user interactions should be minimized. Ideally, users would only recognize short delays and would not be confronted with interaction aborts. *Brown et al.* [29] address the complexity of system reconfiguration as perceived by human administrators. In this context, a reconfiguration subsumes all actions for transferring a system from an operational state into another. According to their work reconfiguration complexity is mainly determined by three factors: *execution complexity*, *parameter complexity*, and *memory complexity*. Execution complexity subsumes the number and complexity of actions to perform a certain reconfiguration. Parameter complexity relates to the configuration parameters for which values must be manually provided during the different actions of reconfiguration. In this context, not only the mere number of parameters is considered, but also the complexity for determining the concrete values. This might, for example, also cover the need to read documentations for identifying potential values and the process of selecting an appropriate one. Finally, memory complexity addresses the demand on administrators to keep different configuration aspects in mind. For all of these complexity factors *Brown et al.* also consider the complexity of context switches during reconfiguration, for example, if different systems must be reconfigured or the underlying infrastructure must be adjusted. *Brown et al.* concentrate on the execution of a concrete reconfiguration and do not cover other aspects such as planning, or the need for coordination among administrators and users. Nevertheless, their work provides an appropriate insight into those factors which directly influence concrete interactions with a system during reconfiguration. As one consequence it can be stated that reconfiguration complexity is strongly influenced by the applied tools and administrative interfaces, as well as by the degree of reconfiguration automation (cf. [18, 40, 92]).

The final phase of the system life cycle is the *Undeployment*. During this phase the system is removed from its execution environment. If underlying data sources or parts of them should be kept or transferred to a replacing system their export or migration must also be considered. The removal of the system might also include other elements of the target environment which were exclusively used by the system to be removed.

1.3. Towards an Autonomic Computing Infrastructure

As discussed in the previous sections, the life cycles of enterprise software and systems cover various sources of complexity. Although software and corresponding systems are closely related, a distinction between their life cycles can be drawn. This results from the targets to address during the particular life cycles, the types of tasks to fulfill, and the groups of affected people. During the software life cycle the majority of tasks are concerned with the development and maintenance of the software. In contrast, the system life cycle mainly concentrates on the management of a concrete system. During the software life cycle the foundation for the corresponding systems is laid. Therefore, results of a software life cycle have direct impacts on the life cycles of the corresponding systems, for example, through the support for mutable compositional adaptation. The other way round, relevant experiences during system usage and newly established or changing requirements for systems might be used as inputs for the maintenance phase of a software life cycle. Thus, a system life cycle might also influence the life cycle of the corresponding software.

Solving the challenges of the software life cycle lies in the domain of software engineering (cf. [142]). Regarding the system life cycle the vision of *Autonomic Computing* [80] demands that low level administrative tasks should be assigned to systems themselves to disburden human administrators. In this context, administrators are responsible to state high-level objectives which are autonomically adapted by the managed system itself. Consequently, the vision of autonomic computing is basically founded on the idea of automating system management. This overarching goal of a managed system is also called *Self-Management* in literature (cf. [105]).

McKinley et al. [111] postulate that the enabling technologies for self-adaptive systems are *Separation of Concerns*, *Computational Reflection*, and *Component oriented Design*. The term *Separation of Concerns* is often used in combination with the paradigm of *Aspect-Oriented Programming* (AOP)

[95]. It describes an approach for separating the development of the core software application logic from so called *Crosscutting Concerns* like security. Through this proceeding it is possible to address the different aspects of a software in isolation which would otherwise be *scattered* across different elements. A detailed discussion of AOP is left out here for brevity. According to *McKinley et al.* separation of concerns also facilitates the explicit manipulation of the different system aspects. The term *Computational Reflection* [106] describes the ability of a system to reason about itself and potentially change its own behavior. In relation to the architecture a reflective system might, for example, be able to analyze and manipulate the structure of its elements. Moreover, it might be able to inspect and intercept ongoing interactions. The concept of *Component Orientation* [151] represents an approach to establish a system in a modular way through functional decomposition. The modules – called *Component*s – a system consists of, encapsulate different functionalities and provide them to their environment through *Interface*s. Components can make use of other components through their interfaces. Consequently, the architecture of a component oriented system consists of loosely coupled components which collaborate among each other through their interfaces. The concept of component orientation is not limited to software systems. It also affects the software life cycle, because it demands for the development of software in form of components.

The goal of this thesis is the design and realization of an infrastructure for autonomic management of enterprise systems.

The provision of such an infrastructure should promote the vision of autonomic computing in general. This should be reached through the provision of facilities and services for managing entities to inspect and manipulate the managed system. For the development of the infrastructure a set of requirements is established. These requirements are organized in four categories, namely *Component Orientation Requirements, Software Re-*

quirements, *Manageability Establishment Requirements*, and *System Requirements*[1]. A following, detailed discussion of these requirements defines the goals of this thesis and explain its relation to the topics discussed so far.

Component Orientation Requirements (COR) Component orientation requirements address the demands on the infrastructure which are directly related to the concept of component orientation, a corresponding standard, and its application by the infrastructure.

Realistic Application Scenario (COR-RAS) The infrastructure should be designed and realized for a realistic environment. For that reason it should address an existing and accepted component standard which is used as foundation for enterprise systems in the real world.

Standard Compliance (COR-SC) The infrastructure itself should be based on the component standard, that is, it should be integrated in an execution environment (container) supporting the component standard. Therefore, the realization of the infrastructure should not violate the underlying component standard, but should be compliant to it as far as possible. Moreover, it should not make use of optional aspects of the supported component standard, if this can be avoided. This requirement should facilitate the potential application of the infrastructure within different environments.

Unchanged Container Implementation (COR-UCI) The infrastructure should be executable in an unchanged container supporting the component standard. It should especially not be necessary to adjust the implementation of a container to make it usable by the infrastructure. Otherwise, the infrastructure would not solely rely on a component

[1] A preliminary version of these requirements was presented in a paper for the *1st ICST/ACM International Conference on Autonomic Computing and Communication Systems (Autonomics 2007)* [34] and discussed in the corresponding talk.

standard, but also on a specific version or a set of versions of a concrete platform to which the necessary adjustments can be applied.

Consequently, the infrastructure should be designed and realized as a layer between a broadly accepted component standard (*COR-RAS*) and its implementation (*COR-UCI*). To avoid limitations of its application the infrastructure should rely on as few aspects not required by the standard as possible (*COR-SC*).

Software Requirements (SoftR) These requirements directly relate to enterprise software of which corresponding systems should be managed with the help of the infrastructure. They mainly address aspects of the development and maintenance phases of the software life cycle. The requirements do not apply to the software life cycle of management software which realizes the self-management goal of autonomic computing or parts of it.

Full Standard Support (SoftR-FSS) For the development of enterprise software the complete component standard should be applicable as defined in the corresponding specification. For that reason, the infrastructure should neither forbid the usage of parts of the standard, nor should it redefine or limit certain aspects of it. Otherwise, the development of enterprise software would be restricted. Furthermore, this would also limit compliance to the *COR-RAS* requirement.

Management Transparency (SoftR-MT) The later application of the infrastructure for enterprise system management should not impose any additional requirements beyond those of the component standard for the software life cycle. It should especially not be necessary to provide an additional specification for the infrastructure or to use an infrastructure-specific *Application Programming Interface* (API) for component development.

Self-managed Component Support (SoftR-SCS) The infrastructure should provide the opportunity to construct self-managing components. Therefore, management facilities provided by the infrastructure should be accessible from inside component source code. This would support developers to realize autonomic components which are able to manage themselves. Nevertheless, this should be an optional part of the infrastructure which is not violating *SoftR-MT*.

Summarizing, the software life cycle should not be affected by the infrastructure with respect to the implementation of the core business logic to avoid additional complexity (*SoftR-FSS* and *SoftR-MT*). Ideally, the infrastructure would only require standard-compliant components without any infrastructure specific extensions as output of software life cycles. Otherwise, the overall goal of complexity reduction would be thwarted at least for the software life cycle. Nevertheless, it should also be possible to consider management aspects as integral part of a component, if desired (*SoftR-SCS*). This does not conflict with the other requirements, because it should only be an opportunity.

Manageability Establishment Requirements (MER) In order to make components manageable by the infrastructure, adjustments and extensions are necessary. Restrictions regarding the proceeding for manageability establishment are defined by these requirements.

Manageability Automation (MER-MA) The establishment of manageability should be performed automatically, for example, through application of a tool without any complex configuration demands. Especially, the need to manually create additional artifacts or to manually adjust components would be a violation of this requirement.

Life Cycle Independence (MER-LCI) Manageability integration should neither be bound to the software life cycle nor to system life cycles. Instead of that, it should be possible to choose during which life cycle manageability is established. Furthermore, the integration should be independent of the concrete execution environment, that is, a prepared component should be usable in any container fulfilling the necessary prerequisites, for example, a needed database or other providers of required interfaces.

Software providers and system administrators should be supported with alternative options for software distribution and obtainment respectively (*MER-LCI*). On the one hand, providers should be enabled to establish manageability as additional step during the development and maintenance phase of the software life cycle. This allows them to distribute ready-to-manage components. On the other hand, administrators could integrate an additional step after the obtainment of components during the deployment or management phase to easily enable their autonomic management in a system based on the infrastructure (*MER-MA*).

System Requirements (SysR) During the system life cycle managing entities should be provided with appropriate facilities for supporting system management. In this context, system requirements define the necessary aspects to guarantee a comprehensive support for the vision of autonomic computing.

Centralized Management Support (SysR-CMS) As an alternative to *SoftR-SCS* the infrastructure should enable the development of management entities independent from concrete component implementations. Consequently, it should be possible to realize a kind of management layer, if desired. Moreover, such a layer or parts of it should not necessarily be bound to the same environment as the managed components. This would allow an integrated management of different systems which are not all based on the same component standard or container.

Reflective Meta Model (SysR-RMM) The infrastructure should provide a reflective meta model as foundation for inspection and manipulation of enterprise systems covering all relevant structural and behavioral aspects. Consequently, model based system management should be supported.

Life Cycle Coverage (SysR-LCC) The management support of the infrastructure should cover the whole life cycle of systems and their constituent deployed components. For affected deployed components – analog to the system life cycle – this covers all phases from their deployment through their management to their undeployment.

Software Relation (SysR-SR) It should be possible to establish a relation from elements of a managed system to the associated elements of the corresponding software. This would allow manifold helpful analyses such as the identification of the affected implementation when a defect is identified in a managed system. Moreover, alternative implementations for a certain functionality could be identified.

Genericity (SysR-G) The infrastructure should be designed to support different aspects of autonomic computing. In contrast, it should not be limited to special objectives.

Extensibility (SysR-E) The infrastructure should provide a foundation for different application areas of autonomic computing. Therefore, it should be extensible with respect to the integration of additional aspects relevant for those application areas.

Client Transparency (SysR-CT) Clients of a manged system should not be affected by the application of the AC-infrastructure, that is, they should not even notice any difference regarding the usage of system elements. This also implies that, during the development of client software, the infrastructure should not require to be considered.

The system requirements should ensure that system management is based on a comprehensive foundation. Moreover, it should be possible to separate management aspects from the core business logic of the system (*SysR-CMS*) or alternatively realize an integrated solution (*SoftR-SCS*). Management should be performed through model based inspection and manipulation (*SysR-RMM*). This should cover all aspects of the system life cycle (*SysR-LCC*), as well as relations to the corresponding software (*SysR-SR*). To support the vision of autonomic computing in general the infrastructure should be designed independent from concrete application areas (*SysR-G*) and should be open for extensions of specific application scenarios (*SysR-E*). Finally, client software and systems should not be affected by the AC-infrastructure at all (*SysR-CT*). Although this requirement also covers software aspects, its integration into *SysR* implies stronger demands. If only applied to client software the requirement would permit client software adjustments after finishing development. *SysR-CT* suppresses this opportunity.

The requirements stated above define the frame for the design and realization of the infrastructure. They also imply restrictions which should not be violated. The infrastructure should be developed on top of a broadly accepted component standard and should be used on top of an existing execution environment. Therefore, the thesis does not include the realization of a component environment itself. Moreover, the realization should concentrate on the establishment of manageability in a single environment. In contrast, aspects of interactions in a distributed component environment consisting of independent containers are not addressed. Finally, the thesis concentrates on system elements and the covered business logic. This implies the exclusion of aspects of graphical user interface (GUI), data sources, and other resources such as legacy systems.

1.4. Thesis Overview

As discussed in section 1.3, this thesis pursues the goal to provide a generic infrastructure for autonomic computing on top of an established and broadly accepted component standard. To provide an overview of related research areas chapter 2 presents the main contributing research topics, namely *Autonomic Computing* and *Component Orientation*. Afterwards, the chosen component standard *Enterprise JavaBeans* (EJB), version 3.0, is presented in chapter 3. The chapters 4 to 6 discuss the infrastructure proposed here in detail. This starts with a general overview in chapter 4 presenting the constituent elements of the approach and explains their responsibilities and relationships among them. The subsequent chapter deals with the external view on the infrastructure, namely the meta model and the realizing API (chapter 5). A white-box-view of the infrastructure is provided in chapter 6. Chapter 7 demonstrates the practical relevance of the infrastructure for different application areas of autonomic computing base on two projects which used the infrastructure as foundation. After the discussion of the approach related work is addressed in chapter 8. Finally, chapter 9 ends this thesis with a conclusion. This also includes an evaluation of the infrastructure against the requirements stated in section 1.3.

2. Background

The generic infrastructure proposed in this thesis is mainly related to two research areas, namely *Autonomic Computing* and *Component Orientation*. These are discussed in the following two sections.

2.1. Autonomic Computing

In 2001 *Paul Horn*, Senior Vice President of the *International Business Machines Corporation* (IBM), stated that the major problem of IT will be the constantly increasing complexity of system administration (cf. [80]). He argued that the complexity of future computer systems would reach a level that threatens to exceed the capabilities of human administrators if no new concepts are developed to cope with it. To address this problem as a whole he sketched the fundamental goal of a new computing paradigm, called *Autonomic Computing* (AC). The vision of AC is based on the idea to automate system administration through assigning management tasks to the system itself (cf. [68, 80, 94]). In relation to the discussion in section 1.2.2 an autonomic system would thus become responsible to self-administrate its own life cycle.

The vision of AC is based on a simplified analogy to the human autonomous nervous system. This system controls the human body and adjusts its behavior in reaction to internal events and conditions, or external influences, for example, through adjusting breathing rate or heart beat. The nervous system acts subconsciously, that is, the mind is neither involved in identifying relevant conditions nor in the selection and realization of appropriate actions for reaching a new balance. Thus, the mind is freed from detailed steering of the human body. Nevertheless, it is equipped with opportunities to control the behavior of the autonomous nervous system and the body, for example, through instructing the nervous system to stop breathing during diving. The autonomous nervous system is not a centralized system, but consists of a collection of autonomous entities with relationships among them. These entities interact and influence each other (cf. [69, 76, 80]).

An autonomic computing system should disburden human administrators from fine-grained steering tasks and allow them to focus on high-level, strategic aspects. In this context, they would become responsible to define goals or policies for the system which it realizes autonomically (cf. [1, 69]). Nevertheless, human administrators should keep control over all management aspects. Thus, they should be placed into the position to perform all necessary adjustments of a system in case of undesired behavior or critical situations. Consequently, the tasks of human administrators would shift from fine-grained steering to coarse-grained goal specification and intervention in situations an autonomic system does not exhibit the intended behavior (cf. [17, 86]). Additionally, an autonomic system would ideally react faster to management demands than human administrators and perform the necessary actions more quickly. Moreover, an idealized autonomic system would be free of human mistakes and realize its objectives free of errors. Summarizing, the vision of AC has the far-reaching goal to address autonomic system management in a holistic way (cf. [124, 144]). It is expected to lead to significantly lower *Total Costs of Ownership* (TCO) and better realization of trustworthy computing aspects with respect to system administration (cf. [110, 144]). Nevertheless, AC is not assumed of being fully realized in the near future, but can be interpreted as an inspiring idea for research and practice (cf. [93, 146]).

The following two sections give an overview of the most relevant aspects of AC against the background of this thesis. Section 2.1.1 discusses the fundamental characteristics a system should exhibit to support the vision of AC. Afterwards, section 2.1.2 presents high-level architectural considerations for the realization of autonomic management. Finally, section 2.1.3 summarizes the discussion.

2.1.1. Self-Management

The overall objective of an autonomic system is to manage itself according to goals stated by human administrators. This overarching ability is called *Self-Management* in literature (cf. [144]). In his constitutive paper [80] *Paul Horn* stated different characteristics an autonomic system must exhibit to reach the goal of self-management. These are adopted broadly in the literature on AC (cf. [105]). The set of characteristics can be divided into *Objectives* an autonomic system has to address and *Capabilities* which it needs to fulfill its objectives ([134, 145]). Thus, objectives can be seen as externally observable characteristics of autonomic systems while capabilities relate to internal characteristics enabling autonomic behavior. In combination, the characteristics are intended to provide a comprehensive, high-level set of aspects which must be addressed to realize the vision of AC in general. The following two sections discuss objectives and capabilities of AC.

2.1.1.1. Objectives

Objectives for autonomic systems are *Self-Protection*, *Self-Healing*, *Self-Optimization*, and *Self-Configuration*. The following discussion of these objectives shortly introduces the addressed aspects. Furthermore, relations to trustworthy computing properties are highlighted.

Self-Protection An autonomic system might be affected by manifold threats (cf. [145, 164]). External threats might arise from malicious interactions, for example, to gain unauthorized access to information or functionalities, to corrupt the system state, or to perform a *Denial of Service* attack (DoS-attack). Additionally, accidental or erroneous interactions might have negative impacts on a system. Finally, internal failures, faults, or unintended behavior of single elements might also harm the system state or behavior as a whole.

A self-protecting system should be able to detect, identify, and defend against the various types of threats (cf. [80]). If a harmful interaction was *"successful"*, the effects on the system should be confined, for example, through isolation of affected system elements. Due to changes in the environment or in its architecture new types of threats might arise which should ideally be anticipated by a self-protecting system in a way that makes it less vulnerable in the future (cf. [1, 94]). Besides reactions to relevant situations and events this might also include proactive countermeasures.

In relation to aspects of trustworthy computing self-protection should address *safety* in that it avoids or handles unintended behavior through appropriate countermeasures. Defensive measures should, amongst others, increase *availability*, for example, through reacting to or the avoidance of DoS-attacks. The handling of internal threats directly contributes to *reliability*, because it should prevent or confine the effects of incorrect system behavior. Finally, *security* and *privacy* are also addressed, because protection against unauthorized access to a system is directly covered in the considerations of self-protection.

Self-Healing Inconsistencies, malfunctions, and failures of a system might result from different reasons, for example, defects, failures of elements, or crashes of resources (cf. [110, 164]). Additionally, harmful interactions might have negative impacts on a system, as discussed in the context of self-protection. Moreover, as one result of self-protection certain elements of a system might be isolated and are not usable by other elements or external clients anymore.

A self-healing system should be able to detect, diagnose, and recover from those situations in a way that keeps or reestablishes its integrity and availability. In order to realize self-healing, a system should be able to identify the affected elements. These must be fixed if possible or otherwise removed or replaced. This might lead to the need for complex system adjustments (cf. [68, 134]). Additionally, inconsistent or corrupted states of underlying data sources should also be corrected, for example, through transaction rollbacks or the import of a backup. To reach a high level of availability system disruption as result of recovery should be minimized. The major tasks of self-healing can be assumed of being reactive, because they are performed in response to relevant situations. Nevertheless, self-healing might also cover proactive or preparatory actions, for example, the creation of backups or the establishment of fallback systems (cf. [68]). This thesis aims to provide a clear distinction between the targets of the different objectives. Therefore, recovery actions resulting from self-protection are subsumed under the self-healing objective. Nevertheless, some authors assign these actions to self-protection (cf., e.g., [68, 134]).

Self-healing mainly addresses the *availability* of autonomic systems through recovery from disruptions. Additionally, *reliability* is enhanced through the fixing, replacement, or removal of defect elements, and the correction of faulty data sources respectively.

Self-Optimization Over time, workloads a system has to cope with might increase or shift. This might lead to a performance degradation of a system and to breaches of *Service Level Agreements* (SLA). Additionally, QoS-requirements of users might increase, resulting in higher demands on performance which cannot be met by the given system configuration (cf. [1, 80]). It might also be possible that the set of assigned resources is changed, for example, when new hardware is integrated into the environment. Finally, software updates with different QoS-attributes might become available.

A self-optimizing system is responsible for autonomically react to those scenarios in that it optimizes itself, as well as the allocation and utilization of resources to better meet end-user expectations (cf. [1, 134]). As preparation a system must know or identify opportunities for optimization. These must be quantified to enable considerations about alternatives (cf. [94]). Self-optimization might, for example, be performed through system adaptation, resource adjustments, or changes of resource allocation. Parameter adaptation might lead to different performance properties which better fulfill user-requirements. Compositional adaptation might be applied to integrate updates or alternative implementations of system elements (cf. [94]). This might also include the integration or reconfiguration of workload managers (cf. [68]). If possible, resources might be reconfigured to obtain different performance properties better matching the new requirements. Finally, resources might be reallocated, for example, through the migration of system elements to new hardware resources (cf. [1, 68]). The above discussion might suggest a reactive nature of self-optimization. Nevertheless, it might also include proactive aspects, for example, workload forecasts and measures to address expected scenarios.

Self-optimization addresses the *performance* attribute of trustworthy computing.

Self-Configuration A system might be affected by changes in its environment which demand for the integration, adjustment, removal, or replacement of architectural elements, for example, to remove deprecated functionalities or to provide new ones. These demands were already discussed in the context of the system life cycle in section 1.2.2. They can be addressed through parameter or compositional adaptation. Nevertheless, adaptation might be a highly complex, error-prone, and time consuming task. Moreover, it might lead to undesired system disruptions.

To facilitate manipulations of a system architecture and its elements a self-configuring system should be able to perform goal-based adaptations. Such a goal might, for instance, consist of the demand for the integration of a new element or the need for a certain functionality. Subsequently, the system should ideally deduce all necessary actions from this goal and perform them autonomically. Thus, it would be sufficient to specify *what* should be done while leaving the realization details (*how*) to the system (cf. [94]). In literature there do exist two different proposals for addressing self-configuration. Some authors demand that autonomic elements should be able to act in a *plug and play* fashion, that is, they should be able to install themselves into an environment, identify providers of needed functionalities, establish connections to them, and finally publish their availability for allowing other elements to establish connections to them (cf. [68, 110]). This might also imply the initiation of self-configurations of other elements in case no provider for a needed functionality could be found. Moreover, self-configuring elements should monitor their environment and adapt to changes of the set of system elements if they are affected (cf. [94]). Consequently, the architecture of such a system consists of autonomic elements being responsible for their integration and for the establishment of connections. Alternatively, a system should be able to react to new or changing user demands through the centralized initiation of parameter or compositional adaptation (cf. [134]). For such a system the constituent elements are the targets of configuration performed by central management instances which instruct them to establish or remove connections, or perform parameter adaptation. The elements themselves would not perform any (re)configuration on their own.

Self-configuration does not address a specific aspect of trustworthy computing. Instead of that, it can be assumed of generally addressing demands for system adjustments during deployment and management.

Although the objectives are intended to address different aspects of self-management there might exist relationships, dependencies, and conflicts among them. When a system has, for example, protected itself against an attack its state might be left inconsistent. Therefore, the system should subsequently self-heal to correct the impacts of the attack. Self-configuration might support the other objectives with generic facilities for system adaptation (cf. [80, 145]). Moreover, there might exist conflicting goals for different objectives. While a self-optimization facility might, for instance, recommend the most efficient implementation of a certain functionality, a self-protection facility might prefer the integration of a different implementation providing a higher level of security. Therefore, the different objectives should not be considered in isolation at long sight. Instead of that, self-management of a system should be addressed in an integrated, holistic way (cf. [164]). A self-managing system should also report relevant incidents to human administrators. Moreover, it should provide facilities for gaining insight into the system architecture, configuration, and behavior to facilitate analyses in case human interventions become necessary (cf. [68]). This might also include histories and explanations of decisions during self-management. Nevertheless, this does not imply that human administrators would have to perform fine-grained tasks in any situation. Instead of that, this aspect represents an opportunity for special situations.

2.1.1.2. Capabilities

While the objectives of autonomic systems are widely adopted in literature, the capabilities stated by *Paul Horn* are not that much widespread. They can be interpreted as first considerations regarding a minimum set of capabilities a system must exhibit for being able to perform self-management. In particular, these capabilities are *Self-Awareness, Context-Awareness* and *Anticipatory* (cf. [80]). Additionally, *Sterritt and Bustard* also demand that a system must exhibit a *Self-Adjustment* capability (cf. [145]). The author of this thesis agrees with this requirement.

Self-Awareness A self-managing system must be able to perform analyses regarding its internal structure and behavior, as well as connections to and collaborations with other systems. If a system would not exhibit this capability it would not be able to reason about its current state and could not initiate any kind of reactive reconfiguration in response to relevant situations. *Sterritt and Bustard* [144, 145] also demand that a system should be able to monitor itself in order to identify situations demanding for adjustment. Therefore, they propose an additional capability, called *Self-Monitoring*, to distinguish pull-oriented inspection (self-awareness) from push-oriented information provision (self-monitoring). Nevertheless, this thesis subsumes both aspects under the term *Self-Awareness*.

Context-Awareness As stated in the context of the different objectives, a self-managing system must react to its environment and changes in it. To identify relevant aspects and situations a system must be aware of its deployment context. Analog to self-awareness, this might cover aspects of structural and behavioral inspection, as well as information provision, for example, through event facilities.

Self-Adjustment To realize the different demands for adjustments an autonomic system must be able to perform reconfiguration operations. Therefore, it must be able to manipulate its internal structure and behavior, that is, it must be enabled to perform parameter and compositional adaptation. Self-adjustment should not be equated with the self-configuration objective. While self-adjustment considers the set of fine-grained adaptation operations enabling the realization of objectives, self-configuration addresses complex procedures for transferring a system from one consistent state into another. Thus, self-configuration is established on top of self-adjustment.

Anticipatory To facilitate its administration a self-managing system should hide its realization details. It should be able to anticipate high-level goals and translate them into corresponding actions transparently. This capability does not subsume the objectives discussed in the previous section. It demands for the provision of an access point for human administrators which they can use to interact with an autonomic system.

Summarizing, capabilities address different interfaces a self-managing system needs to realize its objectives, as well as the need to provide appropriate access points for human administrators (anticipatory). Self-awareness and context-awareness are needed by a system for information collection. Thus, they establish the foundation to identify situations demanding for adjustment. Moreover, they might support the process of finding appropriate solutions in case additional information is needed. Finally, self-adjustment is needed to realize these solutions through the support with interfaces for adaptation. Beyond objectives and capabilities *Horn* demands that a self-managing system should be built on top of open standards to support the integration of different systems into an overall solution for autonomic management (*Openness*). Although this is a desirable characteristic it does not directly contribute to the realization of AC.

2.1.2. Autonomy Realization

The discussion of the previous section is related to the external view on a single self-managing entity or system respectively. It pointed out the desired behavior through objectives, as well as the necessary, underlying capabilities. This section will provide a conceptual insight regarding the realization of autonomic management. Therefore, section 2.1.2.1 discusses the *Control Loop* concept which represents the fundamental idea for realizing autonomy. Afterwards, section 2.1.2.2 provides a short overview of architectural considerations regarding the organization of multiple self-managing entities.

2.1.2.1. The Control Loop Concept

In order to realize self-management, an autonomic entity must be able to reactively or proactively address the different goals and policies stated by human administrators. This section concentrates on the identification of adjustment demands and on reactions to them performed by a single self-managed entity. A conceptual proposal for implementing self-management is provided by the so-called *Control Loop* concept (cf. [1, 62, 94]). Figure 2.1 depicts a schematic overview of this concept.

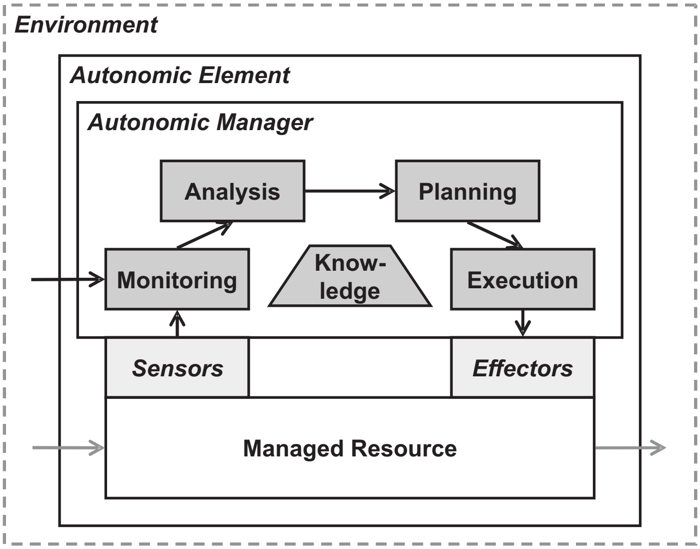


Figure 2.1.: The Control Loop Concept – Schematic Overview

The control loop concept supports separation of concerns regarding functionality and management aspects, as well as the surrounding environment. In this context, the *Environment* consists of all aspects which might influence an element and which might be influenced by the element (cf. [124]). These aspects might, for example, be related to other autonomic elements, other systems, or the underlying infrastructure. An *Autonomic Element* itself consists of two major parts, namely the *Managed Resource* and the *Autonomic Manager*.

A *Managed Resource* is that part of an autonomic element which covers the encapsulated functionality. This functionality might be accessed by the environment, and a managed resource might itself make use of functionalities from the environment. This is depicted in the figure through incoming and outgoing arrows. No restrictions are stated regarding the granularity level of managed resources. Therefore, a managed resource might be a building block of a system, a whole system, or even a collection of systems (cf. [62]). Moreover, managed resources are not limited to application systems. Operating systems, database management systems, or hardware resources might also be part of autonomic elements. In order to enable its management, a managed resource must provide access points for inspection and manipulation which are summarized under the term *Manageability Interfaces* in literature (cf. [1, 69]). These are represented in figure 2.1 on page 39 through *Sensors* and *Effectors*. Sensors represent access points for information discovery regarding structural and behavioral aspects. They might support push- and pull-oriented information discovery (cf. [143]). Consequently, sensors realize the self-awareness capability of autonomic elements. Effectors enable self-adjustment through the provision of manipulation facilities.

An *Autonomic Manager* is responsible for administrating one or many managed resources (cf. [94]). A manager realizes its objectives through application of a *Control Loop* consisting of the four stages *Monitoring, Analysis, Planning,* and *Execution* (cf. [1, 62, 94]). These stages are regarded as conceptual aspects which do not necessarily need to be realized through separate functions (cf. [1]) or components (cf. [62]). Therefore, this thesis considers the different aspects to address during a control loop cycle as stages while leaving their concrete realization open. The concrete tasks of the different stages are as follows (cf. [1,159]):

1. **Monitoring**: During monitoring information about managed resources is collected and preprocessed. Information from the environment might also be included. After preprocessing the information is forwarded to the analysis stage. The execution of the monitoring stage might, for example, be triggered in regular intervals or as a reaction to sensor events.
2. **Analysis**: The analysis stage is responsible for the identification of situations or states which rise demands for adjustment. Examples of those might be crashed resources, connection losses, or unintended behavior of elements. Moreover, relevant future situations might be estimated, for example, through workload forecasts. When a relevant situation is discovered, the planning stage is addressed.
3. **Planning**: In reaction to a relevant situation or state the planning stage constructs reconfiguration plans. These are intended to transfer the autonomic element from the current state into a state better fulfilling its stated goals. This includes the identification, evaluation, and selection of reconfiguration alternatives. Thus, the planning stage is responsible to support self-management through transforming goals into adaptation actions.
4. **Execution**: The generated reconfiguration plans are realized during the execution stage. In order to execute the different actions of the reconfiguration plans, this stage makes use of the effectors of the managed resource.

During execution of a control loop cycle internal *Knowledge* might be used, for example, covering information about symptoms of malicious behavior, goals and options for reconfiguration (cf. [1]). A control loop cycle does not necessarily need be a one-way process. It is, for instance, also conceivable that – during planning – additional information is needed which must be obtained from the managed resource or its environment. This would lead to a selective execution of the monitoring and maybe the analysis stage. Moreover, it might be necessary to observe the progress of the execution stage. This would lead to the demand for additional monitoring. Moreover, the different stages do not need to run to completion. If, for example, certain reconfiguration actions fail during execution it might become necessary to stop further execution and to go back to the planning stage to generate new plans.

2.1.2.2. Architectural Considerations

The discussion up to now focused on one single autonomic element which is responsible to fulfill the objectives of AC. The IT-infrastructure of an organization might consist of multiple autonomic elements which interact among each other to realize different goals. Some of them might be common to a set of elements while others are individual for a particular element. Due to direct or indirect interrelationships between the elements there might exist different kinds of touch points and influences among them. Moreover, different elements might compete among each other, for example, regarding shared resources. Finally, the goals of different elements might also conflict to a certain degree.

In literature different architectural approaches have been proposed for realizing autonomic management of complex IT-infrastructures in a holistic manner. This thesis does neither favor a specific style nor does it aim to propose its own new architectural style for AC. In contrast, the realized infrastructure should not limit or hinder different architectural styles. The architecture for autonomic management does not necessarily need to correspond with the architecture of the underlying managed resources. This would only be given if each resource of a system is assigned to a single autonomic manager. On the opposite side it is also possible that all resources of a system are managed by one single autonomic manager. If there does exist more than one autonomic manager collaborations among managers might be of different nature.

First of all, there does not necessarily need to exist any kind of collaboration at all. For such a scenario each manager might follow its goals in isolation and manage its resources accordingly. This might include the risk of suboptimal results, because managers might influence each other only indirectly through the effects of their actions. Cooperative approaches are not possible in this setting. Hence, human administrators would be responsible to synchronize the individual management goals.

Another approach would organize managers in a hierarchy. In this context, there are two different types of autonomic managers considered. Instances of the first type directly administrate one or many managed resources through their manageability interfaces. These managers are called *Touchpoint Autonomic Manager*s (TAM) (cf. [1, 69]). To reach an overall coordination among different autonomic managers so-called *Orchestrating Autonomic Manager*s (OAM) are applied. These do not directly interact with managed resources, but obtain information and requests from subordinate managers which are used as input for a higher-level control loop cycle to construct orchestrated plans. Although not explicitly considered in literature hierarchies of OAMs are also conceivable. In order to realize this architectural style, autonomic managers themselves must provide sensors which allow higher-level OAMs to inspect the corresponding autonomic element and to be provided with relevant information in a push-oriented way. Furthermore, effectors must be available to allow OAMs the manipulation of underlying managers and the corresponding autonomic elements.

In contrast to the hierarchical approach, it is also conceivable that autonomic elements are responsible to administrate their assigned resources completely autonomically. In this context, they are free to commit relationships with other elements and to negotiate the different aspects of these relationships. In such a scenario autonomic elements behave in an agent-like fashion (cf. [94, 110, 124]). To support this type of architectural style autonomic managers must exhibit access points for interaction and negotiation with other managers.

Hybrid architectures are also conceivable in which certain parts behave in an agent-like fashion while others are organized hierarchically. Moreover, layered approaches are also possible. It would, for instance, be possible that TAMs are managed by OAMs in a hierarchical manner. Each of these OAMs might perform autonomic management regarding a specific objective (cf. [1,94]). If potential effects on the goals of other OAMs or conflicts arise affected OAMs must negotiate about the execution of system adjustments.

Summarizing, autonomic managers do not only need to be able to accept goals from human administrators and provide inspection and manipulation access points for them. They should also be able to interact with other managers to cooperate with them and coordinate system adaptations, if necessary. The concrete nature of these access points depends on the applied architectural style. For a hierarchical approach managers should at least provide inspection facilities and should be able to accept instructions from higher-level managers. In an agent-like architecture access points should support at least negotiations among autonomic managers.

2.1.3. Summary

The vision of autonomic computing addresses the increasing administration complexity of todays and future systems through the automation of administrative tasks. Consequently, autonomic elements become responsible for managing themselves in accordance with high-level goals which relate to different objectives, namely self-protection, self-healing, self-optimization, and self-configuration. An autonomic element must exhibit certain capabilities to realize the overall goal of self-management. It must be able to inspect and monitor itself (self-awareness), observe its environment (context-awareness) and adjust itself, if necessary (self-adjustment). Finally, goals of human administrators must be anticipated (anticipatory).

In order to realize self-management, there does exist a schematic approach, called control loop, which is implemented by an autonomic manager being part of an autonomic element. A control loop consists of the four stages monitoring, analysis, planning, and execution. A manager interacts with a managed resource during the first and the last stage of a control cycle through manageability interfaces consisting of sensors and effectors.

On enterprise-level different self-managing elements might influence, compete, or even conflict with each other. Therefore, self-management should not be treated by each system in isolation at long sight, but should be addressed on enterprise level through an appropriate architecture. Alternative architectures might reach from a purely centralized approach within which one single manager is responsible for administrating the whole enterprise over a hierarchical organization of managers to an agent-like organization of autonomic elements.

2.2. Component Orientation

The concept of component orientation addresses complexity through modular design and functional decomposition. In this thesis the term *Component Orientation* is used in the context of software and systems, as discussed in section 1.1, because the thesis aims to provide an infrastructure for the autonomic management of component oriented enterprise systems. Therefore, the term *Component* is limited to software components in the following while excluding other areas where the term might be used with different meanings, for example, in the context of computer hardware. In this context, the term *Component* is used as synonym for *Software Component*. The term *Component* itself is discussed in section 2.2.1.

Afterwards, the establishment of a component-based system is treated in section 2.2.2. The role of standards for the concept of component orientation is discussed in section 2.2.3. Finally, section 2.2.4 concludes the discussion of component orientation with a summary of the relevant aspects. The thesis mainly addresses component oriented systems and only considers the outcomes of component based software development. The specifics of component oriented software development are not considered any further. Please refer to the corresponding literature for further details (cf., e.g., [28], [52] and [152]).

2.2.1. Components

The basic concept of component orientation is the so-called *Component*. Although there does exist great interest in the concept of component orientation in research and practice, there does not exist consensus about how to define the term *Component* (cf. [30]). A widely cited definition is provided by *Szyperski et al.* [151]:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."* ([151], p. 41)

The definition states that components are subject to deployment. Consequently, components are related to software life cycles, that is, they are the outcomes of the development or maintenance phase. To keep the separation between software and system, as presented in section 1.1, the term *deployed component* is used in the following if an element of a component system is meant. The definition of *Szyperski et al.* does not include such a clear separation. Nevertheless, the following discussion will highlight which parts relate to components and deployed components. The definition concentrates on the externally observable aspects of components. In contrast, it does not state anything about the realization of a components.

Thus, components are treated as black box units of software with externally observable characteristics. These characteristics must cover all aspects necessary to deploy a component into a component system (cf. [78]).

Interfaces According to the definition of *Szyperski et al.* the externally observable characteristics of a component consist of the associated *contractually specified interfaces* and the *explicit context dependencies*. Because of being software units components realize certain functionalities which are exposed through interfaces. Such a functionality specification can be interpreted as a contract between providers and users of a functionality in a system context defining the behavior of an implementation and the responsibilities of the interaction partners (cf. [28]). Therefore, an interface must at least contain syntactic information for accessing the functionality properly such as method signatures. Moreover, semantic aspects should also be covered, for instance, including restrictions regarding the range of values for method parameters and return values, or valid sequences of method invocations in an object oriented context. In combination, they define obligations for providers and users which, if adhered to, should guarantee correct interactions (cf. [116]). Obligations of users relate to all conditions which must be fulfilled before interacting with a provider while obligations of providers relate to the outcomes and effects of interactions. Obligations of one party are benefits of the corresponding counterparty. In this context, a functionality does only need to be provided correctly if a user meets its obligations. The other way round, a user can rely on the fulfillment of the provider's obligations if accessing the functionality correctly. Besides syntactic and semantic functional aspects, non-functional aspects might also be considered as part of an interface, for example, upper bounds for response times. For this thesis the aforementioned characteristics of interfaces are regarded as most relevant for component interfaces. Nevertheless, there might also exist other aspects which might be considered meaningful to be integrated into an interface. Examples of these are human readable documentations or different kinds of metadata (cf. [51]). These aspects are not explicitly covered by the definition of *Szyperski et al.*

Summarizing, interfaces specify the encapsulated functionalities of components and determine opportunities for the establishment of component systems (cf. [78]).

Dependencies In order to provide its encapsulated functionalities in accordance with the corresponding interface specifications, a deployed component might demand that different requirements are fulfilled in its deployment environment. These requirements are revealed through the *explicit context dependencies* on component level. They might address all environmental aspects such as (explicitly) usable main memory, the operating system, the concrete execution environment, or the availability of certain facilities. Moreover, there might also exist the need for access to providers of certain interfaces (cf. [152]). To allow a differentiation the implementation of an interface by a component is called *provided interface* while the dependency on an arbitrary implementation of a certain interface is called *required interface* in the following (cf. [100]). In this context, an interface itself is neither part of its providers nor of its requestors, but an independent functionality specification (cf. [28, 51]). Consequently, interface dependencies of components are independent from implementations, that is, from concrete components. They define the relation between arbitrary users and providers of a certain functionality.

Additional Information To allow the identification and selection of components to apply in a concrete scenario additional information might also be needful, for example, parameterization options, a human readable documentation, or automatically processable metadata. In contrast to the above mentioned information potentially attached to an interface, the information considered here does not directly relate to a concrete interface or dependency. It belongs to the component as a whole and can thus not be covered as part of an interface or dependency. Therefore, it should be treated as an independent aspect of a component.

2.2.2. System Establishment

In section 1.2.2 two types of adaptation were discussed for the life cycle of systems, namely *parameter* and *compositional adaptation*. Component systems are special types of systems to which these two adaptation types might also be applied.

Parameter Adaptation The definition of *Szyperski et al.* does not cover options for parameter adaptation. Although this opportunity does not need to be provided for all conceivable component systems it must be regarded as optional characteristic. Parameter adaptation does not relate to a component system as a whole, but to the constituent deployed components in particular. This is the case because of the stated independent deployment. It is conceivable that a comprehensive interface for parameter adaptation might be provided for a component system as a whole. Nevertheless, system parameters would be mapped to parameters of deployed component. For a concrete component system parameter adaptation might be limited to certain phases of the system life cycle. The discussion of component orientation within this thesis is kept general and subsumes all conceivable types of component systems. Therefore, no restrictions are stated regarding parameter adaptation during system life cycles.

Compositional Adaptation Although not explicitly excluded and theoretically possible deployed components are not intended to cover complete systems, but to be used as their basic building blocks. To integrate a deployed component into a component system, it is necessary to fulfill the *explicit context dependencies*, as stated for the corresponding component. As these are abstractly defined, a deployed component should be able to cooperate successfully with all conceivable realizations of a particular dependency specification. Focusing on component functionalities the architecture of a component system consists of the constituent deployed components and connections among their required and provided interfaces. In this context, compositional adaptation of a component system is related to the establishment, manipulation, and removal of connections among the constituent deployed components. The stated independence of deployment highlights that a deployed component is self-contained with respect to deployment operations. Thus, independence is limited to manipulations of the system architecture and does not imply that deployed components must be stand-alone units although this might be possible. In combination the abstract nature of required and provided interfaces, and the opportunity to deploy components independently lead to a loosely coupling of deployed components. Thus, the architecture of a component system consists of loosely coupled deployed components which interact among each other in accordance with contractually specified interfaces to provide the overall functionality of the system (cf. [137]). From an external point of view certain functionalities of deployed components might be provided to external system users. Thus, provided interfaces of deployed components are not intended to be exclusively used by other deployed components inside a system. Moreover, context dependencies of deployed components do not need to be solely fulfilled inside the system. Instead of that, they might be connected to provided functionalities of other systems.

The definition of *Szyperski et al.* implicitly denotes the potential reuse of components in different systems through mentioning their composition by *third parties*. In this context, *Crnkovic* [51] states that

> *"a component must be well specified, easy to understand, sufficiently general, easy to adapt, easy to deliver and deploy and easy to replace"* (cf. [51]).

The first aspect directly relates to the black-box-view of components with respect to their required and provided interfaces, other context dependencies, and configuration parameters. Secondly (*understand*), the potential application of a component in a system must be exposed to administrators in a way that enables them to decide whether the application of a given component is meaningful. Furthermore, administrators should be enabled to compare different components providing the same or similar functionalities. Reuse of components is only possible if they are *sufficiently general* to be applied in different contexts. Additionally, it is desirable that a component can be *easily adapted* to different user needs. This flexibility widens the range of application scenarios of a certain component. The delivery and deployment aspect addresses the relation between providers and users of components. Only if it is possible to distribute and obtain components, and subsequently integrate them into systems easily, it is possible to reuse components efficiently. Finally, an *easy replacement* of a component facilitates administrative tasks during the management phase of the component system life cycle. This might also include the support of runtime adaptation, as discussed in section 1.2.2.

The above discussion focused on the composition of deployed components, because this thesis addresses the automation of component system management. Nevertheless, it is also conceivable that a component oriented software might be constructed on the basis of existing components. This proceeding and potential effects on the software life cycle is not investigated any further. Please refer to the corresponding literature (cf., e.g., [28] and [52]).

2.2.3. Component Standards

Up to now the discussion focused on the black-box-view on components and deployed components. Moreover, the addressed aspects were only considered with respect to component orientation in general. For a concrete component system there must at least exist a common model according to which components can be specified and deployed components can be composed. According to *Lau and Wang*, a *Software Component Model* is

> *"[. . .] a definition of*
>
> - *the semantics of components, that is, what components are meant to be,*
> - *the syntax of components, that is, how they are defined, constructed, and represented, and*
> - *the composition of components, that is, how they are composed or assembled."* (cf. [100], p. 710)

Semantics addresses the external aspects of components, that is, their provided and required interfaces. Therefore, the semantics of a model relates to the black-box-view of components. This view is intended to provide an insight into required and provided functionalities, as well as relationships among them. The syntax of a component model specifies the concrete representation of the semantics. In this context, *Lau and Wang* distinguish between *component implementation languages* and *component definition languages*. The former language type supports the concrete realization of components based on a concrete platform such as the *Java* programming language. The latter type allows a separate definition of components potentially independent from a concrete implementation language. For such a case, a mapping between definition language and implementation languages is needed. Nevertheless, it is also possible that the two languages coincide. This would lead to a binding of the component model to a concrete implementation language or platform. Finally, the composition of components should be addressed through appropriate facilities. This should also cover supported types of compositional adaptation. Thus, the definition addresses all aspects of components, as discussed in the sections 2.2.1 and 2.2.2. An additional aspect relates to the potential binding of a component model to a concrete realization platform or language.

Summarizing, a component model must – according to *Lau and Wang* – address at least all black box aspects of components. A component model does not necessarily need to cover the internals of components or state any requirements for the infrastructure of component systems.

Szyperski et al. present and compare established component standards (cf. [151], p. 205ff). From their discussion the relevant aspects of these standards can be deduced. In general, a component standard subsumes at least a component model. It might also address additional aspects from concrete programming languages and APIs for component development over formats for component transfer to different phases of the life cycle of deployed components and parts of their internal realization. Moreover, a component standard might prescribe concrete interaction protocols, formats for information interchange, and facilities provided to deployed components. Nevertheless, the designs of concrete component standards might diverge broadly regarding the covered aspects.

Summarizing, the term *Component Standard* is used in this thesis to refer to a specification of requirements and recommendations for the life cycle of components and deployed components. It covers at least a component model addressing the black-box-view of components and the composition of deployed components. It might also address the internals of components, that is, their structure, needed and optional artifacts, as well as guidelines and requirements for their realization. Moreover, a component standard might address the execution environments of deployed components. For those it might require or recommend the existence of certain facilities, as well as mechanisms and proceedings for deployed components to interact with it. Component standards might be bound to other standards such as programming languages or interaction protocols for remote access to deployed components. Standards which are bound to concrete programming languages might also include specific APIs. Finally, a component standard might be dedicated to one or many application areas, for instance, enterprise systems. For a discussion of different component standards please refer to *Szyperski et al.* (cf. [151],p. 205ff) or *Lau and Wang* (cf. [100]). According to *Lau and Wang* [100], containers

> *"[...] are considered to be implementations of component models in that they provide an execution environment for components and their assemblies."* (cf. [100], p. 712)

The thesis follows this definition in that it refers to a container as a concrete execution environment for deployed components of which the corresponding components follow a certain component standard. In this context, the corresponding software is called *Container Implementation*.

2.2.4. Summary

The concept of *Component Orientation* was developed to address the complexity of software and systems. Its major subjects of consideration are the so-called *Component*s. These are units of software encapsulating certain functionalities. Functionalities are specified through *Interface*s which include at least syntactic aspects. Moreover, they might cover semantic and non-functional elements, as well as metadata and documentations. Interfaces represent contracts between providers and users of functionalities. In this context, they are independent specifications which might be referred to by an arbitrary number of providers and users. In order to provide encapsulated functionalities in a deployment context, a component might state different dependencies. These might relate to all conceivable aspects of an execution environment. One special type of dependencies, called *Required Interface*, is the need for access to providers of a certain interface. In addition to *Provided Interface*s and dependencies a component specification might also include metadata and documentations. Moreover, access points for parameter and compositional adaptation might be covered by a component.

A component could theoretically be deployed independently into each execution environment fulfilling the stated dependencies. Thus, components can be treated as self-contained black box units which could be reused in different contexts. The architecture of a component oriented system consists of the constituent deployed components and *Connection*s between their required and provided interfaces. The functionality of a system might be exposed through provided interfaces of the constituent elements while the system itself might be connected to other systems through environmental dependencies of its deployed components. Components are usually developed with respect to a certain *Component Standard*. Such a standard is a specification of requirements and recommendations for the life cycles of components and deployed components. Depending on the concrete standard components might be bound to a specific programming language or platform. The realization of a component standard through an execution environment for deployed components is called *Container Implementation* while the concrete execution environment is referred to as *Container*.

In the context of this thesis, component standards and containers provide the basis for realizing AC for enterprise systems according to the requirements stated in section 1.3.

3. Enterprise JavaBeans, Version 3.0

Enterprise JavaBeans, Version 3.0 (EJB) is a standard for the development and deployment of component-oriented enterprise software on top of the Java platform. It was specified under the leadership of *Sun Microsystems* (Sun) and was released in May 2006 as *Java Specification Request* (JSR) 220 [58–60]. It is supported by application servers of well-known companies like IBM [4], Red Hat [6], Oracle [8], and Sun [9]. The standard is part of the *Java Platform, Enterprise Edition, v5* (Java EE 5) [140] and addresses the business tier of enterprise systems as depicted in figure 3.1.

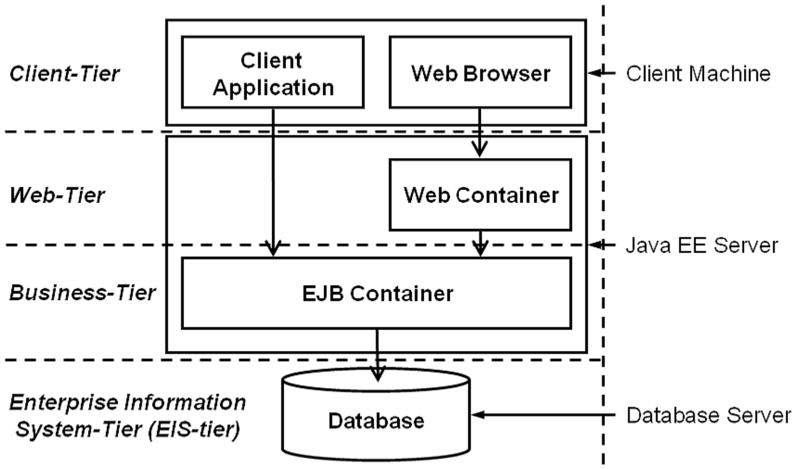


Figure 3.1.: Java EE – Schematic Overview (cf. [91], figure 1-1 and 1-5)

Consequently, EJB 3.0 concentrates on components implementing the business logic of enterprise software. Components are intended to be deployed into an *EJB Container* which is part of a *Java EE Server*. Thus, Enterprise JavaBeans is a server-side approach. Access to deployed components is mediated by the EJB container and is intended to be performed directly from inside *Client Application*s or indirectly from *Web Browser*s through web applications residing in a *Web Container*. In this context, *Web-Tier* and *Client-Tier* are, amongst others, responsible for presentation aspects which are not addressed by the EJB standard. The actual management of underlying data sources is also not directly covered by the EJB standard, but is intended to be part of the *Enterprise Information System-Tier* (EIS-tier).

The standard consists of three documents, namely the *EJB Core Contracts and Requirements* [58], the *Java Persistence API* [59], and the *EJB 3.0 Simplified API* [60]. The first document includes all aspects of the standard regarding the realization of components. Thus, this document builds the foundation for the further discussion in this chapter and for the thesis in general. The second document covers the specification of an object/relational mapping facility. This facility allows EJB developers to interact with relational databases in an object-oriented fashion and therefore provides an abstraction from the EIS-tier. The *Java Persistence API* is not discussed any further, because this thesis concentrates on the business-tier. Finally, the *EJB 3.0 Simplified API* does provide an overview of the APIs of the EJB standard. It does not cover any additional, relevant aspects beyond those of the other two documents. Therefore, it is not addressed any further.

The remainder of this chapter is structured as follows: First, an overview of the EJB component model is provided in section 3.1. The section covers aspects of EJB-based software and systems regarding the design of components, as well as fundamental concepts and techniques defined for their runtime environments. Additional facilities defined in the standard are discussed in section 3.2. The EJB standard considers seven distinct roles which relate to the life cycles of software and systems. These roles are presented in section 3.3. Java EE contains two standards for server and container management, as well as enterprise systems deployed into them. They address, amongst others, the execution environment of deployed EJB components. These standards are presented in section 3.4, because they are of high relevance for the life cycles of EJB-based systems. Finally, section 3.5 summarizes this chapter. This thesis does not aim to provide a detailed insight into the EJB standard. Therefore, the following discussion concentrates on a conceptual overview regarding those aspects which are relevant for the infrastructure proposed in this thesis.

3.1. The EJB Component Model

This section provides an overview of the component model defined in the EJB standard. Section 3.1.1 presents the basic building blocks of EJB-based components addressing the software level of the EJB standard. Afterwards, the deployment and management of component systems is discussed in section 3.1.2. Section 3.1.3 discusses the life cycles of the basic building blocks of deployed EJB components. Finally, the interceptor concept, as specified in the EJB standard, is presented in section 3.1.4.

3.1.1. Building Blocks of Components

EJB-based components are provided in form of *Java Archive*s [147] (JAR) which are called *ejb-jar file*s in the standard (cf. [58], p. 539ff). As elements of an ejb-jar file the standard mainly addresses class-files which are needed at runtime and a *Deployment Descriptor* (DD). Such a DD is based on the *Extensible Markup Language* (XML) [27] and might be used to specify different aspects of the corresponding component. Components are considered of following the *Write Once, Run Anywhere* (WORA) philosophy which states that they can be developed once and afterwards be deployed into different environments without the need for source code manipulation and recompilation (cf. [58], p. 29).

The main building blocks of EJB components are the so-called *Enterprise Bean*s or *Bean*s for short, which are realized as Java classes. Instances of these classes can be accessed internally or by external clients of a deployed component. The standard mainly considers two types of beans, namely *Message-Driven Bean*s (MDB) and *Session Bean*s (SB), which are discussed in the following two sections. Afterwards, section 3.1.1.3 focuses on specification aspects of components.

3.1.1.1. Message-Driven Beans

Message-driven beans are intended to be used as targets of asynchronous interactions. They are bound to message destinations during deployment of the corresponding component. The EJB standard explicitly considers the *Java Message Service,* version 1.1 (JMS) [75], as foundation for these destinations. The JMS standard is not discussed in detail in this thesis. Please refer to the corresponding document for further details (cf. [75]). During binding a *Message Selector* can be attached to an MDB. Such a selector contains the definition of certain criteria which must be fulfilled by arriving messages. Messages which do not match a selector of an MDB are not forwarded to an instance of that MDB (cf. [75], p. 41ff).

The JMS standard distinguishes between two types of message destinations, namely *Queue*s and *Topic*s (cf. [75], p. 75ff). Queues are used for *Point-to-Point* (PTP) interaction. An arbitrary number of *Receivers* might be registered at a single queue. On arrival of a message it is forwarded to at most one receiver, even if multiple receivers with matching selectors are bound to the queue. For such a situation the JMS standard does not prescribe which receiver should be chosen as target for message delivery, but leaves this open to the queue implementation. Moreover, the EJB standard does not address this aspect either. Consequently, not all messages matching the selector of a certain MDB are necessarily processed by instances of that MDB. Topics are intended to be used for a *Publish-and-Subscribe* (Pub/Sub) model. An arbitrary number of *Subscribers* might register for a particular topic. When a message arrives at the topic, it is forwarded to all registered subscribers with matching selectors.

The EJB standard does not define the conditions under which an MDB is instantiated. It only states that a container can construct MDB instances on demand, can hold a pool of instances of a certain MDB, and can destroy instances when they are not needed anymore. MDB instances are non-reentrant by definition. Therefore, a container must ensure that an MDB instance is processing at most one message at any given time. Furthermore, instances must not perform any kind of thread handling such as starting new threads or performing thread synchronization. When a message is received by a destination the container forwards it – depending on the destination type – to one or many instances of different MDBs with matching selectors, if any. Regarding a concrete MDB the container is free to choose which idle instance is selected for message processing. Consequently, not all messages chosen for delivery to an instance of a certain MDB are necessarily handled by the same instance of that MDB. Furthermore, a client cannot rely on interacting with the same instance when sending multiple messages. Therefore, the EJB standard demands that an MDB is implemented in a way that ensures equivalence of all of its instances for potential clients. MDB instances do not hold a client-specific conversational state. Nevertheless, they might keep a client-neutral state covering, for example, open data source connections for performance reasons.

3.1.1.2. Session Beans

Session bean instances are targets of synchronous interactions through method invocations. Analog to MDBs, instances of SBs are non-reentrant and are not allowed to perform thread handling. The EJB standard defines two types of SBs, namely *Stateful Session Bean*s and *Stateless Session Beans*. The main difference between these two types lies within the provision of a client-specific state. Instances of stateful session beans are exclusively used by a single client and retain a client-specific conversational state across multiple invocations. Additionally, a client can rely on interacting with the same instance in case it uses the same reference for multiple invocations. Nevertheless, a client might submit a copy of such a reference to other clients which might also interact with the corresponding SB instance. Stateless SB instances can be used by a container for handling method invocations originating from different clients. In this context, a container might construct an arbitrary number of stateless SB instances and keep them in a pool. From this pool the container might choose arbitrary instances for processing client interactions or for destruction. Furthermore, it is not guaranteed that a client, performing more than one method invocation on the same reference, is always interacting with the same session bean instance. A stateless SB instance might keep a client-neutral state during its lifetime which might, for instance, be the source of performance benefits in the same way as MDB instances.

Access to session bean instances is based on Java interfaces. These are divided into six different types, namely *home, remote, business, local home, local,* and *local business interface*. The former three types might be used to access session beans and their instances from inside a container as well as across its boundary. The latter three types can only be used as foundation for method invocations inside the same *Java Virtual Machine* (JVM). The standard does not prescribe that bean instances of all deployed EJB components inside a container are executed within the same JVM. Thus, access through local-* interfaces does not necessarily need to be provided across the boundaries of a deployed component. They are intended to be used inside deployed components. For the remainder of this thesis the local-* interfaces are summarized under the term *Internal Interfaces* while the other ones are subsumed under the term *External Interfaces*. Local home and local interfaces, as well as their corresponding remote counterparts are provided for downward compatibility with version 2.1 of the EJB standard. They are used in combination in each case. Local home and home interfaces address client accessible management aspects of session beans and their instances. These interfaces might, for instance, be used to create a reference to a corresponding SB instance in accordance with the particular local and remote interface which cannot be obtained directly from the container. Local and remote interfaces themselves provide access to SB instances for making use of the encapsulated application logic. Business and local business interfaces represent one of the new concepts of EJB 3.0. They solely provide access to the encapsulated application logic of a session bean and are considered as preferable choice for provided interfaces of SBs. References to SB instances based on local business or business interfaces can be requested directly from the container.

3.1.1.3. Component Specification

During implementation developers can integrate *Metadata Annotations* into the source code of beans for specification purposes. Alternatively or in combination it is also possible to provide a component specification in form of a DD. Such a DD can address the same aspects as annotations. Additionally, it can cover specifications on component-level. In case certain annotations refer to the same aspects as parts of the DD, the content of the DD is privileged. Hence, it is possible to adjust the specification of a component and its constituting beans without the need to manipulate their source code. The EJB standard also allows the integration of arbitrary artifacts into an ejb-jar file. Nevertheless, these artifacts are not explicitly supported by the EJB standard. Consequently, container implementations do not need to support custom enhancements.

In relation to the concept of component orientation the EJB standard considers Java interfaces as types of access points to SB instances. Thus, the EJB standard concentrates on syntactic aspects while not explicitly addressing semantic or non-functional ones. As it is possible to integrate arbitrary artifacts into ejb-jar files, it might be meaningful to enrich interface specifications and components with semantic and non-functional aspects, as well as with human readable documentations such as an API specification. Nevertheless, these would not be considered or supported by a container.

The EJB standard supports the specification of environment dependencies such as required SB interfaces, Web Service references, or references to the EIS-tier. As this thesis concentrates on the internals of the business tier, only required interfaces are considered in the remainder of this thesis. Please refer to chapter 16 of the EJB standard for further details regarding other dependencies (cf. [58], p. 401ff).

In addition to provided interfaces and environment dependencies the EJB standard allows the parameterization of enterprise beans through so-called *Simple Environment Entries* (SEE). Such an entry can be defined based on the Java types `String`, `Character`, `Integer`, `Boolean`, `Double`, `Byte`, `Short`, `Long`, and `Float`. Additionally, beans can be configured through parameterization with respect to different facilities defined in the standard (see section 3.2).

3.1.2. Component Systems

In this section the foundations of EJB-based systems and the mechanisms for their establishment are presented. Section 3.1.2.1 covers a discussion of the naming facility which builds the foundation for publication of and connection establishment between deployed components inside a container. Subsequently, an overview of the opportunities for the adaptation of components during deployment are addressed in section 3.1.2.2. Finally, the concept of *Dependency Injection* is presented in section 3.1.2.3.

3.1.2.1. Component Publication and Connection Establishment

In the EJB standard, a naming facility based on the *Java Naming and Directory Interface* (JNDI) [148] is considered as foundation for the publication of component access points, as well as for the establishment of connections to instances of the corresponding beans. JNDI provides an abstract Java API for the interaction with naming and directory services. In the context of EJB, JNDI is used for name-to-object resolution. Therefore, a container must provide an implementation of JNDI to deployed components and instances of the constituent enterprise beans which only covers the corresponding parts of the JNDI standard. In contrast, a manipulation of entries at runtime should not be supported. The corresponding naming schema is intended to also contain references to resources which are accessible from inside a container such as JMS-based message destinations. Nevertheless, the creation and binding of resources itself is not addressed by the EJB standard, but is left open to container implementations. During deployment enterprise beans are bound to so-called *Mapped Names* inside the name schema. Regarding session beans these names can be used by clients to request references based on home and business interfaces. A reference based on a home interface is not directly connected to an SB instance, but might be used to request references to instances based on the corresponding remote interface. References based on business interfaces might be resolved also and might be used for interactions directly. References based on local home and local business interfaces are not published inside the naming facility. In contrast, a connection establishment based on these interfaces is only supported through component adaptation, as discussed in the following section. For message-driven beans the submitted mapped names must be bound to access points of existing message destinations for which the MDBs should be registered as listeners. These mapped names can be used by clients to establish connections to the corresponding message destinations at runtime.

The deployment of EJB modules should be supported by container vendors through appropriate tools. Their concrete realization is not addressed within the standard. After the deployment of an EJB component a manipulation of bindings of constituent beans is not supported. Consequently, the publication of beans at a mapped name is limited to the deployment of components.

The naming facility can be accessed from inside or outside an EJB container. The EJB standard demands that an implementation must only support the usage of the naming facility to look up objects through submitting their mapped names, while not allowing the manipulation of entries. Thus, users should not be able to add, remove or change name-to-object bindings. The usage of the naming facility from inside component source code enables, amongst others, a pull-oriented connection establishment to SB instances and message destinations served by MDB instances. If a name is resolved the container must behave in accordance with the discussion of section 3.1. Therefore, it must ensure that each request for a stateful SB yields a reference to a new instance. For stateless SBs it might realize references through a kind of proxies allowing it to schedule method invocations to idle instances. Regarding MDBs the container must provide access to the corresponding message destination. The EJB standard does not restrict the usage of the naming facility to previously defined references, that is, bean instances might obtain references through direct lookups. Therefore, a component specification does not necessarily need to be complete with respect to the declaration of environment dependencies. Consequently, bean instances might establish connections to their execution environment which are not reflected in the corresponding component specification through environment dependencies.

For external clients the EJB standard demands that an EJB container must at least allow interactions with its naming facility in accordance with the *CosNaming* specification [122]. Furthermore, the interaction with SB instances should at least be possible through *CORBA/IIOP* [121]. This should enable the interaction among containers from different vendors. Furthermore, the usage of EJB-based enterprise systems should be possible for clients not being implemented on top of the Java platform as long as they are able to interact through the above stated specifications. Nevertheless, the standard only demands interoperability for accessing session beans through home and remote interfaces (cf. [58], p. 382 - 386 and p. 391f).

3.1.2.2. Component Adaptation

The naming facility discussed in the previous section is called *global* namespace, because it is accessible from inside the source code of components, as well as from clients outside the EJB container. In addition to this namespace an individual *local* namespace is established for each deployed bean containing, amongst others, entries for all of its declared dependencies and simple environment entries. During development, assembling, or deployment of an EJB component, entries of local namespaces might be manipulated. For environment dependencies local names are mapped to global names. The specification of links from local names to global names during development or assembling does not include the actual link establishment, but represents a proposal for later deployment. If the global namespace is manipulated in a deployment context, for example, through changes of the set of deployed components, this has direct impact on affected mappings inside local namespaces. References to local home and local business interfaces are directly bound to the local namespace of enterprise beans. This is only possible for interfaces which are provided by beans inside the same EJB component. Values for SEEs are also directly integrated into the local namespace of a bean and do not have corresponding elements in a global namespace. Both, entries for dependencies and SEEs cannot be manipulated after deployment. Locally bound names of a bean can be looked up through a reference to the global namespace in that they are accessible through a special prefix[2]. Nevertheless, these entries are only accessible from inside the execution context of corresponding bean instances. For all other execution contexts they are neither visible

[2] `java:comp/env`

nor accessible.

Summarizing, parameter and compositional adaptation of EJB components can be performed through manipulation of the local namespace of the constituent beans during development, assembling, or deployment. In relation to the discussion in section 1.2.2 the EJB standard supports static composition only. Therefore, it does not consider opportunities for seamless reconfiguration of EJB-based enterprise systems at runtime. Nevertheless, it is possible to exchange deployed components without the need to access other referencing deployed components. This can be performed through a combination of undeploying the original deployed component and subsequently deploying the replacing component while binding the replacing beans to the mapped names of the original ones. Such a proceeding might lead to system disruptions and connection losses, because the affected names of the global namespace would not be bound during the timespan between undeployment and deployment. Consequently, resolutions for these names could not be fulfilled. Additionally, existing connections to bean instances belonging to the undeployed component would get lost.

3.1.2.3. Dependency Injection

In addition to the usage of the global and local namespaces for pull-oriented obtainment of SEE values and connection establishment it is possible to specify that values and references should be injected into bean instances during construction. These injections are summarized under the term *Dependency Injection* (DI) which is one of the new concepts of EJB 3.0. It facilitates the tasks of component developers, because they do not have to consider the actual resolution of references or SEE values inside source code, but can rely on their availability after instance construction.

After instantiation of an enterprise bean a container must inject values for SEEs and references for dependencies, if specified through annotations or the corresponding DD. The injection itself might either be performed through setting field values or through invocation of special *set*-methods following the naming conventions of *JavaBeans* properties (cf. [74], p. 55).

3.1.3. Bean Instance Life Cycles

Each bean instance passes through a type-specific life cycle. For instances of stateless SBs and MDBs the particular life cycles are almost equal. They mainly consist of three states and transitions initiated by a container as depicted in figure 3.2.

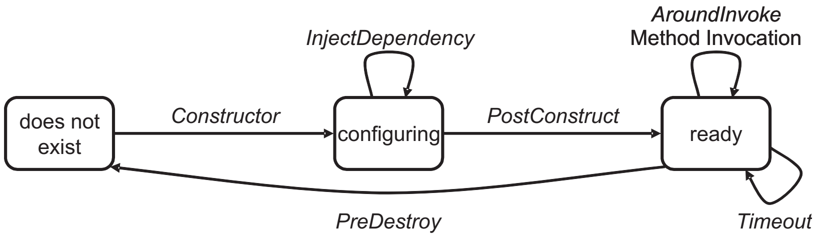


Figure 3.2.: Basic bean instance life cycle (cf. [58], p. 84 and p. 115).

The italic messages in the figure indicate that the corresponding methods are invoked by the container. The life cycle of an instance starts within the state *does not exist* indicating that the considered bean instance does not exist within the runtime environment. Through the invocation of a default *Constructor* the container creates the instance which subsequently is transferred to the state *configuring*. In this state the container must perform *Dependency Injection*, if corresponding targets are defined. Afterwards, it informs the new instance about the end of container initiated configuration through invocation of a *PostConstruct* method, if present. Such a method can be defined either through annotations or in the DD, and must exhibit a special signature. It allows instances to perform their own configuration actions while relying on being properly configured by the container. During execution of *PostConstruct* a bean instance might interact with its environment which is not allowed during the *configuring* state. The bean instance is afterwards transferred to the *ready* state. If no *PostConstruct* method is defined for a bean, a container can directly transfer the new instance from *configuring* to *ready* after finishing its configuration demands, if any. This alternative is not covered in the figure. Within the *ready* state the container might forward *Method Invocations* of clients to SB instances or received messages to MDB instances. Furthermore, *Timeout* methods, relating to the timer service, can be invoked by the container (see section 3.2.3). When a container does not need an instance anymore it can destroy the instance, as described in section 3.1.1. Before doing so it must invoke a *PreDestroy* method allowing the instance to perform cleanup actions, if specified. The definition of such a method is analog to that of a *PostConstruct* method. If no *PreDestroy* method is defined for a bean the container can directly destroy the corresponding instance which is not depicted in the figure.

An enterprise bean can additionally define an *AroundInvoke* method, analog to *PostConstruct* and *PreDestroy* methods. If defined, the container must forward the control flow of client initiated *Method Invocations* or messages to the *AroundInvoke* method before the original processing of the method or message. During execution of an *AroundInvoke* method the instance might access and manipulate the parameters of the original invocation and gain reflective insight into the target method, for instance, to discover its signature. The same holds for a received message analogously. After performing all desired actions the invocation might be forwarded for execution of the original method. After completion the control flow is returned to the *AroundInvoke* method, allowing inspection and manipulation of the return value, if any. During execution of an *AroundInvoke* method it is also possible to prevent the target method from being executed.

The life cycle of stateful SB instances can be interpreted as an extension of the life cycle discussed above. Figure 3.3 depicts the corresponding states and transitions. The upper states in the figure (*does not exist, configuring* and *ready*), as well as the corresponding transitions are equivalent for all bean life cycles.

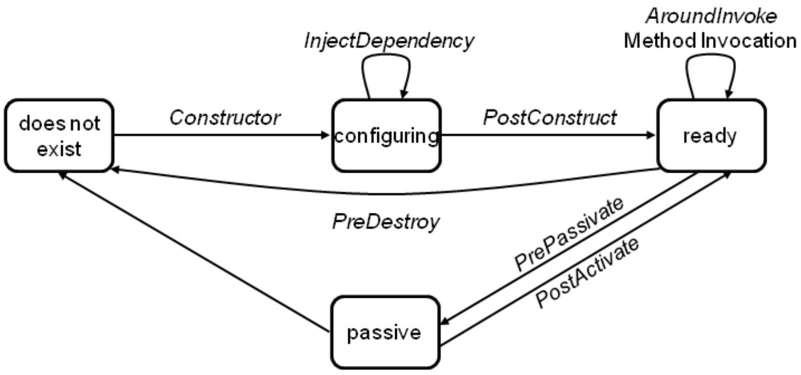


Figure 3.3.: Stateful SB instance life cycle (cf. [58], p. 74).

As for MDBs and stateless SBs a container can choose *ready* instances of stateful session beans for removal from main memory. Nevertheless, it might be meaningful not to remove them completely, but to keep them for later reactivation due to their client-specific conversational state. Therefore, the EJB standard defines a fourth state called *passive*. In this state a stateful SB instance does not exist inside the main memory of a container anymore, but is persisted to secondary storage. Before a container persists an instance, it must invoke a *PrePassivate* method on the corresponding instance, if defined. Such a method can be declared through annotation or inside the DD and must have a specific signature. It is mainly intended to allow instances to prepare their serialization, for example, by closing open resource connections. When a client tries to interact with a *passive* instance, the container deserializes the instance and subsequently invokes a *PostActivate* method on it, if defined. This method represents the counterpart for a *PrePassivate* method and allows the instance to perform preparation actions for being ready to serve client-based method invocations again. A container might also choose *passive* instances for removal from secondary storage. This is shown in figure 3.3 on page 71 through the transition from the *passive* to the *does not exist* state. This transition is not observable by affected bean instances.

3.1.4. Interceptors

Bean instances might supervise their life cycle and the corresponding transitions through method invocations performed by the container, as discussed in the previous section. Additionally, the EJB standard allows the definition of interceptor methods on distinct classes which can be used as interceptors. For their application they must be attached to enterprise beans either through annotations inside the bean source code or through the DD. An arbitrary number of interceptors might be attached to an enterprise bean on different levels, for example, for a single method, a bean as a whole, or all beans of the corresponding module. They are organized in a bean-specific interceptor chain defining the order in which they access an incoming method invocation. At runtime an interceptor instance is bound to a single bean instance, that is, it has the same life cycle regarding all state transitions. An interceptor instance intercepts the affected method invocations on the associated bean instance only. Furthermore, an interceptor shares the local namespace of the associated bean.

Figure 3.4 on page 73 shows the interactions between a container, an interceptor instance, and an SB instance during processing a method invocation of a client, as defined by the EJB standard. For the figure it is assumed that one interceptor (*MyInterceptor*) is attached to an SB (*MySessionBean*) and that there is no *AroundInvoke* method defined on the bean itself. The figure does only represent a schematic view on the different interactions. They do not necessarily need to be realized exactly the same way by a container implementation. Nevertheless, the incoming and outgoing method invocations at the interceptor instance and the bean instance are defined in the standard as shown in the figure.

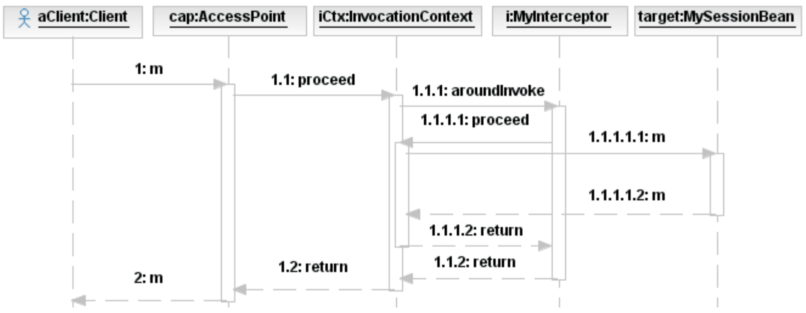


Figure 3.4.: Schematic view on EJB Interception.

The interactions start with a method invocation of a client arriving at the access point (*cap*) on container side (*1:m*). After identification of the target SB instance the container must provide an instance of a so-called *InvocationContext* (*iCtx*) which is passed as a parameter to interception methods. This context provides, amongst others, access to different aspects of the corresponding call for inspection and manipulation purposes such as its parameters. Furthermore, it enables interceptor instances to gain direct access to the target bean instance. Afterwards, the *aroundInvoke* method is invoked (*1.1.1:aroundInvoke*) on the interceptor instance (*i*) allowing it to perform the desired actions. After all desired actions before the execution of the target method were performed, the interceptor must invoke the *proceed*-method (*1.1.1.1:proceed*) on the invocation context instance to forward the invocation. On receiving the invocation *iCtx* interacts with the next interceptor of the corresponding interceptor chain in the same way as discussed before (*1.1.1* and *1.1.1.1*). When the last interceptor instance is passed the container invokes the *aroundInvoke* method on the target bean instance (*target*), if defined. These steps are not addressed in the figure due to the underlying assumptions and for clarity reason. Finally, the original method invocation is performed on *target* (*1.1.1.1.1:m*).

After its completion (*1.1.1.1.2:m*) the *aroundInvoke* invocation on the bean instance and the interceptor chain are passed in reverse order through returning from the *proceed* invocations on *iCtx* (*1.1.1.2:return*). This allows *target* and interceptor instances to perform actions after execution of the target method, for example, to react to the return value or an exception. After finishing execution *i* returns from the *aroundInvoke* invocation (*1.1.2:return*). When the last interceptor instance has returned the (potentially adjusted) result of the invocation is returned to the *cap* (*1.2:return*) and finally handed back to *aClient* (*2:m*). During an invocation on an interceptor instance the instance might throw an exception or return without invoking *proceed* on *iCtx*. If the exception is thrown before invoking the *proceed* method or the invocation of *proceed* is omitted the method *m* is not invoked on *target*.

Summarizing, interceptor instances gain full control over the control flow before and after method invocations. This allows them to perform any desired inspection and manipulation actions regarding parameter values or the return value. In this context, they might also throw exceptions or prevent the original call from being executed on the target bean instance. Finally, they are able to interact with the target instance directly and are provided with the same opportunities for interaction with the container as the target bean instance. An analog proceeding is applied to invocations during life cycle transitions, that is, *PostConstruct*, *PreDestroy*, *PrePassivate*, and *PostActivate*.

3.2. Container Facilities

Beyond the core component model the EJB standard includes the definition of different facilities supporting the development of components and providing a common ground for execution environments of deployed EJB components. The following sections provide an overview of those facilities which are relevant for this thesis. First, the transaction support, as provided by the standard, is presented in section 3.2.1. Afterwards, security aspects are discussed in section 3.2.2. Finally, the timer service which must be provided by containers is presented in section 3.2.3. The sections do not aim to provide a comprehensive introduction into the considered facilities, but only give a short overview. Please refer to the corresponding sections of the EJB standard for further details.

3.2.1. Transaction Support

According to the EJB standard transactions are intended to

> *"[...] free the application programmer from dealing with the complex issues of failure recovery and multi-user programming. If the application programmer uses transactions, the programmer divides the application's work into units called transactions. The transactional system ensures that a unit of work either fully completes, or the work is fully rolled back. Furthermore, transactions make it possible for the programmer to design the application as if it ran in an environment that executes units of work serially."*
> (cf. [58], p. 315)

The standard considers transactions to disburden component developers through addressing situations potentially demanding for the cancellation of effects of previous actions. This might become necessary if failures are identified during execution or if certain required conditions are not fulfilled. In this context, transactions are seen as work units which must be fulfilled in an atomic fashion, that is, they must be either completed successfully as a whole (*commit*) or their effects must be undone (*rollback*). Transactions are considered in the EJB standard with respect to resources accessed by bean instances. The underlying transaction model is a flat model which does not support nested transactions. Transaction support allows developers to design and realize components without the need to consider concurrent access to commonly used resources like data bases, because transactions are isolated from each other. Nevertheless, the concrete support for different isolation levels is left open to resource providers. Bean instances are responsible to configure appropriate isolation levels through the corresponding resource API. In combination with the non-reentrancy property of enterprise beans concurrency does not need to be explicitly considered during component development beyond the specification of transaction properties and the configuration of resource specific isolation levels.

The EJB standard defines two types of transaction demarcation, namely *bean-managed transaction demarcation* and *container-managed transaction demarcation*. The applied demarcation type is defined for an enterprise bean as a whole.

Bean-managed transaction demarcation (BMTD): This type of transaction demarcation allows a fine-grained control over transactions from inside the source code of enterprise beans. This might be performed through application of the *Java Transaction API* (JTA) [47]. This API supports the beginning and committing of transactions, as well as the initiation of rollbacks. Furthermore, the current state of a transaction can be supervised. Through this type of transaction demarcation it is possible to span a transaction across multiple invocations on a stateful SB instance. Moreover, it also allows to split the source code of a method into different transaction units being executed in the context of separate transactions. It is not possible to join transactions started by clients of a bean instance. Such transactions are suspended before execution of an original invocation and is resumed afterwards.

Container-managed transaction demarcation (CMTD): This type is the default if no demarcation specification is provided for an enterprise bean. In contrast to the previous type, it allows a declarative specification of transaction support for interactions with bean instances trough annotations or through the DD. For each method of a bean, which is used for interaction with clients, a transaction attribute might be specified. This attribute determines the transaction semantics for an invocation of the particular method. The supported values for the transaction attribute are:

- **Never**: The corresponding method is executed without any transaction support. Furthermore, a client must invoke such a method without submitting a transaction context.
- **Mandatory**: A client must invoke such a method submitting an existing transaction context. The method itself is executed in the context of the client transaction.
- **Required**: The method demands for a transaction context for its execution. If a client transaction is submitted this transaction is joined. Otherwise, the container must start a new transaction for method execution. If no transaction attribute is specified for a certain method, the standard defines *Required* as default transaction semantics in case CMTD is applied.
- **RequiresNew**: A corresponding method must be executed within a new transaction context. An existing client transaction is suspended and resumed after method execution, if any.
- **NotSupported**: The invocation of a corresponding method is executed in an unspecified transaction context. Client transactions are suspended before starting execution and resumed after finishing, if any.
- **Supports**: If a client invokes a method supporting transactions, the container must execute the method in the transaction context of the client. Otherwise, the method must be executed in an unspecified transaction context.

For *mandatory*, *required*, and *requiresNew* the container must forward the corresponding transaction context to accessed resources during execution if they support transactions. *Never* and *notSupported* demand that no transaction context is forwarded. The forwarding semantics of *supports* depends on the transaction context of the client invocation. If a transaction context exists, it is submitted during resource usage, if possible. Otherwise, no transaction context is forwarded. Inside the source code of a method it is possible to request the current state of an active transaction, if any. Furthermore, it can be declared that a transaction should be rolled back. Stateful SBs might implement the `javax.ejb.SessionSynchronization` interface for being notified about transaction synchronizations, for example, that a transaction is about to be started or stopped[3], or that the startup or stop of a transaction has been finished.

For both demarcation types the EJB standard does not demand that the state of an affected bean instance itself is subject to automated transaction management. Therefore, in case of a rollback the instance state itself is not reset. In this context, the bean developer is responsible to implement appropriate mechanisms if a rollback of an instance state is necessary. Instances realizing BMTD are controlling transactions. Therefore, they can supervise changes in a transaction state and can react appropriately. For CMTD the `javax.ejb.SessionSynchronization` interface enables stateful SB instances to also supervise state transitions and perform a rollback of their state, if necessary. As stateless SBs and MDBs must be implemented in a way that ensures equivalence of instances, a rollback of their state is not necessary.

Summarizing, the EJB standard supports transactions through automated management of transaction execution. In this context, it can be defined how transactions should be supported for a certain bean. The concrete realization at runtime lies within the responsibility of the container. BMTD allows fine-grained control of transactions from inside source code. CMTD might free developers completely from addressing transactions. Nevertheless, bean instances based on CMTD might also supervise and influence transaction execution, if desired. If the state of a

[3] Commit or rollback

bean instance itself should be subject of transaction rollbacks this must be implemented by bean developers.

3.2.2. Security

The EJB standard provides a security model which allows to address security policies in an abstract manner. In this context, component developers do not need to consider the original authentication and authorization process inside the source code of beans. These aspects lay within the responsibility of containers and are transparent to deployed components at runtime. Moreover, component developers can abstract from potential execution environments of their components during development with respect to the concrete security configuration such as different security realms, or existing users and groups.

Figure 3.5 on page 80 presents the security model as described in the standard which is based on so-called *Roles* and *Permissions*. A role represents a logical group of entities defined on component level. A permission is the right to interact with instances of a certain bean either through method invocations (SBs) or through sending messages (MDBs). A permission is related to at least one method of an enterprise bean and is assigned to at least one role. Roles might be associated with multiple permissions and vice versa. Roles and permissions are defined for components and beans either through metadata annotations or inside a DD. In this context, single entities associated with a certain role are not considered. In combination roles and permissions define a logical view on security aspects which is independent from a concrete execution environment. Moreover, fine-grained access control policies might be specified on method level.

During deployment roles must be mapped to the security settings of the target execution environment. A container security model based on *Domains*, *Realms*, *Groups*, and *Accounts* is considered in the standard. Never-

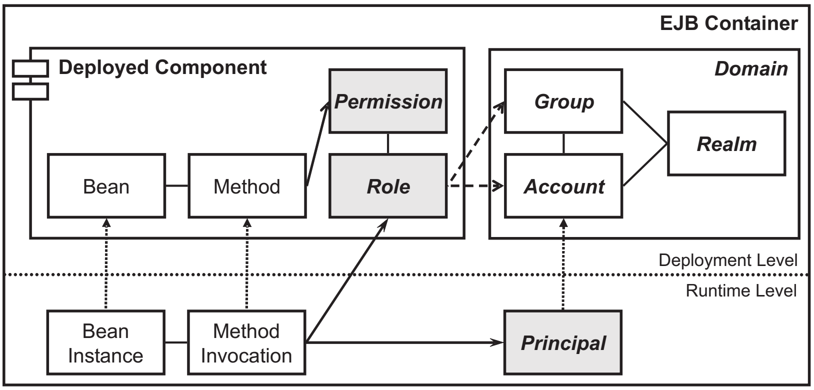


Figure 3.5.: The EJB security model

theless, this model does not necessarily need to be realized by container implementations, but represents a blueprint for container providers. A security domain is considered as the topmost concept in a hosting scenario. Inside a domain there might exist multiple realms, for instance, to distinguish different categories of users such as customers or suppliers. A realm might internally be organized in groups of accounts for administration purposes. In this context, an account represents a single logical user which can be authenticated individually at runtime. The standard proposes that roles are mapped to groups, but also mentions the opportunity to assign a role to a single user. These two alternatives are represented in the figure through the dashed associations.

At runtime a container must guarantee that the security requirements of deployed components are fulfilled. It must ensure that users are correctly authenticated against their accounts. In this context, an authenticated user is called *Principal*. Furthermore, the container must guarantee that principals are only allowed to interact with bean instances in a way that is compliant with the roles defined inside the corresponding component and the mapping to the container security settings.

During method execution a container must provide bean instances with information about the interacting principal. Therefore, the container enables bean instances to request the account name of the principal, as well as to determine if the principal is (indirectly) associated with a certain role.

According to the standard principal information must be forwarded along nested method invocations inside a container. Nevertheless, there are situations conceivable where a change of roles might become necessary[4]. For such a scenario it is possible to define that instances of a bean should act in a specific role at runtime. Therefore, the standard provides the opportunity to define that a bean instance requires certain roles for incoming method invocations while acting in a different role when performing method calls itself. This is not depicted in figure 3.5.

3.2.3. Timer Service

As part of the EJB standard a timer service is defined. This service is intended to support stateless SB and MDB instances with the opportunity to start timers for later execution of callback methods. These can be scheduled after an elapsed duration, at a certain time, or at recurring intervals. For management purposes bean instances can locate timers which are currently associated with the corresponding bean. These can be inspected or canceled individually. A reference to the service can be obtained by instances through dependency injection or can be requested from the container.

Callbacks are not necessarily executed on that bean instance which initiated the scheduling. Instead of that, the container can choose an arbitrary instance of the corresponding bean for execution. In this context, a bean instance can provide a serializable object during timer scheduling. This

[4] Such a situation might, for example, be given if a bean should be accessible for principals which are in the role of a *customer*. Internally, instances of that bean should interact with a warehousing system. Nevertheless, the warehousing system itself should not be accessible for *customer*s directly. Therefore, the bean must interact with the warehousing system in a different role.

object is handed to the instance which executes a callback and is intended to contain context information.

The timer service is not intended to provide real-time properties, that is, it does not guarantee that a timeout callback is started at exactly the specified time. In contrast, it is intended to support long-lived business processes. The EJB standard demands that timers survive crashes and shutdowns of Java EE servers.

Timer-method invocations represent the fourth relevant situation during which bean instances can interact with their environment. The other three situations are the invocation of life cycle calls, the receipt of a JMS message by an MDB instance, and the execution of an invocation upon an SB instance. Within all of these situations bean instances might interact with their environment and other bean instances. Therefore, the Timer Service must be considered by an AC infrastructure at least for controlling interactions in a managed system. This is especially relevant during the realization of dynamic adaptation (cf. section 5.6). The construction of bean instances itself does not belong to the group of relevant situations, because during constructor execution bean instances must not interact with their environment (cf. [58], p. 79, p. 88 and p. 117).

3.3. Role Model

As part of the EJB standard seven different roles are specified. For each of these roles the standard defines responsibilities regarding the different aspects to address during software and system life cycles. In combination these roles give an impression of how the development and application of EJB-based enterprise software and systems are envisioned within the EJB standard. Although the standard does not prescribe these roles, their definition and the corresponding task assignments provide a meaningful foundation regarding development and execution aspects to consider for an AC infrastructure. The different roles are shortly introduced in the remainder of this section.

EJB Container Provider: The responsibilities of container providers lay within the provision of container implementations and tools for supporting deployment preparation and installation of EJB components. For bean instances a container provider must ensure that a container implementation provides all required APIs, services, and facilities. Moreover, a container must show a standard compliant behavior at runtime, for example, regarding the life cycle management of bean instances.

EJB Server Provider: An EJB server provider is responsible for the implementation of a low-level infrastructure of EJB containers such as distributed transaction management. The standard does not define any relationships between servers and containers, because it assumes them of being deeply integrated and being provided by the same vendor.

Persistence Provider: An implementation of the object/relational mapping, as defined in the *Java Persistence API* [59], is provided by a persistence provider. This also covers the realization of a scalable and transaction-enabled execution environment for persistence management. Additionally, a persistence provider supports deployers with tools for preprocessing persistence entities if this is necessary for their mapping to an underlying data source.

Enterprise Bean Provider: The role of an enterprise bean provider, or *Bean Provider* for short, is taken on by software developers. They implement the source code of single enterprise beans and define their external specification through annotations or a DD. Such a specification might cover all aspects discussed in the context of EJB component specifications in section 3.1.1.3 and aspects of container facilities. Nevertheless, bean providers do not configure beans for concrete execution environments.

Application Assembler: An application assembler combines the results of development performed by bean providers. He or she integrates different beans into a component and configures it internally. This might, for instance, include the declaration of connections between provided and required interfaces of constituent enterprise beans. Consequently, the outcomes of assembling are EJB-based components (ejb-jar files). An application assembler considers beans from an external view. Therefore, he or she does not need to know their concrete realization, but only their externally observable properties. An application assembler is considered in a broader context with respect to other parts of Java EE 5. These aspects are not relevant in this thesis.

Deployer: A deployer adjusts an ejb-jar file for its integration into a container. Therefore, he or she must configure the component to be deployed in order to fulfill the external requirements of the constituent beans, for instance, through specifying connections for required interfaces or through mapping security roles to the underlying security system of the target container. Furthermore, he or she might set or change values for simple environment entries. Subsequently, a deployer prepares the component to deploy for its target environment, for example, through the execution of preprocessing tools, if required. Finally, he or she deploys the component into the target container in accordance with instructions stated by application assemblers, if such exist.

System Administrator According to the EJB standard a system administrator is responsible for the management of the IT-infrastructure covering all aspects of computing systems and underlying resources. Nevertheless, the standard concentrates on those aspects which directly relate to the supervision and adjustment of EJB-based systems and the corresponding resources.

The first three roles in combination are responsible for the provision of the software infrastructure of runtime environments of EJB-based systems. EJB server providers and EJB container providers directly address the execution of bean instances and interactions inside the business-tier, as well as interactions with the web-tier. This also covers supported APIs, services, and facilities. Moreover, access points for external clients must also be provided. A persistence provider provides an implementation of an abstraction from the EIS-tier. Bean provider and application assembler concentrate on the development of EJB-based enterprise software. While a bean provider is responsible for the development of component building blocks, an application assembler constructs components out of them. Finally, deployers and system administrators address the management of EJB-based component systems. In relation to section 1.2.2 a deployer is directly concerned with adaptation of component systems, because he or she manipulates the architecture of a component system through the adjustment of bean dependencies during deployment, and through the deployment of components and the undeployment of deployed components. Furthermore, he or she performs parameter adaptation by setting simple environment entries during deployment. System administrators are responsible for the management of the underlying infrastructure such as containers, Java EE servers, or database management systems.

3.4. Related Management Standards

The EJB standard does not address the original deployment of EJB components. Instead of that, the standard is limited to the configuration of components through annotations and DDs as preparation for deployment. Furthermore, execution environments for deployed EJB components are also not addressed with respect to their inspection and manipulation. The standard defines the behavior of containers regarding the interaction with bean instances only, for instance, the provision of facilities or the need for guaranteeing non-reentrancy of bean instances. Consequently, the EJB standard does not support deployers and system administrators with specific requirements for fulfilling their tasks regarding the interaction with a container.

In the context of Java EE 5, there do exist two standards addressing the above stated areas of manipulation and inspection. These standards are not limited to Enterprise JavaBeans 3.0, but address all aspects of Java EE

5 also covering, for instance, web-tier components. The two standards are introduced briefly in the following, because they might provide links for the (autonomic) management of execution environments of EJB-based component systems.

3.4.1. The Java Enterprise Edition 5 Deployment API Specification

The Java Enterprise Edition 5 Deployment API Specification, Version 1.2 is a standard which is intended to unify the inspection and deployment of Java EE-based software also covering EJB components. The standard was first released in July 2002 as JSR 88 [64]. It does not necessarily need to be supported by all Java EE platforms (cf. [140], p. 115). In order to reach the goal of unification, the standard defines two APIs.

The first API should be realized by so-called *Java EE Product Provider*. These are the vendors of Java EE products like Java EE servers. The API should support deployers through opportunities for inspection and manipulation of the set of deployed components of a Java EE server. In this context, there are different actions considered for manipulating the life cycle of deployed components, as depicted in figure 3.6.

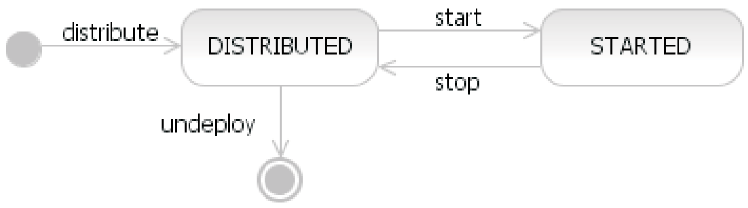


Figure 3.6.: Life cycle of deployed components according to JSR 88.

The life cycle of a deployed component starts with its distribution. During the *distribute* transition the component is validated, preprocessed if necessary, and finally transferred to its target execution environment. A deployed component in the *DISTRIBUTED* state does exist inside a container, but cannot be accessed by clients. In order to allow client access, it must be transferred (*start*) into the state *STARTED*. When a deployed component is not needed anymore it might be transferred (*stop*) to the state *DISTRIBUTED* again and finally be undeployed. During the *undeploy* transition a deployed component is removed from the container and does not exist anymore after completion of the transition. During the whole life cycle of deployed components there is no configuration considered in the JSR 88 standard.

The second API should be realized by so-called *Tool Providers*. These represent the vendors of tools for supporting configuration and deployment of Java EE-based components. In the context of JSR 88, the providers are responsible to support a unified API for inspection of components which is independent from concrete requirements of different execution environments. The API is rather generic and orientates itself at the XML structure of DDs. In this context, nodes are represented through so-called *DDBeans* which might be used for inspection and navigation. Nevertheless, a manipulation of components or applications through the API is not considered.

3.4.2. Java 2 Platform, Enterprise Edition Management Specification

To support system administrators performing their tasks the *Java 2 Platform, Enterprise Edition Management Specification* [81] defines a unified model for managing and monitoring Java EE servers, their deployed components, and managed resources. It was released as JSR 77 [81] in July 2002. The standard represents a required part of Java EE 5 and must therefore be supported by all Java EE platforms (cf. [140], p. 115). JSR 77 defines different interfaces for interacting with a Java EE server, also including an EJB-based access point.

The standard specifies a meta model which is intended to represent the basic concepts of the Java EE 5 domain. This meta model is rather rudimental, because it only defines classes for the different concepts discussed below including basic attributes and operations. It is intended to be extended by vendors with product-specific aspects. As a central concept of the standard so-called *Managed Objects* are considered. A managed object is intended to represent a server itself or included objects which might be subject to management. Management can be performed through inspection and manipulation of attributes of managed objects. Nevertheless, the standard does not require any of the attributes to be mutable. JSR 77 addresses four main areas, namely *Discovery*, *State Management*, *Events*, and *Performance Monitoring*.

Discovery relates to the identification and navigation along managed objects. In this context, deployed EJB components are considered as `EJBModules` containing at least one `EJB`. `EJBs` are further specialized according to the different enterprise bean types. There are no special attributes defined for components and beans. Therefore, JSR 77 only covers navigation regarding the structure of EJB components.

For some managed objects *State Management* can be performed regarding their life cycle, for example, for a Java EE server as a whole or for single deployed components. The standard includes a very basic model which allows to start or stop those objects from inside source code and to observe the process of state transition, as well as its success or failure. State management of deployed EJB components does not need to be supported.

The *Events* part of JSR 77 addresses the provision of notifications about the occurrence of different events within a management environment. A notification contains different attributes such as an identifier of the managed object at which the event occurred or the corresponding event type. The standard defines a basic set of event types which mainly relate to state transitions of managed objects or changes of their attributes. Event support for EJB components is not required.

Under the term *Performance Monitoring* JSR 77 subsumes different types of fundamental statistic measures for managed objects. Regarding managed `EJBs`, the standard supports different statistics which mainly cover the number of bean instances in different life cycle states.

In combination the four areas addressed by JSR 77 provide a fundamental set of inspection facilities for Java EE servers. In this context, notifications about relevant events might be obtained. Furthermore, opportunities for setting attributes of managed objects allow their manipulation. Finally, it is possible to start and stop managed objects if this is supported by the corresponding Java EE server. Therefore, system administrators are equipped with a unified management meta model. Nevertheless, the standard is rather general and can be seen as foundation for vendor-specific extensions.

3.5. Summary

The EJB standard provides a specification for component based enterprise software and systems on top of the Java platform.

The software life cycle of EJB components is addressed through the definition of detailed requirements and guidelines regarding the internals of components. Bean providers can rely on different assertions for the later runtime of deployed components such as guaranteed non-reentrancy, or life cycle management and observation of bean instances. Additionally, the facilities required by the standard allow bean providers to concentrate on the core business logic. The concept of interceptors further supports separation-of-concerns. Finally, the use of deployment descriptors or annotations for configuration purposes frees application assemblers from the need to inspect the source code of components.

The externally observable aspects of components are specified through environmental dependencies of the constituent enterprise beans. Furthermore, simple environment entries represent opportunities for parameterization. These might be inspected and configured by deployers during the deployment phase of the system life cycle. The management phase is not addressed for deployed components, that is, the standard does not support inspection and adaptation of deployed components. The two additional standards discussed in the previous section address the management phase of Java EE-based enterprise systems. While JSR 88 provides effectors for the management of deployed components of a Java EE server, JSR 77 delivers sensors for inspection of servers and deployed components. In combination the standards provide a rudimentary foundation for system management. Nevertheless, they do not cover any opportunities for structural and behavioral inspection of a managed system itself. Furthermore, they do not support dynamic adaptation at runtime. JSR 88 and JSR 77 are not part of the EJB standard, but are integrated in the broader context of Java EE 5.

Regarding the concept of component orientation the EJB standard provides a sound foundation for the software life cycle of component-based enterprise software. Moreover, the underlying component model allows a clear identification of component boundaries, and opportunities for parameter and compositional adaptation. In this context, components are clearly distinguishable from each other. Their constituent enterprise beans provide an appropriate foundation for addressing the internals of components with respect to adaptation and for controlling the runtime behavior of deployed components. Therefore, this component model is considered as promising foundation for the realization of an AC infrastructure although it does not cover any semantic or non-functional aspects as part of component specifications.

The EJB standard does not define any management aspects beyond the deployment phase of system life cycles. Especially these aspects are of vital interest to apply the vision of autonomic computing to EJB-based enterprise systems. For this purpose especially the JSR 88 is considered as meaningful foundation for the realization of the necessary facilities. It is an integral part of Java EE 5 and can therefore be assumed of being specially designed for the management of deployed enterprise components.

Finally, the practical relevance of the realization of an AC infrastructure for the EJB standard can be estimated of being very high. This is the case because of the broad support by well-known companies and its widespread application. Furthermore, the standard can be regarded of having reached a high level of maturity due to the ongoing development since the final release of *Enterprise JavaBeans Specification, v1.1* in 1999 (cf. [109]) and the practical experiences gained in the meantime. Hence there do exist a couple of conceptual, as well as practical reasons to choose this standard as foundation for the realization of a realistic AC-infrastructure.

4. Introduction to the AC-Infrastructure

The infrastructure presented in this thesis aims to provide a generic foundation for autonomic computing in the context of component oriented enterprise systems. Enterprise JavaBeans, version 3.0, is a broadly accepted and widely used standard for the development of enterprise software based on the Java programming language. Therefore, this standard was chosen as technological foundation for designing and realizing the AC-infrastructure. In section 1.3 different groups of requirements were stated which a generic infrastructure has to fulfill for reaching the goal of providing a comprehensive and generic basis for AC. Against these requirements the presented infrastructure will be evaluated in the end of this thesis (see section 9.1).

The AC-infrastructure consists of three main parts for addressing the different aspects of autonomic management of enterprise systems during their system life cycles. These constituent parts are presented in the subsequent chapters in detail. To illustrate the following discussions section 4.1 introduces a case study which is taken as underlying example for the remainder of this thesis. Section 4.2 presents the top level blueprint of the proposed infrastructure which gives an orientation for the discussion in the following chapters. In this context, the constituent parts are shortly introduced and the relationships among them are pointed out.

4.1. The underlying Case Study

The case study, which builds the foundation of the further discussion, is based on the *Java EE 5 Tutorial* [91]. This tutorial is published by *Sun Mi-*

crosystems and is referred to as foundation for developers to learn about the programming of enterprise applications based on Java EE 5. Therefore, it contains introductions to the different parts of the Java EE platform also covering EJB 3.0. The tutorial can be seen as one of the most comprehensive and widely used introductions to Java EE 5.

Inside the *Java EE 5 Tutorial* the *Duke's Bank Application* – or *Duke's Bank* for short – represents the final case study (cf. [91], p. 1233 - 1258). This case study

> *"[...] demonstrates the way that many of the component technologies presented in this tutorial – enterprise beans, application clients, and web components – are applied to provide a simple but functional application."* (cf. [91], p. 1233)

Consequently, Duke's Bank provides a representative example of how the different parts of Java EE are intended to be used in combination to realize enterprise software and systems. Nevertheless, it is not intended to provide a business ready software which might be applied in a real life context, but a simplified demonstration of the underlying technologies. For the discussion in this thesis it is considered appropriate with respect to the integration of different aspects of Java EE as intended by the specifying parties, and to illustrate and analyze the capabilities of the presented AC-infrastructure. The source code and other necessary artifacts for Duke's Bank are part of the *Java EE 5 Tutorial* download [91]. These were taken as foundation for the case study of this thesis.

Duke's Bank realizes an online banking software. A deployed system allows bank customers to inspect their accounts and histories of transactions through a web frontend. Furthermore, customers are enabled to perform banking transactions among their accounts, that is, transfer money between them. Finally, a bank customer might withdraw money from or deposit money to his or her accounts. Bank customers and their accounts are managed by human administrators through a standalone application client. In this context, information about existing customers might be inspected and updated. Additionally, customers might be created or removed. Account management covers the creation and deletion of accounts, the inspection of account information, and the addition and removal of customers to and from accounts.

As pointed out, Duke's Bank mainly addresses the interplay of different technologies. In this context, it does not aim to cover all aspects of EJB 3.0. To include the missing parts, the functionality of the original case study was extended for this thesis. First of all, bank customers are enabled to transfer money from their accounts to other accounts of which they are not stated as owners. This might also include accounts which are hosted by remote banking systems. Furthermore, customers are allowed to declare standing orders to be executed once a month. To establish connections to remote banking systems administrators are provided with management opportunities to add or remove access point information. Finally, two interfaces were integrated into Duke's Bank which might be used by remote banking systems for transfer purposes. The extensions do not induce any changes of the original enterprise beans or their corresponding configurations as provided by the tutorial.

The original Duke's Bank application was packed for deployment as *Enterprise Archive* (EAR). An EAR is a special jar file which is intended to cover a complete enterprise software and allows its deployment as a whole. The original EAR of Duke's Bank includes different archives of the banking application also covering an ejb-jar file with enterprise beans. All archives might be deployed independently which does not affect interactions of system elements at runtime. Nevertheless, different configuration aspects such as mappings of security settings must be performed in each archive instead of globally inside the EAR. As this thesis solely addresses EJB 3.0 and consequently focuses on ejb-jar files in isolation, the EAR of Duke's Bank was decomposed, and the included archives were treated separately.

Figure 4.1 provides an overview of the different parts of Duke's Bank. The elements with dashed borders represent those parts of the case study which were taken from the original case study of the *Java EE 5 Tutorial*. The symbol in the upper right corner of each element highlights its adjustment demand for application as part of the case study of this thesis. A '0' indicates that no adjustments were necessary, a '∗' shows that the corresponding element was extended and a '+' means that the element was newly integrated. Usage relations between the different elements of a concrete banking system are represented through arrows.

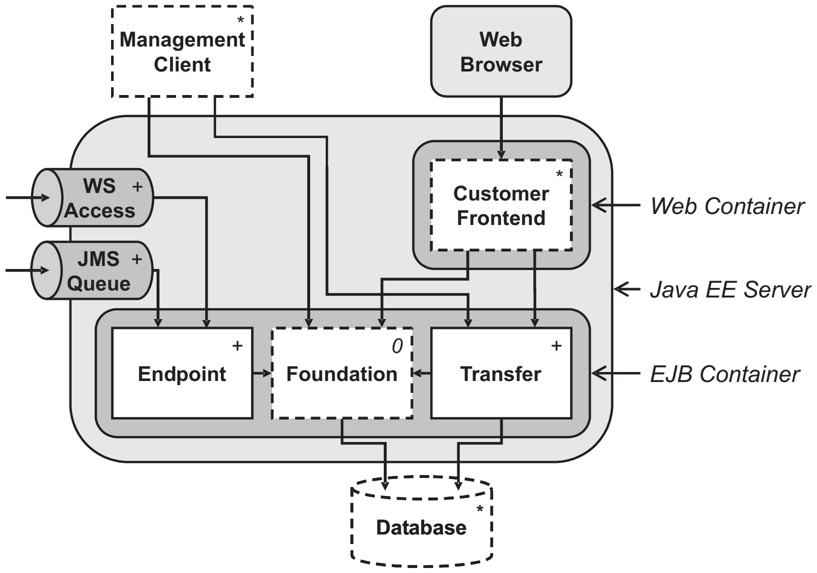


Figure 4.1.: Case Study – Blueprint

The following discussion concentrates on the EJB-specific aspects of Duke's Bank while only sketching the role of the other elements, because these do not lay within the focus of this thesis. For a detailed discussion of the aspects left out here please refer to the *Java EE 5 Tutorial* [91]. Section 4.1.1 presents the abstraction from the underlying *Database* as provided for the case study. Afterwards, section 4.1.2 contains a discussion of the business-tier regarding the three constituent EJB components *Foundation, Endpoint*, and *Transfer*. Furthermore, the intended connection structure in a deployed system is presented. Finally, section 4.1.3 addresses the client frontends consisting of *Management Client* and *Customer Frontend*.

4.1.1. Abstraction from the EIS-Tier

As foundation of Duke's Bank a *Database* is used to cover customer and account information, as well as the performed banking transactions. This database was extended with an additional table for storing access point information of remote banking systems. This new table is completely independent from the original ones. Therefore, no adjustments of those tables were necessary. The concrete design of the database is not relevant for the further discussion.

Figure 4.2 shows the abstraction from the underlying database as provided through entities following the *Java Persistence API* [59].

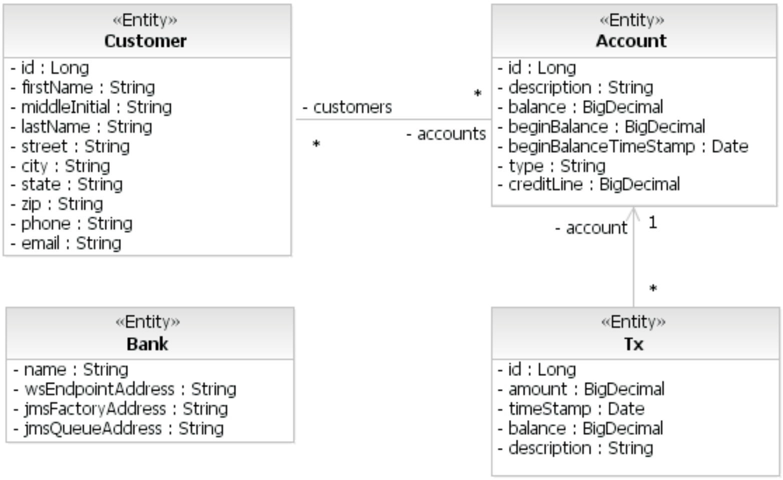


Figure 4.2.: Case Study – Abstraction from the EIS-Tier

The figure only includes the attributes of the different entities. All of them are accessible through corresponding `get` and `set` methods.

The state of a `Customer` consists of personal information like his or her name (`firstName`, `middleInitial`, and `lastName`), address (`street`, `city`, `state`, and `zip` code) and contact information (`phone` and `email`). Additionally, customers are connected to an arbitrary number of banking `accounts` of which they are the owners. This also covers the case that a customer does not possess any account at all. Finally, a unique identifier (`id`) is needed for identification purposes.

An `Account` is identified by its unique identifier (`id`). It might have a human readable `description` and covers its current `balance`. Additionally, the state of an account contains information about the initial balance (`beginBalance`) and a timestamp (`beginBalanceTimeStamp`) covering the time at which the initial balance was set. Furthermore, a banking system supports different types (`type`) of accounts. If the type of an account is equal to '`Credit`', the `creditLine` attribute covers that amount of money, the account might be overdrawn. Each account might be owned by an arbitrary number of `customers`. In this context, the Duke's Bank application theoretically allows that an account is not owned by any customer at all.

Each banking transaction (`Tx`) must reference that account which is the transaction target (`account`). In this context, each transfer results in two transactions. The first one represents the withdrawal of money from the transfer source, and the second one covers the deposit of money onto the target account. These two transactions are not associated inside the data structures of a banking system. Each banking transaction might contain a human readable `description`. It also holds the corresponding `amount` of money, as well as the time (`timeStamp`) when the transaction was performed. Finally, the resulting `balance` of the target account is kept as part of a banking transaction state.

While the previous entities already existed in the original case study, the `Bank` entity represents an extension introduced for this thesis. A bank is intended to keep access point information for a remote banking system which might be identified by a unique `name`. The necessary information for interacting with a remote bank through a Web Service (`wsEndpointAddress`) or through a JMS queue (`jmsFactoryAddress` and `jmsQueueAddress`) build the state of a `Bank` instance.

4.1.2. Realization of the Business-Tier

The business logic of Duke's Bank is realized through three EJB components, namely *Foundation*, *Endpoint*, and *Transfer*. For security reasons two different roles were defined in the original case study, namely `bankCustomer` and `bankAdmin`. The former role is intended to be mapped to the group of customers of a concrete banking system. The `bankAdmin` role represents the group of bank employees who are responsible to manage a banking system. Bank administrators are allowed to obtain general information about customers and accounts, and to perform changes upon them. They are prohibited to execute banking transactions. As extension for this thesis administrators are also responsible for managing access point information of remote banking systems. Customers might inspect accounts and might perform banking transactions. For the case study in this thesis this also covers remote transfers, and the addition and removal of standing orders. All other interactions are prohibited for `bankCustomers`. Regarding transaction management the original case study does not contain any specifications. According to the EJB standard a banking system should be supported by a container through container-managed transaction demarcation. Furthermore, each method invocation on a bean instance should be performed in accordance with the *Required* transaction semantics. These settings were realized for the new components analogously .

The following sections first discusses the three EJB components *Foundation* (section 4.1.2.1), *Endpoint* (section 4.1.2.2), and *Transfer* (section

4.1.2.3) separately. In this context, the required and provided interfaces are presented. Nevertheless, this does not cover details about specified methods, their signatures, and security settings for the particular interfaces and beans. Afterwards, section 4.1.2.4 provides an overview of the intended connection structure of an online banking system based on the components discussed in the previous sections.

4.1.2.1. The Foundation Component

The *Foundation* component was taken from the original case study. It consists of three enterprise beans, namely `CustomerControllerBean`, `AccountControllerBean`, and `TxControllerBean`. Each of these beans is realized as stateful session bean and provides one, individual business interface as depicted in figure 4.3. In a system context, instances of all SBs access and manipulate the underlying database through those entities which were already present in the original case study. This is not depicted in the figure. Furthermore, none of the beans does state any required interfaces for realizing its encapsulated functionality. Finally, no simple environment entries are defined.

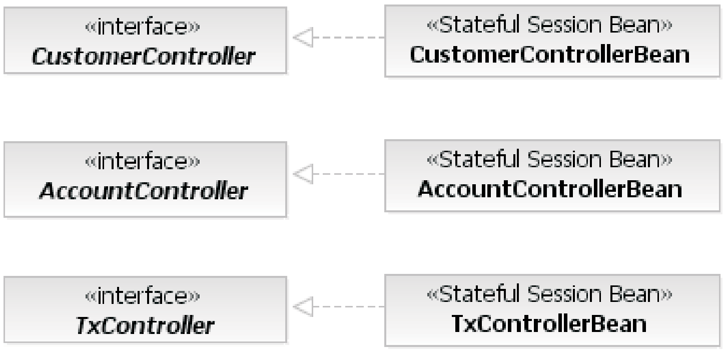


Figure 4.3.: Case Study – The *Foundation* Component

The interface `CustomerController` specifies methods for accessing the set of bank customers. For inspection purposes it defines methods to obtain information about a single customer, the set of customers which are the owners of a given bank account, or about all customers with a provided last name. Regarding the manipulation of customers, the interface provides methods for creating, removing and changing customers. The interface is implemented by the SB `CustomerControllerBean`.

In order to provide access to bank accounts, the `AccountController-Bean` SB realizes the functionality specified by the interface `Account-Controller`. This interface allows to obtain information about a single account or about all accounts associated with a given customer. Furthermore, it defines a method for requesting identifiers of all customers which are owners of a specific account. Finally, there are methods provided for creating and removing bank accounts, as well as for adding or removing customers to or from accounts.

The last SB in the *Foundation* component is the `TxControllerBean` which implements the `TxController` interface. This interface specifies methods for inspection and execution of banking transactions. Information about a single transaction might be obtained based on its identifier. Furthermore, all transactions belonging to a given account might be requested. Additionally, there are methods provided to withdraw or deposit money from or to an account. Finally, the interface provides a method to execute a transfer from one account to another.

4.1.2.2. The Endpoint Component

The *Endpoint* component provides functionalities to accept and integrate incoming bank transfers. The corresponding interfaces and the implementing enterprise beans, as well as their required interfaces are depicted in figure 4.4 on page 102. None of the included beans does specify any SEEs. `InterBankInterface` might be used to verify if a bank account with a certain identifier is hosted by a given banking system. Additionally, a second method accepts transfer information and performs the corresponding banking transaction inside a banking system. The interface is implemented by the stateless SB `InterBankControllerBean`. This bean exposes the interface as Web Service and as business interface. Therefore, it might be used by remote banking systems and by clients of the same system. In order to provide the encapsulated functionality, the bean states two required interfaces, namely `AccountController` and `TxController`. The first one is needed to verify account existence while the second one is used to execute banking transactions. Instances of the `InterBankControllerBean` do not interact with a database at all, because they solely rely on the functionalities of the required interfaces.

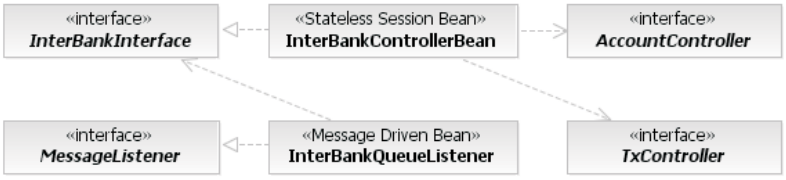


Figure 4.4.: Case Study – The *Endpoint* Component

The second bean (`InterBankQueueListener`) implements the `javax.jms.MessageListener` interface, which is part of the JMS standard [75], for being usable as message-driven bean. It is able to accept messages which represent incoming bank transfers from a message queue and transforms them into invocations on the `InterBankInterface`. Therefore, the bean can be seen as adapter for that interface. Instances of the MDB do not interact with the EIS-tier, but only convert and forward incoming transfer messages.

4.1.2.3. The Transfer Component

The third component of the case study (*Transfer*) encapsulates functionalities for remotely transferring money, and for establishing and removing standing orders. In this context, it also includes a facility for managing the set of known remote banking systems. The corresponding interfaces and beans, as well as their required interfaces are presented in figure 4.5.

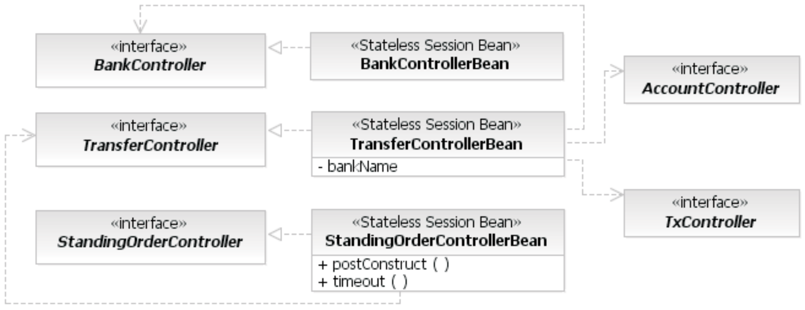


Figure 4.5.: Case Study – The *Transfer* Component

To manage the set of known access points to remote banking systems, the *Transfer* component provides the `BankController` interface. This interface specifies methods for setting and requesting JMS-based and Web Service-based access point information of remote banking systems. Furthermore, it enables users to request the names of all known banking systems and to obtain access point information of a concrete banking system. The interface is implemented by the `BankControllerBean` which internally makes use of the database extension through `Bank` entities. The bean is realized as stateless SB and does not specify any SEEs or required interfaces.

The `TransferController` interface defines a method for performing a banking transfer to an account hosted by the same or a remote banking systems. The functionality is implemented by the stateless SB `TransferControllerBean`. To identify whether a submitted transfer demand has to be performed locally or across banking system borders, the bean defines an SEE called `bankName`. This `String` entry is intended to hold the unique name of the banking system a deployed component belongs to. Its value is compared to the target bank name of a submitted transfer which allows to decide if a banking transfer should be performed locally or not. In order to realize the encapsulated functionality, an instance of the bean requires references to implementations of the interfaces `BankController`, `AccountController`, and `TxController`. Instances of the `TransferControllerBean` do not interact with the EIS-tier. Web Service references and JMS-specific connection aspects are not depicted in the figure, because they do not belong to the required interfaces. Instead of that, the corresponding settings are obtained through the `BankController` interface and are used to establish connections on demand.

The final bean of the *Transfer* component (`StandingOrderControllerBean`) is responsible for managing standing orders. Therefore, it realizes the `StandingOrderController` interface which provides methods for inspecting existing standing orders of a certain account, and for adding or removing standing orders. A standing order is internally realized through a timer (cf. section 3.2.3). Therefore, the `StandingOrderControllerBean` is implemented as stateless SB and provides a `timeout` callback method which adheres to the requirements of the EJB Timer Service. A reference to the timer service is obtained during the transition from the *configuring* to the *ready* state of an instance life cycle. Therefore, the bean defines a `postConstruct` life-cycle-callback-method (cf. section 3.1.3). In order to execute a standing order, instances of the bean forward the request to an implementation of the `TransferController` interface. Instances of `StandingOrderControllerBean` do not require any SEEs and do not interact with the EIS-tier for realizing the encapsulated functionality.

4.1.2.4. Intended Connection Structure in a System Context

The business-tier architecture of an online banking system might be established through the deployment of the three components presented in the previous sections and through connecting their required and provided interfaces. Figure 4.6 shows the resulting business-tier architecture.

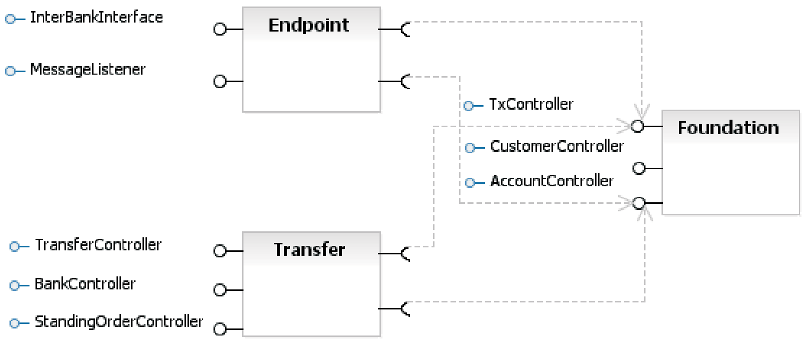


Figure 4.6.: Case Study – Intended Connection Structure

All provided interfaces of the constituent deployed components are also accessed by elements of the web-tier and the client-tier. For the remainder of this thesis this architecture is assumed as being deployed if no other configuration is highlighted.

4.1.3. Web-Tier and Client-Tier

In accordance with the security settings there are two groups of clients considered in the case study, namely *Bank Administrators* and *Bank Customers*. For each of those groups a specific client frontend is provided as depicted in figure 4.1 on page 96.

Administrators interact with an online banking system through a *Management Client*. This standalone application client accesses bean instances directly from outside a Java EE server. In order to provide the basic management functionality to administrators, the client interacts with the business-tier of an online banking system through the interfaces `CustomerController` and `AccountController`. Furthermore, the extensions implemented for this thesis require access to a system through the `BankController` interface to manage access points of remote banking systems.

Bank customers can interact with a banking system through a *Customer Frontend* which is accessed through web browsers. The frontend makes use of all interfaces provided by the *Foundation* and *Transfer* components. Regarding the three interfaces also used by the *Management Client*, the *Customer Frontend* only makes use of methods which are intended for inspection purposes.

4.2. Overview of the AC-Infrastructure

This section contains a short overview of the core elements of the AC-infrastructure. It should improve the understandability of the following chapters which discuss the different elements in detail.

The AC-infrastructure presented in this thesis aims to provide a comprehensive and generic basis for the autonomic management of enterprise systems based on EJB 3.0. Consequently, it can be interpreted as foundation upon which solutions for different AC areas might be realized. The infrastructure is named *mKernel* which is an abbreviation of *Manageable Kernel*. This name should point out the addressed application domain and was used in all publications about the infrastructure (cf. [32, 34, 35, 157]).

To fulfill the requirements stated in section 1.3 *mKernel* consists of three main parts, namely the *Preprocessor*, the *Reflective Meta Model*, and the *Container Plugin*. These are represented in figure 4.7 on page 107 through gray shaded rectangles with dashed borders.

The **Preprocessor** is a command line tool which accepts a standard compliant EJB component as input and preprocesses it for integration in an *mKernel*-based system. This is denoted in figure 4.7 through the arrow from the ejb-jar file in the upper left corner to the preprocessor. Beyond the provision of an ejb-jar file the tool does not demand for additional configuration. Furthermore, the establishment of manageability does not need to be considered inside processed components. Consequently, separation

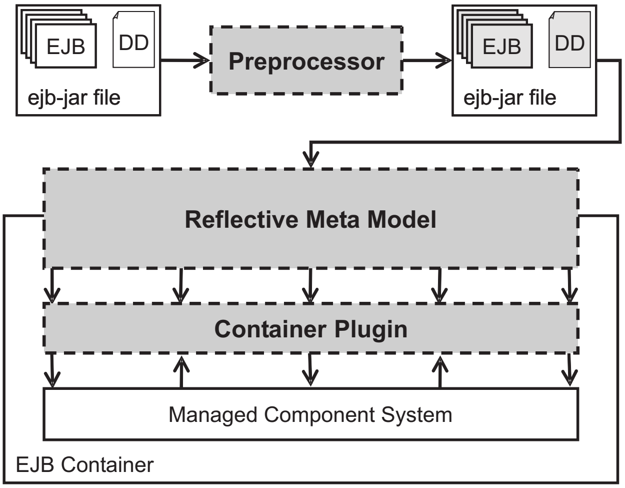


Figure 4.7.: Overview of the AC-infrastructure

of concerns regarding business logic and management aspects is reached. During its execution the tool extracts all necessary information from the component under consideration and constructs a processable representation for later usage in a system context. Additionally, the Java-byte-code of the component is manipulated for inspecting and controlling the later runtime behavior of bean instances. Furthermore, interceptors are attached to the included enterprise beans which are needed by *mKernel*. Finally, a preprocessed component is extended with additional beans which provide access for managing entities from outside a deployed component. In the end of preprocessing the tool emits new, standard-compliant ejb-jar files which might be integrated into *mKernel*-based systems. This is depicted in figure 4.7 through the outgoing arrow of the preprocessor. The gray shading of the included enterprise beans and the DD of the generated ejb-jar file indicates that the corresponding artifacts were manipulated during tool execution. Preprocessed components are not limited to a concrete system, but might be integrated into any system which supports *mKernel*.

In relation to the software life cycle of components the tool is able to process results of the development and maintenance phase. In this context, it would allow software vendors to ship components ready for integration into *mKernel*-managed systems. Alternatively, managers of a concrete system might use the tool to preprocess standard-compliant components. This would allow them to receive components from different sources or vendors, and integrate management aspects after their obtainment. Nevertheless, the tool is not intended to support the concrete configuration before deployment, but solely focuses on software aspects regarding the constituent elements of a component. Consequently, the execution of the tool might be assigned to the *Application Assembler* role defined in the EJB standard.

The **Reflective Meta Model** defines the access point for interaction with an *mKernel*-based system. It is realized as Java API which allows model based management of an enterprise system and provides a comprehensive set of sensors and effectors. The meta model is characterized as *reflective* with respect to the managed component system, because it represents the managed system in a way that allows the inspection and manipulation of its internals. Furthermore, the meta model is causally connected with the managed system, because changes in the system lead to corresponding changes of the model and vice versa (cf. [106]).

A managed system is considered on three different levels. The *Type Level* addresses software aspects of a managed system, that is, artifacts being the result of development and maintenance. The *Deployment Level* concentrates on the concrete system architecture inside a container and the configurations of the constituent elements. Finally, the *Instance Level* addresses bean instances and their states, as well as interactions among them. Relations between elements of the different levels can be identified and inspected. For a given session bean instance (*Instance Level*) it is, for example, possible to identify the deployed session bean (*Deployment Level*) it instantiates. For that session bean the corresponding implementation can be requested (*Type Level*). With this multi-level view subtle management operations become possible. As foundation for system management ejb-jar files – preprocessed by the preprocessor – can be integrated into an *mKernel*-managed system through API operations. This is represented in figure 4.7 on page 107 through the arrow from the ejb-jar file in the upper right corner to the reflective meta model.

The Java API realizing the meta model might be used inside or outside of containers. It supports the *Deployer* and *System Administrator* roles of the EJB standard which are assumed of being taken over by autonomic entities.

The **Container Plugin** builds the foundation for the management inside a container. It is realized through a set of enterprise beans. A deployed plugin is responsible for the coordination between managing entities accessing the deployed plugin through the API and the managed system itself. This is represented in figure 4.7 through the arrows between the meta model and the plugin, and between the plugin and the *Managed Component System*, respectively.

A deployed container plugin internally stores different types of information and artifacts obtained through the API such as preprocessed ejb-jar files or configurations for deployed components. Thus, it frees the meta model implementation from the need to keep persistent information itself. If necessary, configurations are forwarded to deployed components for reaching a causally connection between the managed system and its representation kept by a deployed plugin. Regarding a managed system the corresponding plugin collects and stores different types of information, for instance, to track interactions taken place between instances of enterprise beans. This information is provided to the API on demand.

A deployed container plugin is not intended to be used directly. It is designed and realized as server side extension of the API and should therefore only be used through it.

mKernel was built and tested on top of the *GlassFish Application Server* [11] which is a Java EE 5 server including, amongst others, an EJB container. It has proven itself of being compliant to the EJB standard to a very high degree. This especially includes the opportunity to integrate EJB components without the need for container specific extensions as far as the corresponding aspects were completely covered by the EJB standard. Therefore, the different parts of the AC-infrastructure could almost be designed and realized solely based on the EJB standard.

The following two chapters present the different parts of *mKernel* in detail. Chapter 5 starts with the meta model, highlighting the different opportunities provided by *mKernel* for the management of a component system. Afterwards, chapter 6 discusses how these opportunities are realized through the different parts of the infrastructure. The chapter also describes the general aspects of component preprocessing.

5. The mKernel Meta Model

This chapter presents the reflective meta model which builds the foundation for interaction with an *mKernel*-managed system[5]. It is realized as Java API, providing sensors and effectors for autonomic management. In this context, all elements of the API representing a managed system are realized as Java interfaces, because the complete system representation should be under control of the API. Therefore, no new elements should be instantiated by managing entities. Furthermore, the abstraction through interfaces would facilitate the realization of alternative implementations of the API that, for example, rely on vendor specific extensions of the EJB standard.

The meta model of *mKernel* considers managed systems on three different levels. The *Type Level* is intended to provide an insight into software aspects of a managed system. It can be interpreted as an interface to a kind of component repository which, for example, enables investigations for system reconfiguration. Furthermore, it allows to analyze relations between deployed components and their corresponding components. On *Deployment Level* the architecture of a managed system is represented through deployed EJB components and connections among them, as well as their concrete configurations. It provides the foundation for parameter and compositional adaptation. Finally, the *Instance Level* addresses instances of enterprise beans and interactions among them. It might, for instance, be used to identify faults within a managed system and to ana-

[5] An overview of the meta model was presented in a paper for the *6th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA-08)* [32] and was discussed in a corresponding talk.

lyze their context. The terminology used for the meta model is oriented at the *Java Platform, Enterprise Edition Specification, v5* [140]. In this context, an EJB component is called *EJB Module*, because it is considered as modular part of an enterprise software. The level of consideration can be identified through the suffix of the corresponding meta model element for *Type Level* and *Instance Level* while the elements of the *Deployment Level* do not have suffixes. On *Type Level* an `EnterpriseBeanType` represents, for example, an enterprise bean implementation inside an ejb-jar file while on *Deployment Level* an `EnterpriseBean` is part of a deployed component. Finally, on *Instance Level* an `EnterpriseBeanInstance` represents a concrete instantiation of an `EnterpriseBean`.

The following sections present the different aspects covered by the meta model. In this context, only the relevant parts of the corresponding elements are considered in order to focus on the particular topic. Therefore, not all aspects of the meta model elements are discussed completely at one single point within the following sections, but only those that are relevant in the particular context. This proceeding was chosen because of the various relationships between certain elements in different contexts which might even span more than one level. Furthermore, this thesis concentrates on aspects which are relevant for describing the general concepts. Those parts which do not contribute to the understanding of the concepts are left out to keep the discussion focused.

As central access point to an *mKernel*-managed system the interface `Container` is intended to be used. It provides methods for obtaining different access points to the three levels of a managed system. A `Container`-based reference can be obtained through the static method `getNewContainer` from the class `ContainerFactory` as follows:

```
Container c = ContainerFactory.getNewContainer();
```

For the underlying implementations it is necessary that the execution environment is properly configured for interactions with the target container[6].

The following discussion starts with the presentation of the layering of software and system architectures in section 5.1. This layering is specific to *mKernel* and supports, amongst others, dynamic reconfiguration. Afterwards, the three different levels of the meta model are discussed in the sections 5.2 to 5.4. To support managing entities with opportunities to externally keep information about elements of a managed system and to facilitate synchronization with the information base of an *mKernel*-based system the meta model provides a *Notification Facility*. This facility is presented in section 5.5, because it is not limited to a specific level. The *Seamless Reconfiguration* of a system affects the *Deployment Level*, as well as the *Instance Level*. Therefore, this aspect is discussed separately in section 5.6. Finally, section 5.7 draws a conclusion of the meta model.

5.1. Layering of Software and System Architectures

The *mKernel* infrastructure focuses on business-tier management of EJB-based enterprise systems which are the targets of inspection and manipulation. Those systems might interact with their clients through various access points like web frontends, standalone clients, Web Service endpoints, or JMS queues as depicted in figure 4.1 on page 96. For *mKernel* it is assumed that managing entities do not necessarily need to control client systems. In fact, it does not even demand that they are managed autonomically at all. Therefore, the infrastructure is designed and realized in a way that allows its application in isolation, that is, it does not require any management support facilities beyond those defined in the *Java Platform, Enterprise Edition (Java EE) Specification, v5* [140]. To realize the management of an enterprise system components which provide business logic are adjusted and extended during preprocessing. This includes, amongst

6 It is especially necessary that the default constructor of `javax.naming.InitialContext` creates a reference to the naming context of the managed container.

others, the manipulation of interfaces and method signatures for transferring management information. These changes must be hidden from system clients, because clients should not even recognize *mKernel*-based management of a system they interact with[7]. Furthermore, for supporting dynamic adaptation of managed systems, the AC-infrastructure must be able to isolate the system partially or completely from client initiated interactions[8]. To fulfill these requirements deployed components of a managed system are organized in two separate layers, namely *Managed Layer* and *Access Layer*. This distinction does not induce any additional demands on component development, because the preprocessor tool is able to generate the corresponding ejb-jar files automatically from the results of development and maintenance[9]. Nevertheless, a generated ejb-jar file can either be deployed as part of the *Managed Layer* or the *Access Layer*. Therefore, the distinction between the two layers is not limited to the *Type Level*, but also affects the other levels of the meta model.

The *Managed Layer* of an *mKernel*-based system contains the provided business logic. Module types belonging to the *Managed Layer* on *Type Level* are directly derived from the results of development or maintenance during preprocessing. Therefore, the constituent enterprise bean types and other elements of an original ejb-jar file are adopted, adjusted, and extended. Consequently, ejb-jar files belonging to the *Managed Layer* cover the original business logic which might be integrated into a system architecture through deployment. On *Deployment Level* EJB modules belonging to the *Managed Layer* and their enterprise beans are the constituent elements of the internal system architecture of an *mKernel*-managed system. This part of the business-tier architecture is characterized as *internal*, because its elements are intended to be referenced only inside the business-

7 Besides small delays resulting from the management overhead.

8 For further details regarding seamless reconfiguration, please refer to section 5.6.

9 Please refer to section 6.2 for further detail regarding preprocessing.

tier, that is, from the *Managed Layer* or the *Access Layer* on *Deployment Level*. External clients are not allowed to reference elements of this layer. Compositional adaptations performed on *Deployment Level* of the *Managed Layer* have direct impacts on available functionality inside the business-tier of a managed system. According to the restriction that only other elements of a managed system are allowed to reference elements of the *Managed Layer* on *Deployment Level*, interactions where instances of the *Managed Layer* are the targets always originate from instances of other elements controlled by *mKernel*. Based on this foundation full control over interactions arriving at the *Managed Layer* is established on *Instance Level*.

In contrast to the *Managed Layer*, elements of the *Access Layer* on *Type Level* do not cover any business logic. They represent opportunities for the deployment of client access points into the business-tier of managed systems. Access point realizations are derived from the results of development and maintenance during their preprocessing. Beyond the ability to forward client initiated interactions to elements of the *Managed Layer* on *Instance Level*, they also include functionalities to block or hold client invocations for supporting dynamic adaptation inside the *Managed Layer*. Furthermore, they are realized as adapters, because their provided interfaces are those which were defined during development or maintenance. Internally, they make use of extended interfaces which are provided by elements of the *Managed Layer* to support, amongst others, the transfer of administrative information during interactions.

On *Deployment Level*, the internal business-tier architecture is defined through the establishment of connections between *Managed Layer* enterprise beans. Furthermore, client access points are created through the deployment of module types belonging to the *Access Layer*. Afterwards, those must be connected to *Managed Layer* beans for being able to forward client interactions on *Instance Level*. Figure 5.1 on page 116 shows a part of the case study covering the resulting system architecture in an *mKernel*-managed system analog to the corresponding part of the system architecture presented in figure 4.6 on page 105.

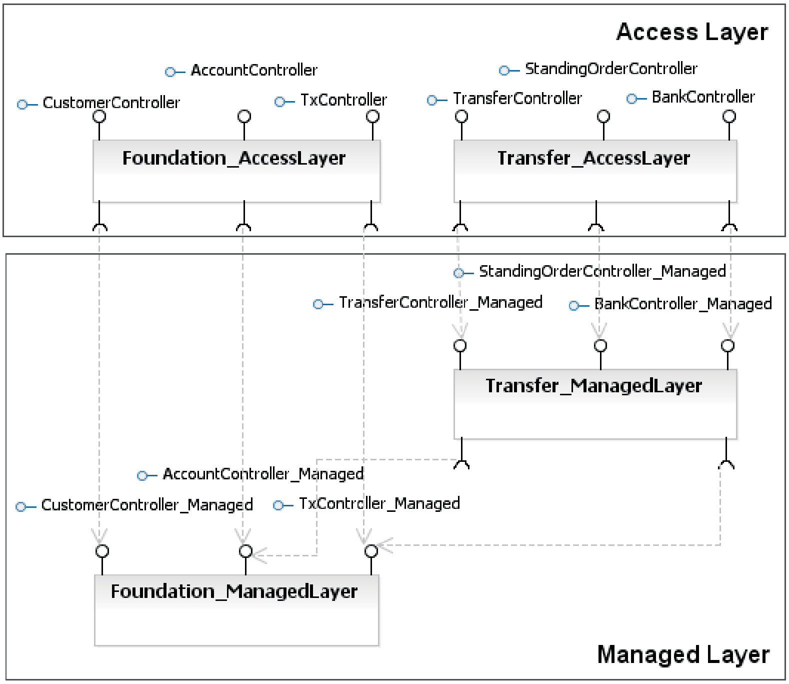


Figure 5.1.: Exemplary layered System Architecture

Inside the figure, EJB modules have the same name as those of the case study followed by the suffix `_AccessLayer` for modules belonging to the *Access Layer* and the suffix `_ManagedLayer` for modules of the *Managed Layer*. *Access Layer* modules provide the original interfaces of the case study to external clients such as the provided interface `TxController` of the module `Foundation_AccessLayer`. Those are mapped to providers of extended interfaces belonging to the *Managed Layer* such as the provided interface `TxController_Managed` of the module `Foundation_ManagedLayer`. In this context, it is possible that more than one provider of an interface on *Access Layer* is connected to the same provider on *Managed*

Layer. Such a situation is not shown in the figure. Internal interfaces are represented by interfaces with the same name as the original interfaces followed by the suffix `_Managed` to indicate that they are derived from the original ones. For interfaces required by *Managed Layer* modules direct connections to session beans providing those interfaces might be established inside the *Managed Layer*. For these connections the *Access Layer* does not participate in corresponding interactions at runtime. An example of such a constellation is given by the connection from the `EJB` module `Transfer_ManagedLayer` to the module `Foundation_ManagedLayer` through the adjusted interface `TxController_Managed`. Finally, an interface provided inside the *Managed Layer* does not necessarily need to have a corresponding provided interface inside the *Access Layer*. This is the case if a certain interface is only provided inside the business-tier, but not to external clients. Such a situation is also not depicted in the figure.

The integration of *Access Layer* modules allows the provision of client access points at certain mapped names over a relatively long timespan. Client implementations might rely on the provision of those access points, because the underlying beans do not cover any business logic on their own, but only forward invocations to the original targets on *Managed Layer*. If adjustments of the business-tier become necessary which result in the need for compositional adaptation, these adaptations would not affect the availability of access points themselves. The only reasons to undeploy an *Access Layer* module would be, if one or many of the provided access points should not be supported anymore, if access points should be provided under a different mapped name, or if the set of provided interfaces at a certain mapped name should be changed. In contrast, adjustments of business logic result in the need for deployment operations inside the *Managed Layer*, as discussed in section 3.1.2.2. If clients would interact with the *Managed Layer* directly, they might notice these operations through temporal unavailability of access points.

5.2. The Type Level

On *Type Level* the API represents all aspects of ejb-jar files which were integrated into an *mKernel*-managed system. These archives are not necessarily deployed inside a managed system, but can be interpreted as the constituent elements of a kind of component repository for a given system. For this repository the *Type Level* of the meta model provides sensors for inspection and effectors for manipulating the set of repository elements. All aspects and settings exposed by the meta model are transferred to corresponding modules on *Deployment Level* during their creation. Consequently, the *Type Level* view on ejb-jar files provides insight into the default configuration of derived modules.

The discussion of the *Type Level* is structured as follows: Section 5.2.1 gives an overview of how the API addresses the general structure of ejb-jar files. Afterwards, section 5.2.2 discusses the representation of interface types inside the meta model. The following section 5.2.3 addresses parameterization options. Section 5.2.4 presents the access point to the *Type Level* provided by the `Container` interface. Subsequently, section 5.2.5 explains the concrete usage of the *Type Level* API based on the case study.

5.2.1. Structural Representation of ejb-jar files

The meta model represents ejb-jar files on *Type Level* through the interface `EjbModuleType` as depicted in figure 5.2.

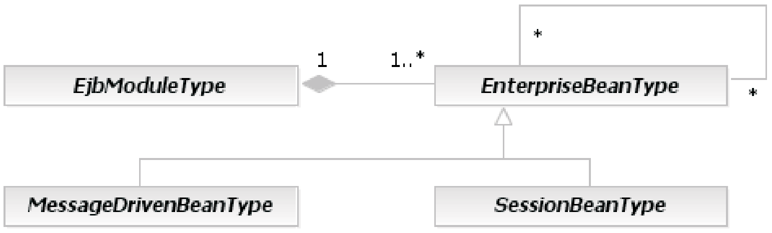


Figure 5.2.: *Type Level* View on an ejb-jar file

A module type is characterized by a `uniqueIdentifier` which can be used to identify the module type inside a managed system. This identifier is calculated during preprocessing of the corresponding ejb-jar file through application of the *US Secure Hash Algorithm 1* (SHA1) [66] over the binary content of the extracted archive. During integration of a module type the corresponding unique identifier enables an *mKernel*-based system to identify whether the module type is already known. If this is the case, the creation of a redundant representation is avoided. Consequently, each ejb-jar file is represented exactly once, even if autonomic entities try to integrate it multiple times. To identify whether a module type belongs to the *Managed Layer* a corresponding method (`isManagedLayer`) can be used. On invocation it returns a `boolean` value indicating whether a *Managed Layer* module type is given (`true`) or not (`false`). *mKernel* stores the original ejb-jar file of a module type and exposes it to autonomic entities as byte arrays. The ejb-jar file might be used as extension point of the API for managing entities, because it enables the inspection of EJB module types with respect to aspects not covered by *mKernel* like vendor specific extensions. Furthermore, it contains the constituent implementations of a module type. These might be very helpful for in depth analyses when errors are detected at runtime. Additionally, the EJB standard allows the extension of ejb-jar files with additional artifacts, as discussed in section 3.1.1.3. Therefore, managing entities might enrich the original ejb-jar file with custom artifacts before integrating them into an *mKernel*-managed system. The included artifacts might afterwards be requested indirectly from *mKernel* through the original ejb-jar file. Nevertheless – depending on the size of the particular archive – this proceeding might be costly, because in order to gain access to entries the complete archive must be transferred. Finally, managing entities can remove a module type from an *mKernel*-managed system. This might be meaningful, for example, when a module type becomes deprecated and should not be kept within a system anymore. Nevertheless, a removal is only valid if there do not exist any corresponding modules on *Deployment Level* of that module type. *mKernel* would react with an exception to an invalid attempt to remove a module type, because otherwise the underlying data source would become inconsistent.

Each module type contains at least one enterprise bean type (`EnterpriseBeanType`). These types represent the constituent elements of the module type and can be accessed through the association between `EjbModuleType` and `EnterpriseBeanType`. As an enterprise bean type is an integral part of exactly one module type, it has a composition relationship with the corresponding module type. Consequently, if a module type is removed from an *mKernel*-managed system, the included enterprise bean types are removed also. In contrast, EJB types might not be removed independently.

For each enterprise bean type a unique identifier is generated during its integration into an *mKernel*-managed system (`getUniqueIdentifier`). This identifier might be used for identification purposes, but is not directly bound to the underlying implementation of the EJB type. In contrast, it relates to the application of the bean type as part of a concrete module type. Consequently, more than one enterprise bean type inside a system might be based on the same implementation while all of them have different unique identifiers. The fully qualified class name (FQN) of its implementation might also be requested from a bean type. Nevertheless, this FQN is not sufficient to deduce if two distinct enterprise bean types have exactly the same underlying implementation. It is, for instance, conceivable that they are based on the same implementation, but on different revisions. For certain situations it might be of special interest to find out which other enterprise bean types are based on exactly the same implementation as a given one. This might, for example, be the case, if an error is identified within the corresponding source code. Therefore, `EnterpriseBeanType` provides a special method (`isSameClass`) which permits to request whether two bean types have exactly the same underlying implementation. Internally, the comparison of the implementations is based on the identifiers of the *class* files of the particular bean types which are calculated – analog to the unique identifiers of EJB module types – through the *SHA-1* hash function[10]. An invocation of the method `isSameClass` is only meaningful in the context of two enterprise bean types of the *Managed Layer*, because only those bean types contain business logic.

Bean types themselves are specialized into message-driven bean types (`MessageDrivenBeanType`) and session bean types (`SessionBeanType`). Beyond the functionality specified in the `EnterpriseBeanType` interface, `MessageDrivenBeanType` allows autonomic entities to request the message selector of the MDB type. For a `SessionBeanType` it can be requested whether it is a stateful session bean type.

5.2.2. Representation of Interfaces

The *Type Level* of the meta model provides a comprehensive representation of interface types being part of EJB types inside an integrated ejb-jar file. In this context, the three meta model interfaces `JavaInterfaceType`, `EjbInterfaceType` and `EjbReferenceType` build the foundation as depicted in figure 5.3 on page 122.
The interface `JavaInterfaceType` is used to represent Java interfaces which are types for the specification of provided and required interface types. Each Java interface type has a unique identifier which is calculated analog to those of EJB module types to ensure that exactly the same Java interfaces being part of more than one module type might be identified

[10] Equality of calculated identifiers is based in the assumption that the same class-file is integrated into different archives, or that the results of independent compilations of the same implementation are also equal. This needs not be the case. A counterexample would be, if different compilers are used which emit different class-files for the same source code. For this case *mKernel* would not be able to identify that equality of implementations is given.

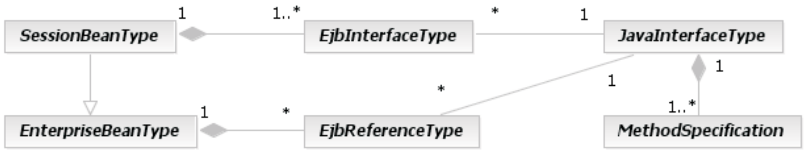


Figure 5.3.: *Type Level* Representation of Interfaces

correctly by the infrastructure. This is not possible based on FQNs, as already discussed in the context of enterprise bean types in section 5.2.1. *mKernel* recognizes whether a Java interface type is already known in a system during integration of a module type. If this is the case, *mKernel* prevents the creation of redundant representation of that interface type. Consequently, each known Java interface is represented once inside an *mKernel*-managed system. Each Java interface type contains a collection of method specifications (`MethodSpecification`) which represent the specified functionality of the interface. As they are an integral part of the interface, they are in a composition relation to it and are removed on deletion of the `JavaInterfaceType`. Each method specification provides information about its signature regarding its name, return type, parameter types, and thrown exceptions. Finally, a Java interface type allows managing entities to identify whether it might be exposed as Web Service interface on *Deployment Level*.

An `EjbReferenceType` represents the specification of a required interface for an enterprise bean type. It is an integral part of the EJB type and, therefore, stands in a composition relationship to it. Beyond the association to the EJB type and the Java interface type, a reference type exposes its mapped name in the local namespace of the corresponding enterprise bean type. This allows to distinguish reference types which are based on the same Java interface type and belong to the same EJB type. Each *Access Layer* SB type contains a reference type for each of its provided interfaces, because at runtime its instances need a corresponding reference to forward incoming method invocations to the *Managed Layer*.

Specifications of provided interface types of SB types are represented through `EjbInterfaceTypes`. Analog to reference types they stand in a composition association to the corresponding session bean type and are associated with the corresponding Java interface type. Furthermore, EJB interface types allow autonomic entities to request the type of provision, that is, whether the interface might, for instance, be provided as business interface on *Deployment Level*.

`JavaInterfaceTypes` considered on *Type Level* always relate to the original Java interfaces. The extension of interfaces for the *Managed Layer* and their application in a managed system are hidden from managing entities. Therefore, from the *Type Level* point of view `EjbInterfaceTypes` and `EjbReferenceTypes` are always associated with representations of the original Java interfaces.

5.2.3. Representation of Parameterization Options

According to the EJB standard EJB module types might contain configurations for different aspects either through annotations or inside their DD. *mKernel* supports inspection of configuration opportunities on *Type Level* through the meta model interfaces shown in figure 5.4 on page 124. In this context, a concrete model reflects the configuration *mKernel* has identified during preprocessing of the corresponding ejb-jar file.

Simple Environment Entries As discussed in section 3.1.1.3, simple environment entries might be used for parameterization of enterprise beans. On *Type Level, mKernel* represents those opportunities for parameterization through `SimpleEnvironmentEntryTypes`. They are characterized by a `name` which is unique for the particular EJB type and a `type` for its values in a concrete configuration scenario. Furthermore, its default value (`defaultValue`) might be requested in case *mKernel* was able to identify

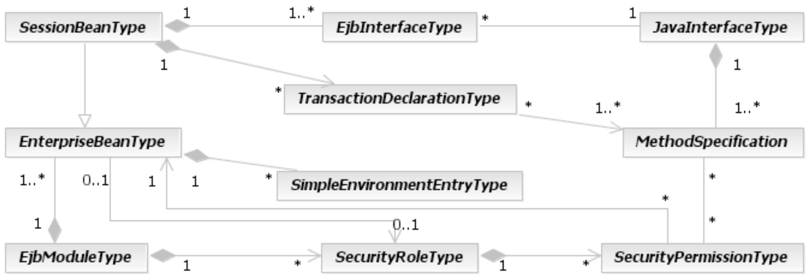


Figure 5.4.: *Type Level* Representation of Parameterization Options

it during preprocessing of the module type. SEE types are integral parts of the corresponding EJB type. In this context, each enterprise bean type might posses an arbitrary number of SEE types which provide an overview of the options for configuration in case of deployment.

Transaction Settings To identify whether the implementation of an enterprise bean type performs bean managed transaction demarcation or relies on container managed transaction demarcation, the method `isContainerManagedTransactionDemarcation` on `EnterpriseBeanType` might be used which returns `true` if CMTD is given.

For MDB types relying on CMTD the method `getTransactionAttribute` returns an enum constant of `javax.ejb.TransactionAttributeType`[11] which applies to the method for receiving a message from a JMS queue or topic.

For session bean types relying on CMTD each SB type might own an arbitrary number of declaration types. Those types are characterized by a `javax.ejb.TransactionAttributeType` and are related to at least one `MethodSpecification`. It might be possible that the method signature, the declaration type relates to, matches with more than one method spec-

[11] This enum is used in the context of the EJB standard to define transaction attributes through metadata annotations.

ification of different Java interfaces an SB type provides. Due to the underlying mapping to the EJB standard it is, consequently, possible that a declaration type applies to more than one method specification which is reflected through the `1..*`-cardinality at the corresponding association. In this context, each method specification a transaction declaration type refers to is provided by the session bean type through its `EjbInterfaceTypes` and the corresponding `JavaInterfaceType`.

Security Settings Each ejb-jar file might contain an arbitrary number of security role definitions. Those are represented on *Type Level* through `SecurityRoleTypes`. Each security role type has a unique name within the module type and might itself be the owner of an arbitrary number of `SecurityPermissionTypes`. Each permission type refers to an enterprise bean type to which the corresponding permission specification applies. If the EJB type is an SB type, the security permission type refers to at least one method specification for which the represented permission specification was defined. Those method specifications are deduced the same way as described for transaction declaration types. For MDB types no method specifications are referred to. For this case it can be assumed that it relates to the method for receiving a message from a JMS queue or topic. Additionally, each enterprise bean might refer to a security role which should be used in a system context when instances of the bean type perform method invocations themselves. This might, for instance, be necessary if different roles are required by referenced bean instances or to define the security role which should be used during the execution of timer callbacks.

Message Selectors of Message-Driven Bean Types MDB types might cover a message selector for their binding to a JMS message destination during deployment. This selector is used to identify messages which are relevant for that MDB at runtime, as discussed in section 3.1.1.1. The selector might be requested from a `MessageDrivenBeanType` and is represented as `java.lang.String`. This is not depicted in figure 5.4 on page 124, because message selectors are not represented through special meta model elements.

5.2.4. Type Level Access Points

As mentioned in the introduction of chapter 5, the `Container` interface provides the central access point to an *mKernel*-managed system. Regarding the *Type Level* of a concrete model direct access to module types, EJB types, or Java interface types can be obtained. For each of those two methods are provided, one for requesting a representation of a specific type and one for obtaining a collection of all representations of the particular type. The methods for gaining access to a single representation expect the corresponding unique identifier as a parameter. Table 5.1 summarizes the methods provided by the `Container` interface.

Type	Specific Representation	All Representations
`EjbModuleType`	`getEjbModuleType`	`getEjbModuleTypes`
`EnterpriseBeanType`	`getEjbType`	`getEjbTypes`
`JavaInterfaceType`	`getJavaInterfaceType`	`getJavaInterfaceTypes`

Table 5.1.: Access Points to Elements of the *Type Level*

Additionally, `Container` provides two variations of the overloaded method `createModuleType` to integrate new module types into a managed system. The first one expects two byte arrays as parameters. The first one of those must contain the original ejb-jar file while the second one must cover the preprocessed file. The second variant of `createModuleType` is provided for convenience reasons within environments granting access to file systems. It accepts the FQNs of the two ejb-jar files which prevents users from the need to serialize them into byte arrays for integration.

Finally, the method `getJavaInterfaceTypesByName` might be used to obtain a collection of representations for all Java interface types with a certain name from an *mKernel* system.

5.2.5. Application Example

After the discussion of the different meta model elements on *Type Level* this section explains how the corresponding API might be used based on the case study presented in section 4.1. As foundation for the managed system it is assumed that the container plugin is integrated into the managed system successfully. The ejb-jar files of the case study must have been processed with the preprocessor tool (section 6.2)[12]. Finally, it is assumed that the example source code is executed in an environment that grants access to a file system the original and processed ejb-jar files can be read from.

The remainder of this section presents a straight forward approach of how the *Type Level* of the API might be used for planning the deployment of `EjbModuleTypes` based on a set of Java interface types to provide. It is not intended to provide a comprehensive solution for self-configuration based on Java interface demands, but should give a first impression of how the *Type Level* of the meta model might be used. Nevertheless, it is kept generic and is not limited to the case study. During planning, `EjbModuleTypes` to deploy and `SessionBeanTypes` to configure are identified. Furthermore, connections to establish are proposed based on `EjbInterfaceTypes` and `EjbReferenceTypes`. As result of algorithm execution a plan is constructed which contains alternative options for connections to establish. These must be resolved by the client of the plan before realization. The plan does not consider parameterization of the affected module types and SB types. Nevertheless, a plan observer might inspect parameterization options with the help of the API.

12 This tool emits two ejb-jar files, one for the *Access Layer* and one for the *Managed Layer*.

Plans are realized by instances of the class `TypeLevelPlan` which ensures plan feasibility during execution. Furthermore, it supports its clients through automated adjustments during selection of alternatives. In the following the internal representation of a plan through data structures is discussed, followed by inspection opportunities. Afterwards, the processing of client requests for the provision of Java interface types is explained. Finally, the proceeding for reaching unambiguity of a plan is discussed.

While in this section only the relevant parts of the example are presented, appendix A contains the complete, source code.

5.2.5.1. Internal Representation of Plans

The implementation of `TypeLevelPlan` internally uses two maps for representing providers for desired Java interface types and for alternative connections in a deployment context based on `EjbReferenceTypes` and `EjbInterfaceTypes`:

- Java interface type providers p: The keys of this `java.util.Map` are the `JavaInterfaceTypes` which should be provided as goals of the plan. As values sets of `EjbInterfaceTypes` are referred to which might be used as alternative providers of the corresponding Java interface type on *Type Level*.
- Connection alternatives a: This `java.util.Map` contains required interface types identified during plan construction as keys. The values are provided interface types which might be used as alternative connection targets during further planning.

These data structures are sufficient for representing alternatives of possible system architectures fulfilling the planner demands based on *Type Level* elements. The first map is needed to cover the goals of the planner through required Java interface types and their alternative provider types.

The second map contains the current alternatives for fulfilling the goals with respect to proposed connections inside the managed system.

5.2.5.2. Inspection Opportunities

The following list contains a compilation of those methods provided by the class `TypeLevelPlan` which allow the inspection of a concrete plan.

1. `getProvidedJavaInterfaces`: This method delivers all `JavaInterfaceTypes` which should be provided as goals of the plan, that is, the key set of `p`.
2. `getJavaInterfaceTypeProviders`: As result the potential provider types of a concrete Java interface type are returned by this method. They are given by the corresponding entry in `p`.
3. `getRequiredReferences`: This method returns the key set of `a`, representing the required interface types which must be fulfilled for realizing the current state of the plan. If the return value contains more than one element, these represent alternative options for connection establishment.
4. `getConnectionAlternatives`: For a concrete reference type the corresponding value in `a` can be requested through this method. The result might again contain a set of alternative options for fulfilling the interface demand.
5. `getSessionBeanTypes`: The set of session bean types of which corresponding elements might be configured in a deployment context is delivered by this method. The result set consists of those `SessionBeanTypes` which are connected to at least one `EjbInterfaceType` in the value sets of `p` or `a`.
6. `getModuleTypesToDeploy`: The method provides the set of all affected modules types. It is derived from all module types associated with at least one of the `SB` types returned by the previous method.

7. `isUnambiguous`: This method returns `true` if the plan is unambiguous. This is given if all values in `p` and `a` contain exactly one element. For this case the plan does not require any decisions on *Type Level* for choosing between alternatives for the provision of `JavaInterfaceTypes` (`p`) or alternatives for connections (`a`).

Through the methods 1 and 3 an overview of the required Java interface types and reference types might be obtained. The corresponding methods 2 and 4 allow an in depth inspection regarding connection alternatives for a concrete Java interface type and reference type respectively. These might, for example, be used for navigation purposes, because the returned `EjbInterfaceTypes` provide access to the corresponding SB types which might be associated with `EjbReferenceTypes` themselves. These reference types can be submitted to method 4 in a subsequent invocation. Method 5 can be used to obtain additional configuration options on the level of SB types such as SEE types or default transaction settings. The same is possible on the level of `EjbModuleTypes` through method 6. Finally, method 7 can be used to identify whether a plan has reached a state which might be realized without the need for any further decisions on *Type Level* regarding connections to establish.

5.2.5.3. Plan Construction

In order to construct a plan, a planner must define whether the plan is intended to provide Java interface types on *Access Layer* or on *Managed Layer* during construction. Afterwards, the planner can provide instances of `JavaInterfaceTypes` as parameter to the method `addJavaInterface-TypeProvider` which should be provided as goals of the plan. After finishing execution, the method returns a `boolean` value indicating whether the interface might be provided (`true`) or not (`false`). If the provision of the interface was possible, the corresponding entries in `p` and `a` are integrated. Otherwise, the plan state is not changed. The method is realized as depicted in listing 5.1.

```
1  public boolean addJavaInterfaceType(
2     JavaInterfaceType j){
3   if (this.p.containsKey(j)) return true;
4   Set<EjbInterfaceType> r =
5      new HashSet<EjbInterfaceType>();
6   for(EjbInterfaceType ei:j.getEjbInterfaceTypes()){
7    SessionBeanType s = ei.getSessionBeanType();
8    if(s.getEjbModuleType().isManagedLayer() == this.m)
9       {
10    Map<EjbReferenceType, Set<EjbInterfaceType>> tmpA =
11       new HashMap<EjbReferenceType,
12       Set<EjbInterfaceType>>();
13    tmpA.putAll(this.a);
14    for(EjbReferenceType er:s.getEjbReferenceTypes()){
15     tmpA = this.provideEjbReferenceType(er, tmpA);
16     if(tmpA == null) break;
17    }
18    if(tmpA != null){
19     this.a = tmpA;
20     r.add(ei);
21    }
22   }
23  }
24  if(r.size() > 0){
25   this.p.put(j, r);
26  }
27  return r.size() > 0;
28 }
```

Listing 5.1: Integration of Java Interface Type into *Type Level* Plan

First of all, it is analyzed whether the desired Java interface type (`j`) is already provided as part of the plan. If this is the case, the method directly returns, indicating that provision was successful (line 3). The set constructed in lines 4 to 5 is used to hold all `EjbInterfaceTypes` which might be used to provide `j`. Afterwards, all possible providers of the desired Java interface type are analyzed in an iteration whether they can be integrated into the plan (lines 6 to 23). This is the case if all of the required interface types of the corresponding SB type might be connected to provided interface types of other session bean types recursively. Therefore, the corresponding SB type (s) is identified in the first step of the iteration (line 7). If s is associated with a module type belonging to the target provision level, (m) the proceeding for the current SB type is continued (lines 8 to 22). Otherwise, the surrounding iteration would continue with the next provided interface type, if such exist. If the desired layer is given, a copy of a is created (tmpA) in the lines 10 to 13. In this context, no deep copy is necessary, because only on top level new mappings might be integrated during further planning. The copy is used in the following lines to represent the current progress regarding the mapping from required interface types to alternatives of provided interface types. Therefore, the creation of a copy of a ensures that planning for the particular EjbInterfaceType is started based on the current state of the plan. Afterwards, it is tried to find a provided interface type for each of the required reference types of s (lines 14 to 17). This is done with the help of the method provideEjbReferenceType which is discussed separately below. From an external point of view, this method expects the required reference type and the current mappings from EjbReferenceTypes to sets of EjbInterfaceTypes as parameters. As return value it provides a new mapping from required to provided interface types also covering the submitted ones. If it was not possible to find at least one provider type for the required interface type, null is returned. Furthermore, it is ensured that the method does not change the original mapping a or its content. In line 15 the progress of planning is updated through assigning the return value of method execution to tmpA. If the method returns null, the iteration over the EjbReferenceTypes of s must be aborted, because for at least one required interface type no provider could be found (line 16). If the iteration led to a new mapping or no EjbReferenceTypes are required by s, a is replaced with tmpA (line 19), and the set of potential providers is enhanced with the new EjbInterfaceType (line 22). In combination, these two steps represent the extension of the plan. After all possible

EjbInterfaceTypes for j have been analyzed it is analyzed whether at least one EjbInterfaceType could be provided in a deployment context (lines 24 to 26). If this is the case, p is updated. This represents the end of planning regarding j. Finally, true is returned as result of method execution if at least one provider could be found for j (line 27).

Internally (line 15), the method provideEjbReference is used. This method contains the source code of listing 5.2 on page 134. The method requires an EjbReferenceType (r) for which provided interface types should be found and a map of connection alternatives representing the current state of planning (tmpA) as parameter values. In line 4 the Java interface type (j) associated with the required interface type is identified. The boolean value success indicates whether at least on provided interface type could be found which could be integrated into the plan, that is, for all required interface types of the corresponding SB type EjbInterfaceTypes belonging to the *Managed Layer* could be found recursively. It is initialized with false in line 5, because at the beginning of execution no EjbInterfaceTypes are identified. Afterwards, all provided interface types for j are analyzed (lines 6 to 34). First of all, the corresponding session bean type (s) and EJB module type (m) of the provided interface type (i) are identified in the lines 7 and 8. Afterwards, it is analyzed whether further investigations regarding i are meaningful (lines 9 to 11). This is the case if m belongs to the *Managed Layer*, because only *Managed Layer* EjbInterfaceTypes might be used as connection targets inside a managed system. Furthermore, if i provides j locally[13], it must be part of the same module type as r for being potentially usable as target of a connection in a deployment context. If i does not provide j locally, the module types of i and r are not relevant.

[13] Either as local home or local business interface.

```
private Map<EjbReferenceType, Set<EjbInterfaceType>>
  provideEjbReferenceType(EjbReferenceType r,
  Map<EjbReferenceType, Set<EjbInterfaceType>> tmpA){
 JavaInterfaceType j = r.getJavaInterfaceType();
 boolean success = false;
 for(EjbInterfaceType i:j.getEjbInterfaceTypes()){
  SessionBeanType s = i.getSessionBeanType();
  EjbModuleType m = s.getEjbModuleType();
  if(m.isManagedLayer()&&((i.isLocal() &&
    r.getEjbType().getEjbModuleType().equals(m)) ||
    !i.isLocal())){
   if(this.getSBTypesFrom(tmpA.values()).contains(s))
   {
    this.addEjbInterfaceProvider(r,i,tmpA);
    success = true;
   }else{
    Map<EjbReferenceType, Set<EjbInterfaceType>>
      subTempA = new HashMap<EjbReferenceType,
      Set<EjbInterfaceType>>();
    subTempA.putAll(tmpA);
    this.addEjbInterfaceProvider(r, i,subTempA);
    for(EjbReferenceType sr:s.getEjbReferenceTypes())
    {
     subTempA =
       this.provideEjbReferenceType(sr, subTempA);
     if(subTempA == null) break;
    }
    if(subTempA != null){
     tmpA = subTempA;
     success = true;
    }
   }
  }
 }
 if(success) return tmpA;
 return null;
}
```

Listing 5.2: Recursive Provision of `EjbInterfaceTypes` for an `EjbReferenceType`

If further considerations regarding `i` are meaningful, it is investigated whether `s` is already integrated into the plan in line 12[14]. If this is the case, only the new proposed connection must be integrated into `tmpA` (line 14)[15], and it must be indicated that a provider was found (line 15). If `s` is not yet integrated into the plan, the proceeding in the lines 17 to 33 is analog to that in the lines 8 to 22 of listing 5.1 on page 131 except that the submitted set must be extended with the currently investigated `EjbInterfaceType` (line 21). This is necessary, because scenarios with circular connection proposals would otherwise lead to endless loops. If the execution of the method led at least to one possible provider type for `r`, the resulting data structure is returned (line 35), `null` is returned otherwise (line 36).

5.2.5.4. Plan Adjustment

A plan might contain alternatives for connections to establish or for the provision of desired Java interface types within `a` or `p`, that is, at least one of the particular value sets might contain more than one element. In such a situation a plan would not be unambiguous. To reach unambiguity, planners must remove `EjbInterfaceTypes` from `a` and `p` until each of the value sets of both maps contains exactly one element. The plan supports this through the two methods `removeJavaInterfaceTypeProvider` and `removeReferenceTypeProvider`. The first method only expects the `Ejb-InterfaceType` which should be removed as parameter while the second one additionally requires the `EjbReferenceType` from which the provider

[14] The method `getSBTypesFrom` constructs a set of all SB types from a set of `EjbInterfaceTypes` through iterating over them and requesting their SB type. It is not discussed here any further.

[15] The integration into `tempA` is straight forward: The method `addEjbInterfaceProvider` requests the entry of `r` in `tempA`. If none exists, a new entry set is created and integrated. Afterwards, `i` is integrated into the set. `addEjbInterfaceProvider` is not discussed here any further.

should be removed as parameter. Both methods first analyze whether the set of `EjbInterfaceTypes` from which the submitted interface type should be removed contains only one element. If this is the case, the removal cannot be performed, because this might lead to an infeasible plan. Otherwise, the corresponding entry is removed. Additionally, the `SessionBeanType` belonging to the removed interface provider type is requested and submitted to the method `recalculateReferences` as parameter `s`. This method is responsible for internally removing the corresponding `EjbReferenceTypes` from `a` recursively. The source code of `recalculateReferences` is shown in listing 5.3.

```
1  private void recalculateReferences(
2    SessionBeanType s){
3   if(this.getSessionBeanTypes().contains(s)) return;
4   Set<EjbInterfaceType> is =
5     new HashSet<EjbInterfaceType>();
6   for (EjbReferenceType r:s.getEjbReferenceTypes()){
7    Set<EjbInterfaceType> rs = this.a.remove(r);
8    if(rs != null){
9     is.addAll(rs);
10   }
11  }
12  for(EjbInterfaceType i:is){
13   this.recalculateReferences(i.getSessionBeanType());
14  }
15 }
```

Listing 5.3: Recursive Removal of `EjbReferenceTypes` from *Type Level* Plan

If at least one `EjbReferenceType` of the `SessionBeanType` is still in use, no further adjustments are necessary. This is verified through analyzing the result of `getSessionBeanTypes`, because the result set only contains those SB types which are potential providers of at least one `EjbInterfaceType` (line 3). Otherwise, all entries for the `EjbReferenceTypes` of the considered SB type are removed from `a` (lines 6 to 11). The affected `EjbInterfaceTypes` are stored in a temporary data structure (`is`). For each of the corresponding session bean types the method `recalculate-References` is invoked recursively in the lines 12 to 14. Consequently, after returning from all invocations of the method, the data structures of the plan only contain `EjbInterfaceTypes` which are still needed for the configuration alternatives the plan represents.

5.2.5.5. Application to Case Study

As preparation for the approach at least the *Access Layer* and *Managed Layer* ejb-jar files of the *Transfer* component and the *Managed Layer* ejb-jar file of the *Foundation* component must be integrated into the target system. Figure 5.5 on page 138 shows the proposed connection structure on execution of `addJavaInterfaceType` for the Java interface type `TransferController`, `BankController`, and `StandingOrderController` on a plan intended for the *Access Layer*.

The result is based on the assumption that for each of the required interface types only one possible provider type is known within the system, that is, beyond the required ejb-jar files, no other EJB module type contains `EjbInterfaceTypes` associated with the affected `JavaInterfaceTypes`. In the figure provided interface types are depicted through circles while required interface types are shown as semicircles. All of them are annotated with the names of the corresponding Java interface types. Proposed connections are shown as arrows from the required interface type to the provided interface type. SB types are represented through rectangles with rounded corners, and EJB module type are the rectangles surrounding the including elements. The gray shaded SB type `CustomerControllerBean` is not affected by the plan, because it was not part of any connection proposal during plan construction. Consequently, it would not be reachable, and its configuration could therefore be neglected. The plan would not need any adjustments after submitting the desired `JavaInterfaceTypes`, because its construction would directly lead to unambiguity. It is compli-

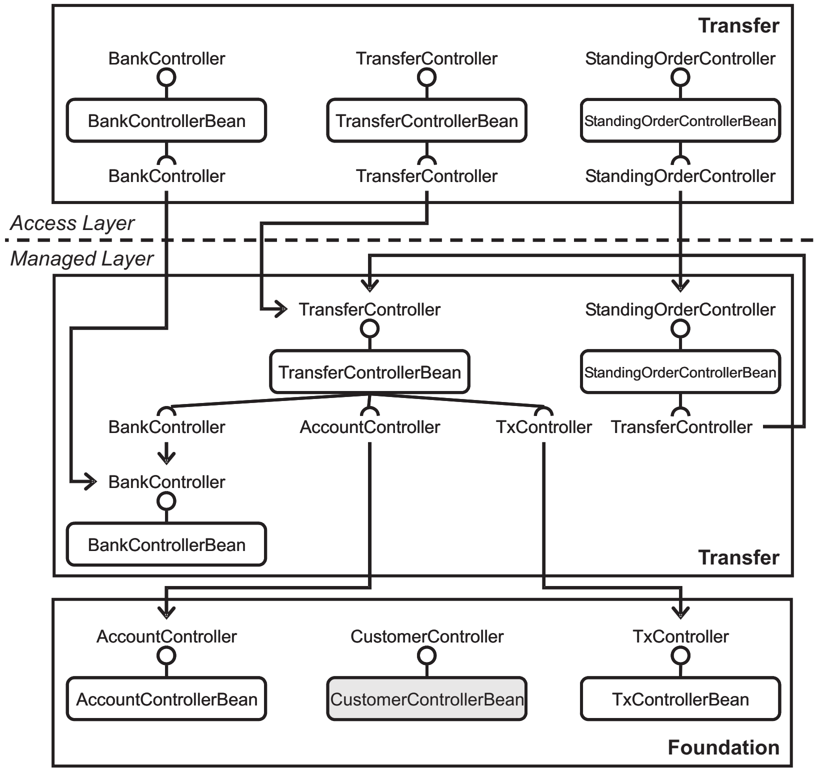


Figure 5.5.: *Type Level* Plan Proposal

ant with the intended structure, as presented in section 4.1.2.

In order to plan the provision of an MDB type on *Access Layer*, a matching MDB type must be found on *Managed Layer*. Afterwards, a *Managed Layer* plan with all `JavaInterfaceTypes` of its `EjbReferenceTypes` must be constructed according the proceeding in the previous sections. A detailed discussion is not provided in this thesis.

5.3. The Deployment Level

On *Deployment Level*, the meta model addresses the architecture of a concrete system. Therefore, the corresponding elements of the meta model are intended to provide sound sensors for inspection of the current configuration of a system with respect to the established connections, as well as the concrete parameter settings of the constituent modules and beans. Furthermore, the API provides comprehensive effectors for compositional and parameter adaptation. Those go partially beyond the opportunities and facilities provided by the EJB standard. This section focuses on the discussion of what is enabled by the infrastructure on *Deployment Level* while abstracting from the underlying conceptual and technological realization.

EJB modules are considered as the constituent elements of a system architecture on *Deployment Level*. They are derived from EJB module types being part of the *Type Level*. Therefore, each EJB module belongs to a corresponding EJB module type. The same holds for each included element of a module such as enterprise beans which are derived from EJB types. Table 5.2 on page 140 contains a compilation of the *Type Level* elements and the related *Deployment Level* elements.
Consequently, most[16] elements on *Deployment Level* can be interpreted as instantiations of *Type Level* elements. In this context, nearly each element of the *Type Level* might be instantiated an arbitrary number of time, because a module type might be deployed more than once within a system. This also includes the case that a module type is not instantiated at all. Each of the related elements of the two levels are associated in the meta model. Therefore, navigation from *Type Level* elements to the corresponding *Deployment Level* elements and vice versa is possible[17].

[16] Only `JavaInterfaceType` and `MethodSpecification` have no corresponding elements on *Deployment Level*.

[17] Navigation from *Type Level* to *Deployment Level* is only possible if corresponding elements

Type Level Element	*Deployment Level* Element
`EjbModuleType`	`EjbModule`
`EnterpriseBeanType`	`EnterpriseBean`
`MessageDrivenBeanType`	`MessageDrivenBean`
`SessionBeanType`	`SessionBean`
`EjbReferenceType`	`EjbReference`
`EjbInterfaceType`	`EjbInterface`
`SimpleEnvironmentEntryType`	`SimpleEnvironmentEntry`
`TransactionDeclarationType`	`TransactionDeclaration`
`SecurityRoleType`	`SecurityRole`
`SecurityPermissionType`	`SecurityPermission`

Table 5.2.: Compilation of related *Type Level* and *Deployment Level* Elements

The remainder of this section is structured as follows: Section 5.3.1 discusses the life cycle of EJB modules. Compositional and parameter adaptation as supported by the meta model are addressed in section 5.3.2 and section 5.3.3. Afterwards, the provided access points to the *Deployment Level* are presented in section 5.3.4. Finally, section 5.3.5 explains the application of the *Deployment Level* API through an example.

5.3.1. The Life Cycle of EJB Modules

A deployed EJB component is represented through an `EjbModule`. The included enterprise beans are derived from the EJB types of the corresponding module type. Enterprise beans stand in a composition association to their corresponding module, because they are created and destroyed with this module. The same holds for required interfaces of beans and

exist.

provided interfaces of session beans, as well as for their associated parameterization settings.

The creation of an EJB module from an EJB module type is supported by the `Container` interface. Therefore, the module type, of which a deployable module should be created, must be submitted to the method `createEjbModule` as single parameter which delivers an `EjbModule` as return value.

During its life cycle an `EjbModule` might pass through different deployment `states`. These states are relevant in the context of configuration and adaptation, as discussed in the following sections. Therefore, EJB modules expose their current state to managing entities. The set of possible states represents an extension of those states considered by the *Java Enterprise Edition 5 Deployment API Specification, Version 1.2* discussed in section 3.4.1. The corresponding state diagram is depicted in figure 5.6.

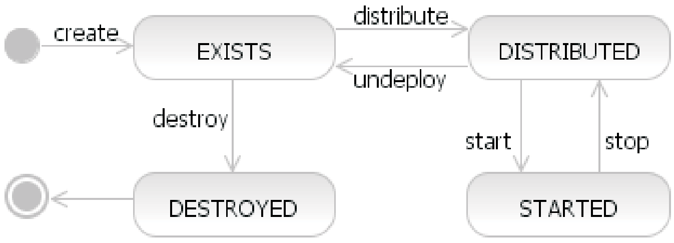


Figure 5.6.: EJB Module Deployment States

The life cycle of an EJB module starts within the state `EXISTS` which is reached after its creation through the `Container` interface. In this state the module is not deployed within its target container, but only exists as representation inside *mKernel*. Consequently, this state is specific to *mKernel* and is not covered by the states considered by the JSR 88.

Through a successful invocation of the `distribute` method upon a module in the state `EXISTS` the module is deployed into the target EJB container and subsequently reaches the state `DISTRIBUTED`. Within this state the module does exist inside the container, but its constituent enterprise beans are not accessible for clients. This state is directly adopted from JSR 88.

In order to make the beans of a module available to clients, its `start` method must be invoked in the state `DISTRIBUTED`. After successful execution, the module is in the state `STARTED` which is also adopted from JSR 88.

In case a module should not be reachable for clients anymore, an invocation of the `stop` method in the state `STARTED` transfers the module back into the state `DISTRIBUTED`. Through a subsequent invocation of the `undeploy` method, the module can be removed from the container and transferred back into the state `EXISTS`.

If a module is not needed anymore and should be removed from the system completely, autonomic entities can invoke the `destroy` method upon that module in the state `EXISTS`. This leads to a removal of the representation from the *mKernel*-managed system and to a transition into the state `DESTROYED` which is specific to the meta model. Within this state no interactions with the *mKernel*-managed system are possible through the affected `EjbModule` reference anymore. Nevertheless, autonomic entities might still hold corresponding references. To allow those entities to request the correct state of the module the `DESTROYED` state was integrated into the meta model. Therefore, no further interactions are necessary to reach the final state of the state machine. It is reached automatically when the last reference to the module representation is destroyed.

5.3.2. Compositional Adaptation

The architecture of a managed system is given through the constituent enterprise beans of its EJB modules and connections between those beans. Figure 5.7 on page 143 shows the corresponding elements of the meta model and their associations on *Deployment Level*.

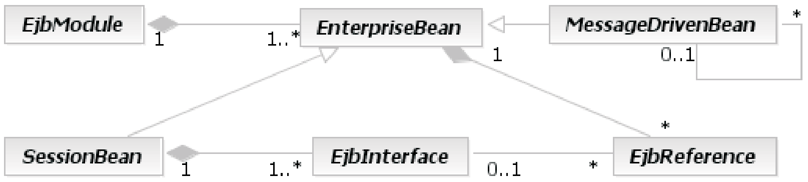


Figure 5.7.: System Architecture Representation

Each EJB module contains at least one enterprise bean (`EnterpriseBean`). Each bean itself is either a message-driven bean (`MessageDrivenBean`) or a session bean (`SessionBean`). Enterprise beans are integral elements of their associated module. Therefore, they stand in a composition association to it. Required interfaces of beans are represented through `EjbReferences`, and provided interfaces of session beans are exposed through `EjbInterfaces`. A composition association is given, because references and interfaces are integral parts of their beans. Associations between `EjbInterfaces` and `EjbReferences` are used to illustrate connections between required and provided interfaces. The circular association of `MessageDrivenBean` represents the connection between MDBs of the *Access Layer* and MDBs of the *Managed Layer*. All of the depicted associations are bidirectional, meaning that the meta model supports navigation in both directions.

mKernel allows the establishment, rerouting, and removal of connections in the architecture of a given system without the need for deployment operations. Consequently, *mKernel* supports dynamic adaptation which goes beyond adaption opportunities, as envisioned by the EJB standard, which assign the determination of connections to the deployment phase. Furthermore, the meta model provides facilities to analyze a system architecture with respect to missing or insufficiently configured connections also considering the deployment state of modules. In this context, *mKernel* treats connections between required and provided interfaces fundamentally different from connections to message-driven beans through message destinations. Therefore, these two kinds of connections are discussed separately within the following two sections.

5.3.2.1. Connections to provided Interfaces

As targets of connections provided interfaces of session beans belonging to the *Managed Layer* might be used. In contrast, provided interfaces of the *Access Layer* are only intended to be used by external clients. In order to declare and establish connections inside a managed system, *mKernel* provides two opportunities, namely the establishment of connections between required and provided interfaces, and global reroutings.

On *Deployment Level*, a required interface might only be connected to a provided interface if the corresponding `EjbReferenceType` and `EjbInterfaceType` are associated with the same `JavaInterfaceType` on *Type Level*. Otherwise, the attempt to establish a connection would be responded with an exception by the API. Consequently, *mKernel* enforces type safeness regarding established connections between required and provided interfaces within a system architecture. This goes beyond the requirements of the EJB standard which does not treat this aspect explicitly. In contrast, the standard solely relies on mapped names for the establishment of connections and thus would theoretically allow the specification of invalid connections.

Beyond the specification of EJB references, the EJB standard allows enterprise beans to use the global namespace to establish connections to session bean instances. *mKernel* supports dynamic adaptation of this namespace through global reroutings. These allow the binding of a provided interface to mapped names inside the global namespace of a container as seen by the subjects of rerouting. In this context, *subject of rerouting* denotes that entity for which the rerouting is declared. If a global rerouting is, for example, defined for an enterprise bean, each lookup in the global namespace with the affected mapped name would result in the obtainment of a reference to an instance of the session bean to which the bound provided interface belongs[18]. Global reroutings might be declared for a single bean, a module as a whole, or a complete system. In this context, more specific settings take precedence over more general ones. If, for instance, the same mapped name is bound to a different target on system level than on bean level, the rerouting on bean level would be used when an instance of that bean performs a lookup with the affected mapped name.

Establishment and Rerouting of Connections *mKernel* supports two variants for the establishment and rerouting of connections, as well as for the declaration of global reroutings. The first variant (*default*) does not only affect the creation of new references on *Instance Level*, but also reroutes existing references to instances of the new target. The second variant (*lazy*) does not consider existing references, but only takes effect for newly created references.

Consequently, both variants are equivalent for the establishment of new connections and reroutings, because in such a situation no references can exist. Furthermore, they are also equivalent if no references exist on *Instance Level* based on the connection or rerouting to change, that is, no enterprise bean instance holds a reference in accordance with the originally established connection or rerouting. Nevertheless, the application of the *default* variant to existing references to stateful SB instances leads to a loss of state for the targets of those references, because *mKernel* does not support the transfer of state from the original to the new target of such an existing reference automatically. This is, because no generic proceeding does exist for mapping the state of the original target to state elements of the new target. The state realizations of the original and the new target

[18] This only holds if the module of the session bean is in state `STARTED` and the bean is properly configured.

might be completely different. Therefore, a context specific proceeding for state transfer would be necessary. Section 5.6, amongst others, addresses this aspect in the context of seamless reconfiguration. As a result of the *default* variant, exceptions or other undesired effects might occur. This would not be the case if the *lazy* variant is applied, because existing references are kept as long as they are needed. Nevertheless, the application of the *lazy* variant might not always be meaningful, because some situations might demand for a complete transfer of connections, including existing references. This might, for instance, be the case if the original target on *Deployment Level* is erroneous or corrupted, or the underlying database must not be used by instances of the original and the new target SB concurrently. The application of the *default* variant to stateless targets would not lead to the problems mentioned for the stateful case, because instances of stateless session beans do not keep any conversational state. Consequently, holders of references would not notice any difference after application, because their references would still provide access to an arbitrary instance of a target interface implementation[19].

Summarizing, none of the two variants provides a solution for seamless rerouting in all possible situations. Moreover, there are constellations conceivable where none of them might be applied as single solution successfully, for instance, if multiple connections should be rerouted atomically or if references to stateful targets on *Instance Level* should be rerouted transparently. Nevertheless, the two variants presented here are easy to use opportunities for realizing dynamic reconfiguration of a system architecture. A more comprehensive foundation for seamless rerouting provided by the meta model is discussed in section 5.6. Additionally,

[19] This statement only holds if the new target of connection replaces the original target in a completely consistent way. A counterexample would be if the new target of connection uses a transformed data source with different identifiers for its elements. In this case exceptions or undesired behavior might be the result of ongoing interactions if the original identifiers are used by clients.

section 7.1 presents an extension of *mKernel* for addressing autonomic self-configuration.

Beyond the opportunity to establish or change connections and reroutings autonomic entities are enabled to disconnect established connections or remove global reroutings.

Reference State Each `EjbReference` exposes a corresponding reference state in order to indicate its usability. Such a reference state provides information about the connection structure regarding the provision of the required functionality.

- `DISCONNECTED`: A disconnected reference is not associated with an `EjbInterface`.
- `CONNECTED`: This state is given if the reference is associated with an `EjbInterface`, but the corresponding session bean cannot be used properly. This might, for instance, be the case if it is itself missing connections for its EJB references.
- `ACTIVATABLE`: An activatable reference is associated with an `EjbInterface`, and the `EjbModule`, the reference belongs to, is in state `DISTRIBUTED` or `STARTED`. Additionally, all `EjbReferences` of the session bean belonging to the associated interface are either in state `ACTIVATABLE` or `ACCESSIBLE` recursively, if such exist. Finally, at least one module being part of the transitive closure given through the reference-interface-connections is in state `DISTRIBUTED`. Consequently, the usability of the corresponding reference can be established solely through starting a set of modules.
- `ACCESSIBLE`: For this state the same conditions hold as for `ACTIVATABLE`, but all affected modules are in state `STARTED`. Consequently, the reference can be used directly without any further demands for compositional adaptation or deployment actions.

Each enterprise bean provides an aggregated reference state which is derived from the states of all of its required references. According to the order of reference states from above, each state is analyzed whether at least one reference is in it. If this is the case, the corresponding state is given. Consequently, the aggregated state indicates the most evident need for action for making the corresponding bean usable.

Interface State Analog to `EjbReferences` each `EjbInterface` has a corresponding interface state which covers information of how the interface is integrated in a system architecture. It might be used to identify whether a particular `EjbInterface` is used as target of connections and to estimate the impact of manipulations regarding the associated session bean and module, respectively.

- `NOT_REFERENCED`: No `EjbReference` or global rerouting is connected to the interface. Regarding the interface changing the state of the corresponding module would not have any effects on other modules.
- `PASSIVELY_REFERENCED`: Only references are connected to the interface of which the corresponding modules are in the states `EXISTS` or `DISTRIBUTED`. The same holds for global reroutings of which the interface is the target. Changing the state of the module the interface belongs to would only have indirect effects, for instance, at least one `EjbReference` would become `DISCONNECTED` in case of removal. As the corresponding modules are not in state `STARTED`, this would not have any direct effects on available beans.
- `REFERENCED`: At least one reference or rerouting is connected to the interface of which the corresponding module is in state `STARTED`. Therefore, a state transition of the module, the considered interface belongs to, would have direct impact on beans accessible by internal or external clients.

A session bean grants access to an aggregated interface state derived from the states of all of its provided interfaces. The calculation is performed analog to the way the reference state of a bean is derived. The aggregated interface state supports the identification of the most evident impact of changes affecting the session bean either through state transition of the corresponding module or through changes of required interface connections. The interface state does not cover any information about the actual usability of a provided interface. This information could be obtained through requesting the reference state of the corresponding session bean.

5.3.2.2. Connections to Message-Driven Beans

Message-driven beans are envisioned by the EJB standard as targets of asynchronous interactions. In this context, they are bound to JMS-based message destinations during deployment and are treated as message receivers by their execution environment. The JMS standard does only define different types of transferred messages such as messages for object transfer (`javax.jms.ObjectMessage`) or text transfer (`javax.jms.TextMessage`). It does not contain any opportunity which allows to restrict the usage of message destinations such as type, structure, or content of sent messages. Therefore, the standard does not support the definition of any kind of interfaces for message destinations. This aspect is also not addressed by the EJB standard. Consequently, *mKernel* cannot support type safeness for MDB-based connections as it was possible for required and provided interfaces based on Java interfaces. Furthermore, the EJB standard does not define that a message destination to which an MDB is bound must be exclusively reserved for that bean. Consequently, it is possible that a message destination is jointly used by multiple MDBs or even by MDBs and other message receivers directly using JMS.

mKernel supports the binding to message destinations only for *Access Layer* MDBs. This might be performed in accordance with the EJB standard through setting the mapped name of an MDB to the same name as that of the message destination from which messages should be received.

MDBs of the *Managed Layer* might be dynamically bound to and unbound from *Access Layer* MDBs. In this context, an arbitrary number of *Managed Layer* MDBs might be bound to each *Access Layer* MDB. As MDB instances do not hold client-specific conversational states and must be equivalent to potential clients, *mKernel* only supports *default* rerouting for MDBs. An *Access Layer* MDB acts as an adapter for messages arriving at the message destination it is bound to, that is, it enriches the messages with information needed by *mKernel* and forwards them to their targets on *Managed Layer*.

From the point of view of an *Access Layer* MDB the connected *Managed Layer* MDBs represent references to which messages might be transferred. Therefore, the aggregated reference state of an *Access Layer* MDB is derived from the reference states of all of its connected MDBs. If no *Managed Layer* MDB is bound to an *Access Layer* MDB, the corresponding state would be `DISCONNECTED`. For all other situations the state is calculated in accordance with the discussion in the previous section.

Inside the source code of enterprise beans of the *Managed Layer* connections to message destinations might be established, and messages might be sent to them. These messages might be intended for MDBs of the *Managed Layer*, but also for direct users of JMS[20]. In this context, the interaction schema differs from that of interactions with session beans in two major aspects. First of all, direct connections between EJBs through message destinations are not considered by the meta model. Secondly, connections to MDBs through message destinations are mediated through MDBs of the *Access Layer*, because only those are bound to message destinations directly. Consequently, a rerouting of connections through message destinations is not as flexible as for Java interface based connections: If a *Managed Layer* MDB is unbound from or a new MDB is bound to

[20] This might also include bean instances which register at a destination as message receiver from inside their source code.

an *Access Layer* MDB, this might affect all senders of the corresponding destination. In contrast, the rerouting of messages of a single sender or a group of senders is not supported.

5.3.3. Parameter Adaptation

Section 5.2.3 presented parameterization options considered on *Type Level*. The corresponding elements of a concrete model represent settings derived from an ejb-jar file which was integrated into an *mKernel*-managed system. All of those settings are adopted during the creation of an EJB module. Figure 5.8 depicts parameterization opportunities provided on *Deployment Level* of the meta model. The rectangle in the upper right corner of the figure covers elements which belong to the *Type Level* of the meta model. Those are referred to in the context of transaction and security settings.

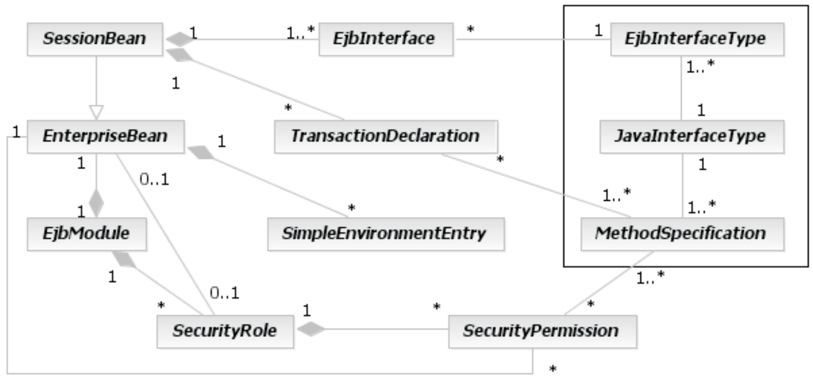


Figure 5.8.: *Deployment Level* View on Parameter Adaptation

mKernel internally makes use of the DD of a module for parameterization which can only be manipulated before the integration of the module into its target container, that is, if it is in state EXISTS. In this context, the EJB standard does not cover any opportunities for parameter adaptation of deployed modules. Therefore, the meta model supports parameter adaptation of modules which are in the state EXISTS except for simple environment entries. Those might be manipulated independent from the state of the corresponding module which represents an extension of the configuration opportunities of the standard. The remainder of this section discusses the different opportunities for parameter adaptation provided by the API on *Deployment Level*.

Mapped Names Section 3.1.2.1 mentioned the global namespace as part of the EJB standard. This namespace builds the foundation for the publication of enterprise beans at a certain mapped name and the establishment of connections to instances of them. The inspection and manipulation of mapped names are supported by *mKernel* only for enterprise beans belonging to the *Access Layer*, because only instances of those should be accessible to external clients. In contrast, connections to beans of the *Managed Layer* should only be permitted inside an *mKernel*-managed system. The corresponding opportunities for compositional adaptation were presented in section 5.3.2 and do not directly rely on the global namespace of the container. In contrast, connections between required and provided interfaces are established directly based on the corresponding elements. For global reroutings mapped names are not set for the targets of reroutings, but in the context of a rerouting subject. Therefore, an inspection or manipulation of mapped names of *Managed Layer* beans is obsolete.

Simple Environment Entries SEEs (SimpleEnvironmentEntry) might be inspected and set for enterprise beans during the whole life cycle of the corresponding module, except the DESTROYED state. When an SEE should be changed the new value of the entry must be of the same type as the one defined for the corresponding SEE type on *Type Level*. Otherwise, the API would react with an exception.

Analog to the creation, manipulation, and removal of connections and reroutings, the manipulation of SEEs is supported by *mKernel* through two variants. For the *lazy* variant value changes take effect only for new instances of the affected enterprise bean. Thus, lookups in the local namespace performed by instances which were created before changing the SEE value would return the original value. For new instances lookups or dependency injection would lead to the obtainment of the new value. For the *default* variant the new value of an SEE is directly integrated into the local namespace of the affected bean, overwriting the original one. Therefore, each result of a lookup in the local namespace and each dependency injection would be based on the new value. The *lazy* variant might, for example, be applied meaningfully if stateful session bean instances perform different actions across multiple invocations based on the value of an SEE. If the source code of the bean demands that all of these actions must performed based on the same value for keeping consistency, a new value for the SEE should be set based on the *lazy* variant. On the other hand, the *default* variant should be chosen if a new value should take effect as soon as possible and no consistency problems could arise. The two variants provide a certain level of freedom to autonomic entities to react to context specific aspects. Nevertheless, they do not cover solutions for all conceivable situations. It is, for example, not possible to change more than one SEE value atomically. These more complex situations are addressed in section 5.6 in the context of seamless reconfiguration.

Transaction Settings Transaction settings are represented for message-driven beans and session beans differently.

For MDBs a `javax.ejb.TransactionAttributeType` might be requested or set directly through corresponding methods. This enum is also used by the EJB standard for specifying transaction settings through annotations. An MDB type must contain exactly one method for receiving messages through JMS. To this method, a submitted or returned transaction attribute value applies.

For each session bean a set of transaction declarations (`Transaction-`

`Declaration`) might be requested to inspect the current transaction settings. Such a declaration exposes a `TransactionAttributeType` value and refers to a set of *Type Level* `MethodSpecifications`. Analog to the *Type Level* representations of transaction settings a transaction declaration is referring to more than one method specification if their signatures are equal, that is, they have the same name, return type, and parameter types. For that case a transaction declaration cannot be assigned individually, because *mKernel* relies on the EJB standard for realizing declarations in a system context which does not support transaction settings based on Java interfaces. In order to specify a transaction setting, the interface `SessionBean` provides a corresponding method which accepts a `TransactionAttributeType` and a `MethodSpecification` as parameters. The method specification must be connected to the session bean through the connections between `MethodSpecification`, `JavaInterfaceType`, `EjbInterfaceType`, `EjbInterface`, and `SessionBean`, that is, the corresponding method must be provided by the session bean in a system context. In case of signature equivalence, the specification of a transaction setting might implicitly also affect method specifications of other Java interfaces, as discussed above.

Security Settings EJB modules might define an arbitrary number of security roles (`SecurityRole`) which can be identified based on their name which is unique on module level. To each of these roles an arbitrary number of security permissions (`SecurityPermission`) can be assigned. A permission relates to an enterprise bean which must be part of the module. For SBs a method specification must additionally be referenced which indicates the method to which the permission should be applied. This method must be provided by the SB through its `EjbInterfaces` the same way, as discussed for transaction settings in the previous paragraph. If more than one `MethodSpecification` is associated with a permission this is because of the same reason as for transaction settings. For MDBs no method specification is referenced. It can be assumed that the permission relates to the method for receiving messages from JMS message destinations. Security roles can be created or removed through corresponding methods provided by the interface `EjbModule`. Security roles themselves allow the declaration of permissions based on a method specification. This might implicitly induce the application to other method specification in case of signature equivalence. Furthermore, permissions might be removed from security roles. Finally, each enterprise bean might refer to a single security role which should be used as foundation for method invocations performed by instances of that bean in a system context, that is, these invocations should be treated as if they were invoked by a principal belonging to that role. This might, for example, be necessary if instances perform invocations on other bean instances which demand this role for execution.

MDB Message Selector For their binding to JMS-based message destinations message-driven beans might define a message selector. Analog to the *Type Level* the selector of an MDB is represented as `java.lang.String` and can be requested through a corresponding method. Furthermore, a new selector can be set and an existing one can be changed.

Vendor specific Extensions All parameterization options discussed up to now directly relate to the EJB standard. Nevertheless, it might be possible that certain aspects require for vendor specific artifacts such as a file containing the mapping of security roles to the security domain of the target container, as discussed in section 3.2.2. Therefore, *mKernel* allows the integration, inspection, and removal of vendor specific artifacts which are integrated into the ejb-jar file used for module deployment. An artifact can be submitted to the module representation as byte array. Alternatively, the FQN of the artifact file might also be submitted if the API is used in an environment which allows file access.

5.3.4. Deployment Level Access Points

Analog to the *Type Level* the `Container` interface can be used to request references to EJB modules and enterprise beans as access points to the *Deployment Level*. For both of them a collection of all known representations can be requested through a corresponding method, or a specific representation can be selected based on its unique identifier. The corresponding methods are depicted in table 5.3.

Interface	**Specific Representation**	**All Representations**
`EjbModule`	`getEjbModule`	`getEjbModules`
`EnterpriseBean`	`getEnterpriseBean`	`getEnterpriseBeans`

Table 5.3.: Access Points to Elements of the *Deployment Level*

As discussed in section 5.3.2.1, global reroutings can be, amongst others, defined on system level. A mapping from all globally rerouted mapped names to `EjbInterfaces` can be requested from the *Container* interface. Furthermore, new global reroutings can be declared and existing ones can be changed or removed.

5.3.5. Application Example

The *Deployment Level* part of the meta model might, amongst others, be used to realize *Type Level* plans, as discussed in section 5.2.5. This is presented in the remainder of this section. As for the *Type Level* example, the example presented in this section does not aim to be a holistic solution for autonomic deployment, but should illustrate the opportunities provided by the *Deployment Level* of the API. As it is able to realize plans based on instances of `TypeLevelPlan`, it provides a generic approach which is not limited to the case study. Only the relevant parts of the corresponding source code are shown in this section. The complete source code can be found in appendix B.

As foundation of the example the class `DeploymentLevelPlan` is used. During construction an instance of `TypeLevelPlan` must be submitted which represents an unambiguous *Type Level* plan. An instance of `DeploymentLevelPlan` is able to realize the structure proposed by the submitted `TypeLevelPlan` through the creation, deployment, and activation of the necessary modules, and the establishment of the required connections among the affected session beans. It implements a stepwise proceeding allowing clients to perform additional configurations, if necessary. Internally, it holds a reference to the type level plan (`tp`), a reference to a `Container` (`c`), and a set of `EjbModules` which are created during plan realization (`ms`).

It is assumed that for none of the `EjbModuleTypes` being considered by `tp`, corresponding EJB modules do exist within the target system. Otherwise, it might become necessary to take the existing system architecture into account during plan realization. Although this would be possible it is excluded from the example, because it would complicate the source code disproportionately.

5.3.5.1. Inspection Opportunities

For the inspection of a concrete plan `DeploymentLevelPlan` provides the methods depicted in the following list.

1. `getSessionBeans`: A set of all session beans which correspond to SB types affected by `tp` is provided by this method. Internally, all session beans of all affected modules are analyzed whether the corresponding SB type is contained in the set of the SB types being part of the type level plan.
2. `getSessionBeanForSessionBeanType`: The method returns a representation of the session bean which is a *Deployment Level* instance of the submitted SB type. The method is realized based on an iteration over the result set of `getSessionBeans`. If the corresponding

type of the current session bean is equal to the submitted one, the desired SB is found.

3. `getEjbModules`: This method returns all modules which were created during plan execution, that is, the entries of `ms`.
4. `getEjbModuleForEjbModuleType`: As result of this method the `EjbModule` is returned which was created from the submitted module type during plan realization. The method is implemented analog to `getSessionBeanForSessionBeanType`.
5. `getEjbInterfaceForJavaInterfaceType`: This method can be used to obtain the provider of one of the `TypeLevelPlan` goals in a system architecture. Internally, the corresponding `EjbInterface-Type` (`eit`) is requested from the *Type Level* plan. Afterwards, all session beans of all modules are analyzed whether the associated type of one of their provided `EjbInterfaces` is equal to `eit`. If this is the case, the requested `EjbInterface` is found and returned.

Through these methods all necessary information about a `Deployment-LevelPlan` are provided. Method 1 and 3 provide all SBs and modules which might demand for configuration. In order to obtain the *Deployment Level* pendant for a module type or an SB type of `tp`, the methods 2 and 4 can be used. Method 5 can be used to obtain those `EjbInterfaces` which were the ultimate goals of the plan execution. The connections between `EjbReferences` and `EjbInterfaces` do not need to be exposed by a plan, because they can be requested from the API directly.

5.3.5.2. Module Creation and Compositional Adaptation

The execution of a deployment plan is oriented at the life cycle of EJB modules as presented in section 5.3.1. Therefore, `DeploymentLevelPlan` provides the three methods `create`, `deploy`, and `start` to its clients. Each of them leads to the corresponding state transitions of all EJB modules which are affected by the plan, that is, the entries of `ms`.

The method `create` is responsible for the creation of `EjbModules` and for the establishment of connections between the constituent session beans where necessary. It is implemented as depicted in listing 5.4.

```
 1 public void create() {
 2  for(EjbModuleType m:
 3     this.tp.getModuleTypesToDeploy()){
 4   this.ms.add(this.c.createEjbModule(m));
 5  }
 6  for(SessionBean s:this.getSessionBeans()){
 7   for(EjbReference r:s.getEjbReferences()){
 8    EjbInterfaceType i =
 9       this.tp.getConnectionAlternatives(
10       r.getEjbReferenceType()).iterator().next();
11    boolean success = false;
12    for(SessionBean sp:this.getSessionBeans()){
13     for(EjbInterface ip:sp.getEjbInterfaces()){
14      if(ip.getEjbInterfaceType().equals(i)){
15       r.connectTo(ip);
16       success = true;
17       break;
18      }
19     }
20     if(success) break;
21    }
22   }
23  }
24 }
```

Listing 5.4: Creation of EJB Modules and Connection Establishment

As first step during creation for all module types of the *Type Level* plan corresponding modules are created and integrated into `ms` (lines 2 to 4), because their constituent session beans are needed in the subsequent steps for connection establishment. Afterwards, all required interfaces (`r`) of all affected session beans (`s`) are connected to provided interfaces (lines 5 to 21). Therefore, at first the `EjbInterfaceType` (`i`) to which the `EjbReferenceType` of the current `EjbReference` should be connected according to `tp` is identified in the lines 7 and 8. The `boolean` variable `success`, which is created in line 9, is used to identify whether the particular reference was connected successfully. Initially, this is not given (`false`). Afterwards, all provided interfaces (`ip`) of all session beans of the plan (`sp`) are analyzed whether they are the target of connections for `r`. This is the case if their corresponding `EjbInterfaceType` is equal to `i` (line 12). If this is given, the connection is established (line 13), and the establishment is kept in `success` (line 14). Because of the successful connection establishment no further `EjbInterfaces` and providing session beans need to be analyzed (lines 15 and 18). After finishing execution of `create` all required modules are in state `EXISTS` and all connections proposed by `tp` are established. Before this method is executed, the methods for inspection would not return any results, because none of the requested elements would exist. Because of the modules being in state `EXISTS` all kinds of parameter and composition adaptation opportunities of *mKernel* might be used, if desired.

5.3.5.3. Parameter Adaptation

As foundation for the example in this section the resulting `TypeLevelPlan` from section 5.2.5.5 is taken. This must be submitted to the new `DeploymentLevelPlan` (`d`) during construction. Afterwards, the method `create` presented in the previous section must be invoked on `d` to initiate the creation of the *Deployment Level* elements. Listing 5.5 on page 161 identifies all *Access Layer* modules of the plan and configures them.

The presented configuration takes into account that *Access Layer* modules are intended to be provided over a long timespan while the `EjbReferences` of their constituent session beans might be reconnected during their life cycle. In this context, a redeployment of *Access Layer* modules to match with changed requirements from the *Managed Layer*, for example, regarding security or transaction settings, is undesired because of the loss of availability even for a short timespan. Therefore, the *Access Layer* configuration is kept as generic as possible. This part of the overall configuration is not specific to the case study, but represents a generic proposal.

```
1  for(EjbModule m:d.getEjbModules()){
2   if(!m.getEjbModuleType().isManagedLayer()){
3    for(SecurityRole r:m.getSecurityRoles()){
4     m.removeSecurityRole(r);
5    }
6    for(EnterpriseBean b:m.getEnterpriseBeans()){
7     if(d.getSessionBeans().contains(b)){
8      SessionBean s = (SessionBean) b;
9      s.setMappedName(s.getEjbInterfaces().iterator().
10        next().getEjbInterfaceType().
11        getJavaInterfaceType().
12        getFullyQualifiedClassName());
13     for(EjbInterface i:s.getEjbInterfaces()){
14      JavaInterfaceType j = i.getEjbInterfaceType().
15        getJavaInterfaceType();
16      for(MethodSpecification ms:j.
17        getAllProvidedMethods()){
18       s.setTransactionDeclaration(ms,
19         TransactionAttributeType.SUPPORTS);
20      }
21     }
22    }
23   }
24  }
25 }
```

Listing 5.5: *Access Layer* Module Configuration for Case Study

As first configuration step all security roles `r` of each module are removed in the lines 3 to 5. Implicitly all associated permissions of session beans are also deleted, if any exist. As *Access Layer* modules do not cover any business logic, but only forward invocations, this configuration would not imply any security risks. Consequently, the affected *Access Layer* SBs would not induce any additional security requirements beyond those of the connected *Managed Layer* SBs. The advantage of this proceeding lies within the freedom to neglect security aspects in case of reconnection. Afterwards, only those beans of the considered module are configured which are part of `d` (lines 6 and 7). If a bean is affected by the plan, it must be a session bean (`s`), because `DeploymentLevelPlans` do not consider MDBs. In the lines 9 to 12 the mapped name of each `s` is set to the FQN of the first `JavaInterfaceType` which it provides through an `EjbInterface`. Afterwards, the transaction attributes of all externally provided methods are set to `SUPPORTS` in the lines 13 to 20. This is done, because this setting provides the highest level of freedom regarding meditated interactions between external clients and instances of *Managed Layer* session beans, that is, the transaction settings of clients are forwarded. In combination, the settings performed in the above listing lead to a configuration which reaches a degree of freedom for reconnection as high as possible. Only the mapped names of the affected SBs represent a decision which might lead to the need for deployment operations when another interface – maybe a different revision of the same interface or other interfaces – should be provided at that name in the global namespace of the managed container.

For the configuration of the *Managed Layer* modules the default configuration is kept. Only the simple environment entry `bankName` of the `TransactionControllerBean` must be set as depicted in listing 5.6.

```
1  for(SessionBean s:d.getSessionBeans()){
2   if(s.getSessionBeanType().getEjbName().
3      equals("TransferControllerBean") &&
4      s.getEjbModule().getType().isManagedLayer()){
5    SimpleEnvironmentEntry see =
6      s.getSimpleEnvironmentEntry("bankName");
7    see.setValue("Duke's Managed Bank");
8    break;
9   }
10 }
```

Listing 5.6: *Managed Layer* Module Configuration for Case Study

Here, all session beans which are affected by `d` are analyzed in the lines 1 to 10. If the name of the corresponding SB type is ”*TransactionControllerBean*” and the corresponding module is a *Managed Layer* module, the desired session bean is found (lines 2 to 4). After identification the SEE with the name *"bankName"* is requested from the SB (lines 5 and 6) and its value is set to *"Duke's Managed Bank"* (line 7). Afterwards, the configuration of the *Managed Layer* is finished (line 8).

5.3.5.4. Plan Activation

After finishing parameter adaptation the results of the plan must be deployed and started. This can be performed through execution of the two methods `deploy` and `start` on `d` subsequently. Internally, both methods iterate over the set of affected modules and execute the corresponding methods upon them. Listing 5.7 shows the source code of the deploy method. The method `start` is implemented analog.

```
public void deploy(){
 for(EjbModule m: this.ms){
  m.deploy();
 }
}
```

Listing 5.7: Module Deployment

After finishing the execution of both methods, the internal architecture of the managed system is configured properly, and the *Access Layer* modules are accessible to clients. Persistence management was not considered during planning. It was assumed that the underlying data sources do exist at least before the `deploy` method was invoked. Furthermore, it was assumed that all necessary configuration settings were performed outside *mKernel*, for instance, as integral part during uploading the module types into the system or through invocations of the method `setMetaInfArtifact`[21] on the affected modules before deployment.

[21] This method allows the integration of arbitrary, file-based metadata into modules. It is intended to allow the customization of modules through vendor-specific artifacts before deployment.

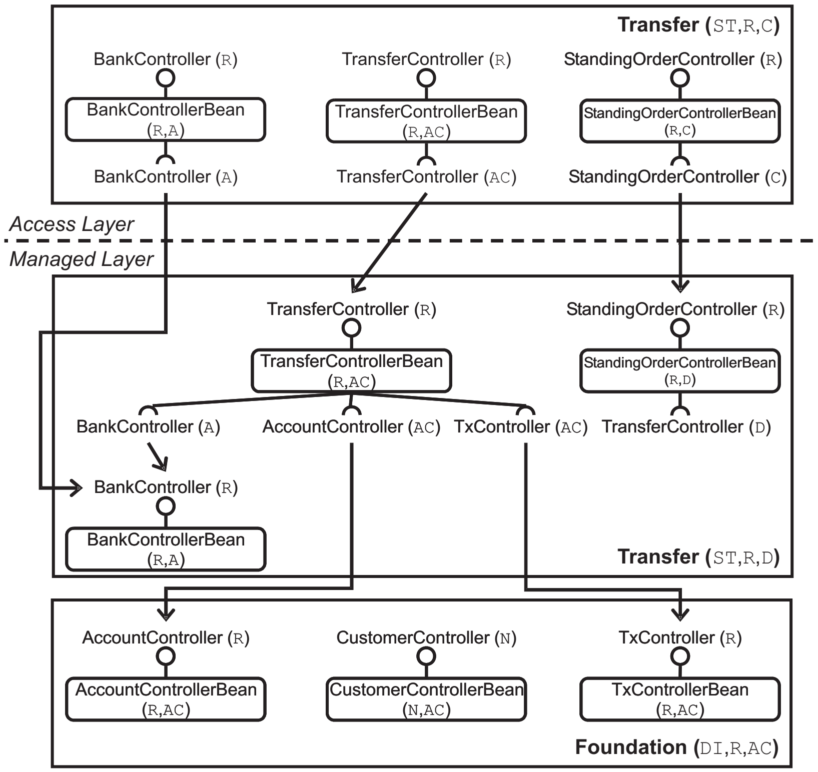
5.3.5.5. Element States

Within the sections 5.3.1 and 5.3.2 different types of element states were introduced. These are explained in combination through a summarizing example in this section. Figure 5.9 on page 165 shows the three modules which were considered in the previous sections. The figure is an adjusted version of figure 5.5 on page 138. Each of the elements is annotated with those states which might be requested from its *mKernel* representation. Module names are annotated with their deployment state, their references state and their interfaces state in brackets. Names of session beans are followed by the corresponding interfaces state and references state in brackets. Finally, interfaces and references are annotated with their corresponding state in brackets. In the remainder of this section not all of the states in figure 5.9 are discussed, but only representative ones.

The reference state of *Access Layer* `EjbInterfaces` is always `REFERENCED`. As *mKernel* concentrates on the business tier, it cannot identify whether clients are connected to interfaces provided on *Access Layer*. Nevertheless, those interfaces are exposed to external clients and thus have great influence on the availability of a managed system. Therefore, they are treated as if they are referenced by default. According to the discussion in section 5.3.2.1 the resulting states of the corresponding session beans and modules are also `REFERENCED`.

On *Managed Layer*, the provided interface of `BankControllerBean` (`BankController`) is referenced by two `EjbInterfaces`. Both of them belong to session beans which are themselves part of modules in state `STARTED`. Therefore, the interface is in state `REFERENCED` which results in the same state for the corresponding SB. As `BankControllerBean` does not demand for any `EjbReference` and the corresponding module is in state `STARTED`, the interface state of the SB is `ACCESSIBLE`, that is, instances of the bean might be used properly by clients[22]. Finally, the state of the con-

[22] This statement is based on the assumption that the bean is configured appropriately.



State Abbreviations

Deployment States	Reference States	Interface States
ST STARTED	A ACCESSIBLE	R REFERENCED
DI DISTRIBUTED	AC ACTIVATABLE	N NOT_REFERENCED
	C CONNECTED	
	D DISCONNECTED	

Figure 5.9.: *Deployment Level* States Example

nected `EjbReferences` results from the state of the `BankControllerBean` and is also `ACCESSIBLE`.

The provided interface of `CustomerControllerBean` is not referenced by any `EjbReference`. Therefore, its state is `NOT_REFERENCED`. The SB itself does not provide any other interfaces which results in its interface state being NOT_REFERENCED. Furthermore, it belongs to a module in state DISTRIBUTED and does not demand for any references. Therefore, its reference state is ACTIVATABLE meaning that it might be brought to usability solely through starting its module.

The reference state of TransferControllerBean (ACTIVATABLE) on *Access Layer* results directly from the corresponding state of the connected session bean on *Managed Layer* which itself derives its state from all of its references. In this case, AccountController and TxController are the relevant ones, because their states demand for adjustments of the architecture in order to reach availability, for example, through starting the *Foundation* module.

If the deployment state of the *Transfer* module on *Access Layer* would be changed to DISTRIBUTED the interface states of the referenced SBs on *Managed Layer* would become PASSIVELY_REFERENCED. Nevertheless, the state of the corresponding module would still be REFERENCED, because BankController would still be referenced by a session bean of a STARTED module.

Finally, the reference state of StandingOrderControllerBean on *Managed Layer* results from its single required interface which is in state CONNECTED. This state indicates that the reference is connected, but that configurations beyond starting modules are necessary. In this case, the connected SB StandingOrderControllerBean on *Managed Layer* is missing a connection for its reference and is therefore in state DISCONNECTED.

5.4. The Instance Level

On the *Instance Level* of the meta model instances of enterprise beans and interactions among them are considered in an ex post manner, that is, the *Instance Level* does not provide any opportunities to engage in ongoing interactions. In contrast, it supports model based analyses of logged information. Analyses of invocations on *Instance Level* might address two aspects, as well as a combination of them. First of all, the interplay of enterprise bean instances can be analyzed. Secondly, the concrete context of a single invocation can be the target of investigation, for example, regarding a thrown exception. The *Instance Level* considers enterprise bean instances from an external point of view. It does not provide insight into the internals of invocation execution with respect to interactions between the constituent elements of an instance itself such as invocations on instances of helper classes. In contrast, only invocations are considered which either reach bean instances from external clients or which are performed from inside the context of an invocation upon an instance of another bean. Consequently, *mKernel* represents interactions between bean instances in a black box manner through the representation of their externally observable behavior. Therefore, *mKernel* does not aim to provide a level of investigation as low as possible, but is designed to keep a balance between the provided information and the efforts necessary for its collection, storage, and provision.

The following two sections discuss the two analysis aspects separately. Afterwards, section 5.4.3 provides an overview of how collection of information on *Instance Level* might be controlled by autonomic entities. The application of the *Instance Level* API is finally presented based on the case study in section 5.4.5.

5.4.1. Representation of Interactions

The *Instance Level* mainly addresses enterprise bean instances and method invocations, as well as specializations of them. In this context, interactions among bean instances are exposed for analysis purposes. The corresponding elements of the meta model are depicted in figure 5.10 on page 168. The three elements at the top – `EnterpriseBeanInstance`, `SessionBeanInstance`, and `MessageDrivenBeanInstance` – represent instantiations of the corresponding *Deployment Level* elements within a running system. Navigation from *Deployment Level* elements to *Instance Level* elements and vice versa is supported by the API, for example, from an instance of `SessionBean` to corresponding instances of `SessionBeanInstance`. These associations are not depicted in the figure. `EnterpriseBeanInstances` stand in a composition association with the corresponding `EnterpriseBeans` on *Deployment Level*, that is, the destruction of an `EjbModule` leads to the destruction of all constituent `EnterpriseBeans` which itself leads to the destruction of all corresponding elements on *Instance Level*.

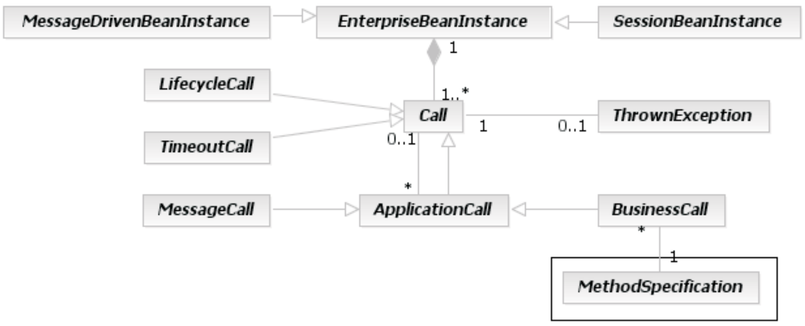


Figure 5.10.: *Instance Level* Overview

Each bean instance on *Instance Level* is associated at least one `Call`. These calls represent method invocations of which the bean instance has been the target and which were logged by *mKernel*. They are ordered in accordance with the order they were executed by the bean instance. In this context, the non-reentrancy property of enterprise beans allows a total ordering. The list of calls of a single bean instance is called *Call History* of that instance in the following. Regarding a single call navigation in both directions of the corresponding history is supported, that is, it is possible to identify the predecessor and successor calls within the same call history, if these exist. This is not depicted in the figure.

There are different specializations of calls considered on *Instance Level*, namely `LifecycleCalls`, `TimeoutCalls`, and `ApplicationCalls`. Life cycle calls are used to represent all invocations which were performed by the container during life cycle transitions of a bean instance, as discussed in section 3.1.3. They expose the corresponding invocation type based on the enum `LifecycleCallType` such as `LifecycleCallType.POST_CONSTRUCT`. Timeout calls depict invocations which were executed in the context of the timer service on timer expirations, as discussed in section 3.2.3. Application calls are further specialized into `MessageCalls` and `BusinessCalls`. A message call is used to represent the invocation on a message-driven bean instance on receiving a message. Business calls depict invocations on session bean instances which were performed through provided interfaces for accessing the encapsulated business logic, that is, local, local business, remote, and business interfaces. In this context, a business call is associated with the corresponding method specification from the *Type Level* through which the invocation was performed. The rectangle in the lower right corner of the figure highlights that `MethodSpecification` does not belong to the *Instance Level*. In case an invocation resulted in an exception, this can be identified through the association with a `ThrownException`. Each `ThrownException` is specific for the associated call and comprises the corresponding information such as its stack trace.

The execution of an invocation might have led to nested invocations on other bean instances. This is depicted in figure 5.10 through the association between `Call` and `ApplicationCall` which represents a hierarchy. It results from the fact that each application call might be invoked from inside of at most one other call. Furthermore, there are no cycles possible, because no invocation could have led to itself directly or transitively. Such a hierarchy is called *Call Chain* within this thesis. The API supports upward and downward navigation within call chains, that is, it is possible to obtain references to representations all invocations which were executed during execution of concrete invocation, its *Sub-Call*s. Furthermore, for

`ApplicationCalls` a representation of the invocation which has led to the current invocation might be requested (*Super-Call*), if one exists.

`EnterpriseBeanInstances` and `Calls` do also cover unique identifiers and expose them to clients through corresponding methods. Furthermore, an enterprise bean instance enables clients to request the representation of a single call through submitting its unique identifier. Therefore, it is, for example, possible for managing entities to store the necessary identifiers for resuming analyses at a given position or to exchange information about situations efficiently without the need to transfer complex object hierarchies.

5.4.2. Invocation Information

Beyond methods for navigation along associations the interface `Call` supports analyses of invocations with respect to security, transactions, and performance. Nevertheless, the API is not intended to be used for complex calculations based on huge numbers of invocations. In contrast, it is designed for detail analyses. Therefore, it is possible to request individual bean instances from `EnterpriseBeans` through submitting their identifiers. These might be used as starting points for further investigations. The same holds for the obtainment of individual `Calls` from `EnterpriseBeanInstances`. Additionally, it is possible to select instances and calls based on timespans. In this context, it is possible to request references to all instances of a bean which were engaged in the processing of calls during the submitted timespan. From each bean instance it is additionally possible to request all calls which started within a given timespan. If, for instance, a problem was identified and it can be assigned to a concrete timespan, this timespan can be taken as foundation for selecting instances and calls for further investigations. For complex calculations based on great numbers of calls the API is assumed of not providing the necessary performance because of the different layers between the underlying data source and the representation through the API. These are in particular the database management system, the JPA abstraction layer, the enterprise beans of the container plugin, and the *mKernel API* itself. In contrast, data intensive calculations are recommended for being performed on the underlying database directly. This was not a goal of the design and realization of *mKernel* and is therefore neglected.

Regarding security it is possible to request the name of the principal under which responsibility an invocation was performed. This information might help to identify the concrete client which performed the invocation. Nevertheless, the invocation did not necessarily need to be initiated by the client directly, because principal information is forwarded along call chains if no other configuration is applied, for example, if instances should act in a role different from that of their clients (cf. section 3.2.2).

In the context of transactions, each call allows to request the state of the corresponding transaction after the invocation has finished and all attached interceptor instances have been passed. The particular state is returned as `int`. The value corresponds to one of the constant fields of `javax.transaction.Status` which are used in the context of Java EE to represent transaction status codes.

For performance analyses invocations support autonomic entities through the provision of their start time as `java.util.Calendar` and their duration in nanosecond precision. This timespan is calculated from the arrival of the call at the instance up to the time when all connected interceptors have been passed[23]. This value might, for example, be used to identify performance degradation when execution times of a particular method constantly increase. Furthermore, it might be useful in order to identify sources of potential optimization within call chains, because, for instance, the fraction of the overall processing time of each of the affected

[23] Actually, the value is calculated by an *mKernel*-specific interceptor. Therefore, the timespan ranges from the arrival of the invocation at the interceptor instance to the time when it passed the interceptor after returning form execution at the bean instance.

invocations might be easily calculated.

`ThrownExceptions` grant access to the FQN of the exception class, the corresponding message, and the stack trace of the exception. This information might, for instance, be used for failure analyses performed by autonomic entities.

5.4.3. Information Logging

By default, information on *Instance Level* is not logged by *mKernel* due to performance reasons. Otherwise, each invocation in a managed system would lead to the transfer of the corresponding information to the container plugin and to the creation of the corresponding data within the underlying database. To reach a minimization of performance overhead, *mKernel* supports autonomic entities with opportunities to control when and which information is collected in a fine-grained manner. In this context, information logging can be performed based on scope, type of collected information, and time scheduling. Furthermore, clients of the API can remove collected information from the system to clean up the underlying data base. The remainder of this section discusses the three different aspects of logging, that is, scope, type, and scheduling.

Information collection might be activated for different targets of a managed system. Targets are considered on *Deployment Level*, meaning that architectural elements of a system might be selected for information collection regarding the corresponding elements on *Instance Level*. A target of supervision might either be the system as a whole, a certain `EjbModule`, or a concrete `EnterpriseBean`. The activation of logging for a certain target means that call chains, starting at corresponding elements of the target on *Instance Level*, should be collected. In this context, logging requests are propagated downward call chains, that is, each invocation for which logging is directly activated automatically leads to the logging of those invocations which are performed during its execution transitively.

After completion, logged call chains can be inspected through the API.

Depending on the concrete objectives of analyses, different types of calls might become relevant. For certain situations application calls might be needed as foundation for investigations, for example, to analyze the context of recurring failures. Therefore, the API allows to specify which types of calls should be logged. These types correspond to the direct specializations of `Call`, that is, `LifecycleCall`, `TimeoutCall`, and `ApplicationCall`. During activation of *Instance Level* information collection all types for which corresponding invocations should be logged must be submitted. It is, for instance, possible to specify that for all beans of a certain module all `TimeoutCalls` and `ApplicationCalls` are of interest.

Scheduling of information collection might be controlled based on its start time and the duration of collection. Furthermore, it is optionally possible to define a recurrence interval for renewed collection. Additionally, it would, for instance, be possible to define that logging should start at a certain time, be performed for a certain timespan, and that this should be repeated in a certain interval.

5.4.4. Support for embedded Inspection and Manipulation

The API itself might be used inside and outside of a managed container. Consequently, it might also be used by instances of self-managing enterprise beans which provide a certain business logic on the one hand while being responsible for their own administration on the other hand. The same holds for interceptors which might be applied to realize different AC aspects. In this context, *mKernel* does not impose any restrictions regarding the usage of the API.

To allow autonomic entities to obtain context information during invocation execution, the API provides the class `CallContext`. Its design is oriented at interfaces provided by the EJB standard in order to allow instances to gain contextual information and perform manipulations, for example, regarding the currently processed invocation through `javax.interceptor.InvocationContext`. The class `CallContext` might be instantiated within the execution context of invocations by interceptor instances or a bean instance itself. It provides methods for requesting the identifiers of the current invocation, the affected instance, and the corresponding enterprise bean. This information is sufficient to identify the corresponding *mKernel* representations through the API. In this context, a call identifier might only be used to store additional information, because the corresponding representation could not be requested through the API until the invocation itself has finished[24]. In order to obtain the corresponding identifiers, the methods `getCallIdentifier`, `getInstanceIdentifier`, and `getEnterpriseBeanIdentifier` can be used.

Additionally, `CallContext` enables interceptor instances to inspect and manipulate the state of the corresponding bean instance. For this purpose `CallContext` provides the following methods:

- `getFieldNames`: This method delivers a set containing the names of all fields defined for the bean class and its super-classes.
- `getFieldType`: Through this method, a client can request the FQN of the field type for a submitted field name.
- `getFieldValue`: The value of the field with the submitted name is returned.
- `setFieldValue`: This method can be used to set the value of the field with the submitted name.

If there does exist more than one field for a given name within the inheritance hierarchy of the particular bean, the field of the class nearest to the bottom of the hierarchy is taken. Additionally, the FQNs of the classes within the inheritance hierarchy can be requested through the method `getEjbClassNames`. To provide access to all fields belonging to a hierarchy there does exist a corresponding method with the suffix `ForClass`

[24] This is based on the assumption that logging is activated.

for each of the above methods. Each of these methods expects an additional parameter covering the FQN of the target class. These methods are realized in the same way as their counterparts presented above, but start searching for fields at the class with the submitted name.

Figure 5.11 shows an exemplary inheritance hierarchy.

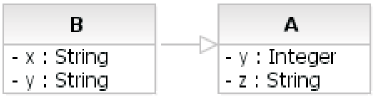


Figure 5.11.: Exemplary Inheritance Hierarchy

Table 5.4 on the page 176 is based on the assumption that class `B` is used as a stateful session bean in a managed system. The upper part of the table contains the values of the different fields before method invocations. The lower part of the table depicts the invocation results of `get`-methods invoked with different parameters. Invocations of the methods with the suffix `ForClass` with the second parameter set to "*B*" would lead to the same result as the invocations of the corresponding methods without the suffix. A presentation of the `set`-methods is left out here, because their effects would be analog to those of the `get`-methods.

Through an instance of `CallContext` interceptor instances might consequently gain full control over the state of the corresponding bean instance. They can, for instance, inspect the state before and after invocation execution to supervise relevant changes. Furthermore, interceptor instances might take snapshots of bean instance states for later analyses or rollbacks, for example, in case of exceptions. In this context, *mKernel* does not perform any kind of cloning regarding exposed states. This lies within the responsibility of the interceptor developers. In this context, snapshots or clones seem only be meaningful for those state elements which are under full control of the bean instance. In contrast, the clone of a reference to a session bean instance or to the EIS-tier might not be useful for later investigations. Summarizing, the class `CallContext` does not support snapshots, but only provides the necessary foundation for their realization.

Field Values		
Class	**Field Name**	**Value**
`A`	`y`	*5*
`A`	`z`	*"A.z"*
`B`	`x`	*"B.x"*
`B`	`y`	*"B.y"*

Return Values of Inspection Methods		
Method Name	**Parameters**	**Return Value**
`getFieldNames`	–	{"x","y","z"}
`getFieldType`	*"x"*	*"java.lang.String"*
`getFieldType`	*"y"*	*"java.lang.String"*
`getFieldType`	*"z"*	*"java.lang.String"*
`getFieldValue`	*"x"*	*"B.x"*
`getFieldValue`	*"y"*	*"B.y"*
`getFieldValue`	*"z"*	*"A.z"*
`getFieldNamesForClass`	*"A"*	{"y","z"}
`getFieldTypeForClass`	*"y","A"*	*"java.lang.Integer"*
`getFieldTypeForClass`	*"z","A"*	*"java.lang.String"*
`getFieldValueForClass`	*"y","A"*	*5*
`getFieldValueForClass`	*"z","A"*	*"A.z"*

Table 5.4.: Inspection Results

5.4.5. Application Example

The observation of *Instance Level* aspects of a running system might help to identify situations which demand for adjustments. Within this section the activation of logging and the subsequent analysis of collected information are discussed. This is done based on the `StandingOrderControllerBean` from the *Managed Layer* `Transfer` module. An example of using the class `CallContext` is not presented here, because its application was already presented abstractly in the previous section.

There might exist manifold reasons for activating logging for a managed system or parts of it such as decreasing performance or user feedback regarding identified failures. Moreover, regular information collection might be performed to constantly observe system behavior. One opportunity might be to activate logging in regular intervals for a limited timespan and subsequently analyze the collected information with respect to occurred exceptions and transactions marked for rollback. Listing 5.8 covers source code which might be used to activate recurring scheduling of logging on system level with respect to application calls and timeout calls.

```
1 long d = 5*60*1000;
2 long i = 60*60*1000;
3 Calendar s = new GregorianCalendar();
4 Set<CallType> t = new HashSet<CallType>();
5 t.add(CallType.APPLICATION_CALL);
6 t.add(CallType.TIMEOUT_CALL);
7 Container c = ContainerFactory.getNewContainer();
8 LoggingTicket ticket =
9   c.createLoggingTicket(d,i,s,t);
```

Listing 5.8: Activation of Logging on System Level

The two long variables initialized in the lines 1 and 2 represent the duration of logging (`d`) and the interval (`i`) between the beginning of two logging rounds in milliseconds. For this case logging should be performed for five minutes and should be started each hour. The Calendar `s` represents that time at which information collection should be started for the first time (line 3). The default constructor of `java.util.GregorianCalendar` creates a representation of the current time. Therefore, it is intended that logging should be started immediately. In the lines 4 to 6 a set of all desired call types (`t`) is created for which logging should be performed. Here, application calls and timeout calls are of special interest. Finally, a connection to the system is established in line 7 which is subsequently used to submit scheduling information in lines 8 to 9.

During logging or in the end of each interval the collected information might be analyzed, for instance, regarding exceptions or invocations for which transaction rollbacks were performed. Instances of the class `Incident` shown in listing 5.9 might be used to store the results of analyses regarding the corresponding call chain and history.

```
public class Incident {
 public Call c = null;
 public Incident superI = null;
 public List<Incident> subIs =
   new LinkedList<Incident>();
 public Incident precI = null;
}
```

Listing 5.9: Class for representing Incidents

In this context, `c` represents that invocation which led to the identification of the incident. The nearest incident within the same call chain of `c` of which the corresponding invocation has led to `c` directly or transitively is intended to be stored in `superI`. On the other hand `subIs` is intended to hold the nearest incidents downward the same call chain, that is, those incidents which occurred during invocations resulting from the execution of `c` directly or indirectly. Finally, `precI` covers the nearest incident which occurred earlier within the call history of the instance upon which `c` was executed. For the three variables `superI`, `subIs`, and `precI`, *nearest* means that the corresponding incident is reachable through navigation along `Call` related associations of `c` without passing other incidents.

In order to identify whether a `Call` relates to an incident and to create the corresponding representation, the static method `analyze` from the following listing 5.10 might be used.

```
1  public static Incident analyze(
2      Call c,Map<Call,Incident> is){
3    Incident i = null;
4    if(is.containsKey(c)){
5      i = is.get(c);
6    }else if(isIncident(c)){
7      i = new Incident();
8      i.c = c;
9      is.put(c,i);
10     i.precI = analyzePrec(c.getPrecedingCall(),is);
11     i.subIs.addAll(analyzeSubs(c.getSubCalls(),is));
12     if(c instanceof ApplicationCall){
13       i.superI = analyzeSuper(
14           ((ApplicationCall)c).getSuperCall(),is);
15     }
16   }
17   return i;
18 }
```

Listing 5.10: Incident Analysis

The method expects a `Call c` which should be analyzed and a `java.-util.Map is` covering mappings from `Calls` to `Incidents` as parameters. As first step, a variable `i` for the return value is created in line 2. Afterwards, it is analyzed whether there does already exist an incident for the current call through requesting the value for `c` from `is`. If there does already exist an incident for `c` it is taken as return value in line 4. Otherwise, it is checked if `c` is an incident through invocation of the method `isIncident` in line 5. This is the case if the transaction status of `c` is either marked for rollback, or `c` has an attached exception. The implementation of `isIncident` is not presented here. If an incident is given the corresponding representation is created and integrated into `is` in the lines 6 to 8. This is necessary to avoid endless loops during the following analysis. The following lines 9 to 14 are needed to identify values for the fields of `i`.

Listing 5.11 presents the method analyzePrec as representative example.

```
1 private static Incident analyzePrec(
2   Call c,Map<Call,Incident> is) {
3  Incident i = null;
4  if(c != null && isIncident(c)){
5   i = analyze(c,is);
6  }else if (c != null){
7   i = analyzePrec(c.getPrecedingCall(),is);
8  }
9  return i;
10 }
```

Listing 5.11: Search for preceding Incident

As parameters, a preceding call and the mapping of calls to known incidents must be submitted. Again, as first step a variable i for the return value is created in line 3. As an invocation of analyzePrec is performed from inside analyze without any validations regarding the existence of a preceding call, this must be performed inside this method (lines 4 and 6). If c exists and an incident is given, again the method analyze is invoked for c in line 5. If c exists but no incident is given, analyzePrec is invoked recursively with the preceding call of c in line 7. This proceeding leads to the identification of an incident upward the call hierarchy of c, if exists. Finally, the resulting nearest preceding call is returned in line 9.

The other methods analyzeSubs and analyzeSuper are not discussed here any further, because they are implemented similar to analyzePrec.

The sequence diagram 5.12 on page 181 represents the call chain of a timeout execution on an instance of the *Managed Layer* SB Standing-OrderControllerBean. For all the corresponding invocations incidents might be identified. Nevertheless, this does not necessarily need to occur on each timeout, but only in special situations. Invocation 1 and 3 are represented through TimeoutCalls by the API while the other ones are BusinessCalls. On timeout occurrence (1/3) the corresponding SB instance (`soc`) initiates the execution of the corresponding transfer on an instance `tc` of `TransferControllerBean` through invocation of the `transfer` method (1.1/3.1). Internally, `tc` delegates the transfer to the instance `txc` of `TxControllerBean` through invoking `transferFunds` (1.1.1/3.1.1). This proceeding indicates a local transfer where no other banking system is affected, because no lookup of a bank access point takes place.

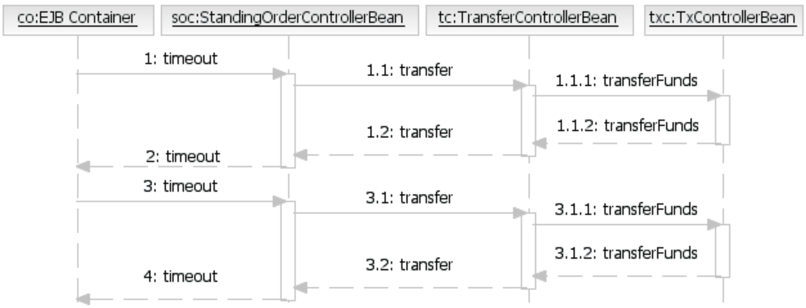


Figure 5.12.: Standing Order Execution

On analyzing the corresponding `Calls`, it can be identified that all transaction status are set to `STATUS_MARKED_ROLLBACK`. Additionally, it can be found out that `txc` threw an `AccountNotFoundException` with an invalid account number as target of the transfer. This information might be a sound foundation for further investigations which would probably show that instances of `StandingOrderController` allow the creation of standing orders with invalid transfer targets, that is, for invalid accounts, because they do not verify the existence of target accounts for accepting order information. In fact, the `StandingOrderControllerBean` on *Managed Layer* does not even require the interfaces `AccountController`. The same holds for the interface `BankController` and the risk that a standing order with an unknown target bank might be created. Nevertheless, this case might not directly be deduced from the above discussed incidents, but might be found out during further investigations. Another possible reason for such an incident might be that the target account of a standing order was removed from the system after the declaration of the order, but before its execution.

The second call chain starting at 3 does not necessarily need to belong to the same overarching incident. Nevertheless, the same behavior and exactly the same account number of the exception might lead to the conclusion that the second timeout invocation was a redelivery attempt of the container. Nevertheless, this can only be assumed based on the incident information.

5.5. Notification Facility

For the application of *mKernel* it is not required that there does exist a centralized management approach. Instead of that, an arbitrary number of autonomic entities might inspect and manipulate a managed system concurrently. Additionally, it is not required that those entities coordinate their actions amongst each other or even publish information about performed manipulations. There might arise situations where one entity performs a certain management action which might be of interest for other entities. If managing entities do not exchange information about their activities, interested entities might request system information in regular intervals and analyze it regarding relevant changes. This might affect the overall system performance more or less negatively depending on the amount of information to analyze, the inspection frequency, and the number of inspecting entities.

To disburden managing entities from the need to constantly poll the system state if they need information about changes and to minimize the performance overhead for information discovery, *mKernel* provides a notification facility. This facility is coarsely oriented at the *Events* facility of the JSR 77 [81] discussed in section 3.4.2. In case of state changes occurring on *Type Level* or *Deployment Level* notifications are published through a well-known JMS topic at which all interested entities might register as listeners. The published messages contain necessary information to identify the affected system elements. Therefore, the information might be used to easily find a starting point for further investigations or to directly perform response actions.

The remainder of this section is structured as follows: Section 5.5.1 discusses the realization of the notification facility with respect to the corresponding meta model elements. Afterwards, section 5.5.2 presents how the facility might be used in the context of keeping externally stored information consistent with the observed system.

5.5.1. Notification Representation

The notification facility addresses state changes of EJB module types on *Type Level*, as well as EJB modules on *Deployment Level*. The corresponding notifications are published through `javax.jms.ObjectMessages` which contain a serializable object as content. In order to transmit notifications, the API uses the three classes presented in figure 5.13 on page 184. The figure does only depict the attributes of the classes, because the classes do not contain any business logic and are intended to be used for information transfer only. For all private variables corresponding `get`-methods are provided. Furthermore, certain JMS properties of the published messages are set to allow the application of message selectors by receivers to only obtain those messages which are relevant for their particular application context. To facilitate the definition of message selectors for receivers most of the static variables of the different classes might be used.

The abstract class `Notification` is used as common super class of *Type Level* and *Deployment Level* related notifications. It is characterized by two static variables. The first variable `TOPIC` holds the name of the JMS topic through which notification messages are published by *mKernel*. It might, for instance, be used in annotations of message-driven beans to

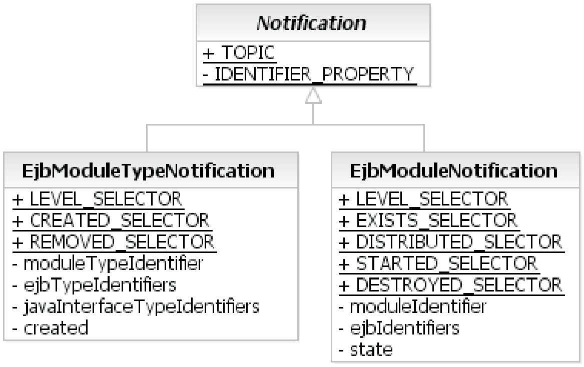


Figure 5.13.: Notification Types

specify their mapped name. The second variable `IDENTIFIER_PROPERTY` holds the property name which is used in notification messages to cover the identifier of the module type or module to which the messages relates depending on the particular level to which the message belongs. It might be used by receivers as part of their message selectors to specify that only messages of a certain module type or a collection of module types are of interest. The same holds for *Deployment Level* related messages and EJB modules, respectively.

The two specializations of `Notification` are specific for a concrete level. `EjbModuleTypeNotifications` are used for *Type Level* related notifications while `EjbModuleNotifications` are used for the *Deployment Level*. Both classes contain static variables which might directly be integrated into message selectors inside annotations. The particular `LEVEL_-SELECTOR` specifies that only messages of the corresponding level are of interest. The other static variables define that only messages relating to a specific state are relevant. For the *Type Level* these relate to the creation (`CREATED_SELECTOR`) and removal (`REMOVED_SELECTOR`) of module types. `EjbModuleNotification` provides selector elements through static variables which might be used to define that only notification about the arrival of modules in a certain state are of interest. The different selector elements might be used in combination to construct more complex message selectors.

Each `EjbModuleTypeNotifiation` contains the unique identifier of the affected `EjbModuleType` as value of the variable `moduleTypeIdentifier`. Additionally, the variable `ejbTypeIdentifier` covers a set of all identifiers of the affected EJB types. Especially for the case of module type removal this might be of special interest, because these identifiers and the corresponding EJB types cannot be identified through the API anymore. The same holds for all affected Java interface types of which the corresponding identifiers are kept in the variable `javaInterfaceTypeIdentifiers`. In this set only identifiers of Java interface types are contained which are newly integrated during module type creation or deleted from the system during module removal. Finally, a *Type Level* notification covers a `boolean` variable `created` which is `true` in case of a module type creation and `false` when a removal occurred.

Module related notifications cover the unique identifier of the affected module (`moduleIdentifier`), identifiers for the corresponding enterprise beans (`ejbIdentifiers`), and the resulting deployment state (`state`). Especially if a module has reached the `DESTROYED` state the bean identifiers might be of special interest because of the same reason as for *Type Level* notifications.

5.5.2. Application Example

As discussed in the sections 5.2 to 5.4, the API provides a rich set of opportunities for inspection and manipulation of a managed system. These opportunities were designed comprehensive with respect to the EJB standard, that is, they are intended to focus on aspects of the standard while not addressing concrete application areas of AC. For different application contexts it might be necessary to store additional information regarding certain elements of a managed system. These might reach from rather simple comments and remarks regarding, for instance, a single enterprise bean or module on *Deployment Level* up to complex data structures which relate to multiple elements of a system on different levels. Storing and keeping these kinds of information is not directly supported by the meta model. Nevertheless, the notification facility might be used to update externally managed information for keeping them consistent with the managed system.

As mentioned in section 5.4, *Instance Level* information might be used to calculate statistical information regarding system performance. As one rather simple example, it would be possible to record information about the number of processed invocations, the average execution time, or the minimal and maximal invocation execution times for each enterprise bean in a managed system. These measures might be calculated based on samples from logged *Instance Level* information. After calculation the information might be stored in a data base for later analyses, for example, regarding usage peeks or performance evolution over time. To allow the later identification of a corresponding `EnterpriseBean` through the API, its unique identifier might be stored as part of a statistics record during collection.

If a module is removed from the managed system, the corresponding statistics should also be removed to keep information consistency. For this purpose the message-driven bean depicted in listing 5.12 on page 187 might be used.

The annotations in the lines 1 to 9 are used for configuration purposes. As mentioned in the previous section, the static variable `TOPIC` of `Notification` can be used to register the MDB at the notification topic through its mapped name. The message selector defined in the lines 4 to 8 is based on a combination of the static variables `LEVEL_SELECTOR` and `DESTROYED_SELECTOR` of `EjbModuleNotification`. It has the effect that only *Deployment Level* notifications relating to the destruction of a module are received by instances of the MDB.

```
1  @MessageDriven(
2   mappedName = Notification.TOPIC,
3   activationConfig = {
4    @ActivationConfigProperty(
5     propertyName = "messageSelector",
6     propertyValue =
7        EjbModuleNotification.LEVEL_SELECTOR+
8        " AND "+
9        EjbModuleNotification.DESTROYED_SELECTOR)
10  }
11 )
12 public class SystemObserver implements
13    MessageListener{
14
15  @PersistenceContext
16  private EntityManager em;
17
18  public void onMessage(Message m){
19    ObjectMessage o =(ObjectMessage)m;
20    EjbModuleNotification n = null;
21    try{
22     n = (EjbModuleNotification)o.getObject();
23    }catch(JMSException e) {
24     // Exception handling
25    }
26    for(String ejbId:n.getEjbIdentifiers()){
27     Query q = this.em.
28        createNamedQuery("deleteUsageStatisticsForEjb");
29     q.setParameter("ejbId", ejbId);
30     q.executeUpdate();
31    }
32  }
33
34 }
```

Listing 5.12: Application of Notifications

The entity manager `em` is used for interactions with the EIS-tier and to remove statistic information (lines 13 and 27 to 30). On arrival of a message the included notification object is extracted for further processing (lines 19 to 25). The handling of potential exceptions is not relevant for the example presented here (lines 23 to 25). Afterwards, the unique identifiers of the affected beans are used to remove the corresponding statistics information from the underlying data source. In this context, the `NamedQuerydeleteUsageStatisticsForEjb` is defined for the entity of which instances are used to store statistical information for a single bean in a single observation interval. The parameter `ejbId` is used to identify which entries should be removed during update execution.

The original representation of statistics information is not relevant in the context of this thesis. It is therefore omitted.

5.6. Support for seamless Reconfiguration

In section 1.2.2 it was highlighted that dynamic composition is a desirable capability of a managed system. Nevertheless, the EJB standard does not address this aspect, as discussed in section 3.1.2.2. In section 5.3.2 the fundamental primitives for dynamic composition with respect to rerouting of connections of a managed system were presented. These are limited to situations where no state transfer between bean instances is necessary and no atomic rerouting of more than one connection is required. The same holds for the parameter adaptation with respect to simple environment entries. These might be changed through the API individually, but not in combination atomically. Furthermore, the realization of the API does not guarantee that new values affect existing instances.

Within this section the foundation for more complex scenarios is presented which does not adhere to the above stated restrictions. In contrast, dynamic compositional adaptation is supported for an arbitrary set of managed components. This also includes the integration of new modules and the removal of those which are not needed anymore. Additionally, atomic adaptation of multiple parameters can be realized through API elements.

Seamless reconfiguration, as supported by *mKernel*, is based on the concept of so-called *Quiescence* (cf. [99]). This concept was not newly developed for the *mKernel* system, but is well established in literature. The fundamental idea and the corresponding concepts are discussed in section 5.6.1. Afterwards, section 5.6.2 presents how these concepts are supported by *mKernel* through meta model elements. The proceeding of state transfer from modules to be replaced to replacing ones is explained in section 5.6.3 followed by a discussion of the implicit limitations of the approach realized by *mKernel* in section 5.6.4. Finally, section 5.6.5 contains an example for seamless reconfiguration based on the case study.

5.6.1. Background

The basic goal of dynamic composition is to reconfigure the architecture of a system while it is running. Beyond the core goal of reaching the desired system architecture dynamic composition has two major objectives, namely to keep consistency of the system state and to minimize system disruption [71,99,117]. In this context, *State* does not relate to the architecture of the system regarding its constituent components and connections among them, but to the underlying data source and potentially ongoing interactions.

The preservation of consistency can be seen as mandatory objective for reconfiguration execution. Otherwise, a system might show unintended or erroneous behavior, and the underlying data structures might become corrupted. For enterprise systems this is highly critical because of the potential legal consequences and the potential loss of trust and reputation due to poor usage experiences of its users.

The minimization of system disruption can be considered on different levels. First of all, there might occur situations which lead to disruption of availability such as connection losses or transaction aborts due to reconfiguration execution. A weaker form of disruption might occur when perceived performance degrades due to reconfiguration. This might, for instance, be the case if processing of interactions is blocked during certain phases of reconfiguration and continued after their completion. Consequently, algorithms for performing dynamic adaptation should first of all avoid situations which harm availability, that is, they should work seamlessly. Secondly, they should minimize delays noticed by system clients.

A general and transparent approach for seamless reconfiguration is necessary for *mKernel*, because it aims to provide a generic infrastructure for AC. Furthermore, special restrictions or guidelines for component development should be avoided (cf. objective *SoftR-MT* in section 1.3). The approach realized as part of *mKernel* is based on the fundamental ideas of *Kramer and Magee* [99]. They assume that at runtime a system or parts of it pass through consistent states which are safe for reconfiguration. During interactions consistency might be violated, but is restored after finishing them. Consequently, consistent reconfigurations can be performed at least in situations where no interactions are active on the affected parts of a system. Such a situation is called *Quiescent State*. Regarding compositional adaptation, *Kramer and Magee* explicitly consider the addition and removal of components, and the manipulation of connections. In particular, *Kramer and Magee* state four properties a quiescent element has to fulfill:

1. *it is not currently engaged in a transaction that it initiated,*
2. *it will not initiate new transactions,*
3. *it is not currently engaged in servicing a transaction, and*
4. *no transactions have been or will be initiated by other nodes which require service from this node.* (cf. [99], p. 1296)

The term *Transaction* relates to a single interaction and should not be confused with transactions as considered in the EJB standard. The first two properties are summarized under the term *Passive Properties* in the work of *Kramer and Magee*. They demand that a quiescent node is neither currently executing a self-initiated interaction nor will initiate new ones during reconfiguration. Consequently, its behavior is characterized by passiveness. The other two properties demand that quiescent nodes are not currently executing any interactions initiated by other nodes, and that it is ensured that no new interactions are started upon a quiescent node during reconfiguration. Summarizing, a quiescent node is not affected by any interactions during reconfiguration execution. In this context, interactions relate to the core application logic of the node. In contrast, the execution of administrative interactions such as state extraction and transfer would be valid.

In order to reach the properties 1 and 3, active transactions on elements for which quiescence should be reached must be finished. This might demand that *consequent* transactions are also finished. In this context, a consequent transaction is a transaction which is started during the execution of another transaction. In order to finish the original transaction, all consequent transactions must be finished. Consequently, it might be necessary that new transactions must be allowed on an element for which quiescence should be reached to finish other already active transactions. The properties 2 and 4 demand that new, non-consequent transactions are neither started by those elements for which quiescence is desired nor by elements which possess outgoing connections to the former elements transitively. Therefore, *Kramer and Magee* demand in their original approach that all elements which are connected to those elements which should be subject to reconfiguration must also be transferred to a quiescent state. The design of the original approach envisions a programming interface for a quiescence manager to instruct elements to behave passively. This would be used for all affected elements to instruct them not to violate property 2. In combination, this would lead to a situation where no new transactions are started on the affected part of a system, because no incoming connections through which new transactions might reach the affected system part would exist. This design comprises the important implication that the complete system is under the control of a quiescence manager, that is, it is possible to transfer all relevant parts of a system into a passive state to avoid the occurrence of new transactions completely also including its clients. Furthermore, the need to transfer more elements into a passive state than only those which should be subject to reconfiguration might induce situations where transactions are avoided even if they would have been possible. This might, for example, be the case if such a transaction is executed without the need for performing any consequent transaction upon an element which is subject to quiescence. Consequently, system disruption is not necessarily minimized.

As alternatives to the original approach which is characterized as *static* in literature (cf. [79]) there do exist other approaches which take additional information about ongoing transactions into account to minimize disruption (cf., e.g., [13, 44, 161]). These approaches are based on the idea to identify whether a transaction was initiated by an element for which quiescence is desired or whether it is a consequent transaction which must be finished. All other transactions are blocked until the reconfiguration has finished. Consequently, these approaches do not require that no new transactions are started by elements outside the affected part of a system. In contrast, they rely on the opportunity to analyze call chains to identify those transactions which must be forwarded and those which must be blocked before they reach their original target. As they reduce the set of necessary nodes to transfer to quiescence and do only affect those transactions which would actually affect quiescent nodes, they perform at least as well as the original, static approach with respect to the minimizing system disruption. These approaches are called *dynamic* in literature (cf. [79]).

Beyond consequent transactions there might also occur situations where transactions depend on other transactions which do not belong to the same call chain. This might, for example, be the case if a node is processing an interaction during which it waits for a notification from a distinct transaction. These scenarios are considered by *Warren and Sommerville* [160] under the term *Constrained Requests*. They are not considered by the above mentioned reconfiguration approaches. Moreover, the existence of such constellations in a concrete reconfiguration situation can not directly be deduced solely based on call chain information and connections in a managed system. In contrast, they can be assumed of being application specific.

During execution of a reconfiguration state transfers from replaced to replacing elements might become necessary. Most of the proposed approaches do not try to automate the original transfer because of the wide range of potential syntactic and semantic differences between the state representations of two or more elements between which states should be transferred. Although there do exist approaches which try to partially automate state transfers (cf. [155]), a complete automation for all conceivable situations without any additional specifications does not seem to be feasible at the time this thesis is written.

5.6.2. The Quiescence Region

The API of *mKernel* provides the opportunity to define a set of system elements which should be brought to a quiescent state through a `Container` reference. This set is called *Quiescence Region* and can consist of single enterprise beans and complete EJB modules, as well as a combination of both. It is represented through the interface `QuiescenceRegion`.

At any given time there might exist at most one quiescence region inside a managed system. This is not a restriction regarding the provided functionality for seamless reconfiguration, because a quiescence region might contain an arbitrary set of EJB modules and beans independent from their corresponding deployment state. Consequently, elements in use in the beginning of reconfiguration, as well as newly created ones in an arbitrary state might be integrated into a region. On the other hand, a support for the existence of multiple regions in parallel would cause additional overhead regarding API usage such as the need to identify the relevant region or additional configuration demands for defining relationships and priorities among regions. Summarizing, the decision to only allow at most one region at any given time was made to keep the API simple to use.

During its life cycle a `QuiescenceRegion` might pass through different states which are depicted in figure 5.14.

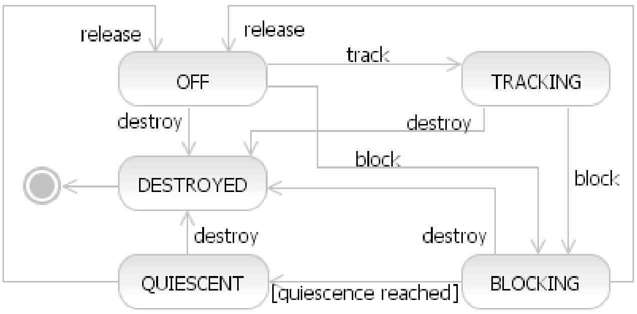


Figure 5.14.: Quiescence Region States

The life cycle of a quiescent region starts in the state `OFF`. This state indicates that the region does exist within the system, but does not have any effects on the managed system. From the `OFF` state the `TRACKING` and `BLOCKING` states can be reached through successful execution of the corresponding operations `track` and `block` respectively.

An *mKernel*-based system does not hold references to active SB instances due to memory reasons. Nevertheless, such references are needed if state transfers of stateful SB instances are desired during reconfiguration, and an exchange of references to replaced instances with references to replacing ones should be performed. During the `TRACKING` state references to instances are tracked in case they are used. Therefore, the `TRACKING` state can be seen as optional state for collecting active references. During this state no interactions are blocked. Consequently, tracking is recommended to be activated a certain time before the actual reconfiguration should take place to allow the system to determine all relevant references. From the

TRACKING state a quiescence region can be transferred to the BLOCKING state through an invocation of the `block` method.

During the BLOCKING state all invocations and incoming messages directed to an element of the quiescence region are analyzed whether their call chains have already passed another element of the region. If this is the case, they are forwarded to their original targets. Otherwise, their execution is blocked at their source. This also holds for the creation of new references to bean instances belonging to the region. Consequently, there are no invocations or messages blocked inside elements which are the targets of reconfiguration. On entering the BLOCKING state, timers of stateless SBs and MDBs are suspended. Furthermore, newly created timers are not actually activated, but added to the set of suspended timers. Summarizing, no new interactions can enter the quiescence region during the BLOCKING state, and no interactions can be started from inside the region. Therefore, the properties 2 and 4 for quiescent elements, as discussed in the previous section, are enforced during the BLOCKING state regarding invocations related to the business logic of the affected elements. The collection of references as in the TRACKING state is continued during BLOCKING. Through the `release` operation the region might be transferred back to the OFF state. This implies the release of all blocked invocations, the restart of all suspended timers and a reset of tracked reference information.

If there are no interactions active within the quiescent region anymore, it is transferred to the QUIESCENT state automatically. The system behavior is the same as for the BLOCKING state, but the properties 1 and 3 of quiescent elements discussed in section 5.6.1 are also given. This also covers life cycle calls on enterprise beans instances. Internally, *mKernel* simulates passivation of stateful session bean instances. This would lead to a state of all known instances which would allow their serialization. Furthermore, underlying data sources would be left in a consistent state, because this is one of the requirements a bean provider has to fulfill as preparation for passivation. If instances perform invocations on other instances during simulated passivation, this is also covered by *mKernel*. For the case that the passivation of the target instance was already simulated its activation is simulated to service the new request. Afterwards, its passivation is simulated again. This proceeding might lead to an endless loop if two or more instances require their mutual activation. Although this is theoretically possible, it can be seen as design error, because this situation might also occur without the application of *mKernel*. After finishing passivation life cycle invocations performed by the container are avoided for all bean instances during the `QUIESCENT` state, that is, they are hindered from reaching their original target. Therefore, no new interactions could be started from inside the quiescence region. This proceeding might lead to the omission of *PreDestroy* invocations on stateless SB instances and instances of MDBs. Nevertheless, this should not lead to any consistency problems, because the EJB standard does explicitly highlight that bean implementations must not rely on the container always invoking the corresponding method (cf. [58], p. 81 and p. 114). Summarizing, the `QUIESCENT` state ensures that the required properties of quiescent elements, as discussed in section 5.6.1, are fulfilled. Therefore, seamless reconfiguration can be performed in this state. After finishing reconfiguration, the state of a region can be transferred back to the `OFF` state through an invocation of `release`. The results of execution are the same as for the `BLOCKING` state. Additionally, life cycle invocations performed by the container are again forwarded to their targets. This might imply a synchronization of the instance life cycle with the simulated life cycle. If, for example, a *PreDestroy* invocation arrives at an instance for which passivation was simulated, an activation is simulated before the destruction call is forwarded.

Finally, the `DESTROYED` state can be directly reached from each other state through execution of the `destroy` operation. Depending on the state from which the operation is started, all corresponding effects of the quiescence region are canceled. Furthermore, no actions regarding the region are possible if it reached this state. This state represents the final state of a region. It is considered explicitly in the life cycle of a region, because managing entities might hold references to representations of the region after the execution of destroy and might try to use this reference. In order to provide them with correct state information, the `DESTROYED` state was integrated.

5.6.3. State Transfer

The API supports state transfer for stateful SB instances, as well as the transfer and creation of timers for stateless SBs and MDBs. Furthermore, all types of inspection and manipulation, as discussed in the previous sections, are also available during reconfiguration. States of underlying data sources are not considered by *mKernel*. In case manipulations or transfers are necessary, these must be treated outside of *mKernel*, for example, through direct interactions with the corresponding database management system.

References to stateful SB instances References to instances of stateful SBs are represented through the interface `HoldingReference`. There are two opportunities provided by the API to obtain such a reference. First of all, a reference to an existing quiescence region provides access to all blocked references. These might be used to access the corresponding instances for state extraction. Secondly, new `HoldingReferences` might be created for stateful `SessionBeans` which belong to modules in state `STARTED` through invocation of the `createReferenceTo` on a `Container` reference, submitting the corresponding bean representation. These references are intended to be used for state injection.

The interface `HoldingReference` allows to request values of all fields of the corresponding SB instance. All of these fields must either be `null` or contain values which are serializable for references obtained from a quiescent region, because the EJB standard stated this as requirement for stateful SBs in state *passive*[25]. Elements of an instance state are represented through the interface `StateElement`. This interface allows to access the type and name of a field, as well as its value. The type of a field might either be obtained as class or as `java.lang.String` representing the corresponding FQN. Values of fields might be requested as instances of the corresponding classes or as `byte` arrays. These alternatives were chosen to allow the transfer of states even through managing entities which do not have the classes for all fields in their classpath. Analog to the discussion in section 5.4.2, the state of a reference is divided according to the inheritance hierarchy of the corresponding session bean. Therefore, the state is represented through a map containing the fully qualified class names as keys, and a set of the corresponding state elements as values. The injection of the state for a new reference might be performed through corresponding methods provided by `HoldingReference`. Therefore, the FQNs of the target classes, the names of the affected fields, and the corresponding values to be set must be submitted. Finally, the replacement of the original reference must be published within the system. This can be performed through invocation of the method `replaceWith` upon the original `HoldingReference`. The new reference must be submitted as parameter to this method. During transfer of instance states entities are also supported. If they are attached to a persistence context within the original instance, this attachment is reestablished inside the new reference. For the default case the attachment is created to a persistence context with the same name in the target instance. Alternatively, it is possible to define a mapping from persistence context names of the source instance to names of the target instance. This does not necessarily cover all con-

[25] The requirements discussed in the EJB standard are also supported by *mKernel* (cf. [58], p. 64).

ceivable cases. The corresponding design and realization are intended to provide a first proposal of how persistence aspects might be treated. Nevertheless, addressing persistence aspects was not a goal for the development of *mKernel*.

Timers Timers are represented through the class `EnterpriseBeanTimer`. It provides access to the next activation time of the timer, as well as the interval between two activations if a recurring timer is given. Furthermore, the encapsulated context data might be requested. As for instance state elements context data might either be accessed through a `byte` array or as object. All state elements of a timer might be inspected and manipulated. To obtain representations of all timers of a stateless SB or MDB, the method `getTimers` on the corresponding `EnterpriseBean` might be invoked. A `setTimers` method is also provided by the interface `EnterpriseBean` for setting timers. Furthermore, managing entities might construct new instances of `EnterpriseBeanTimer`. In combination, entities are enabled to inspect existing timers and create a new set of timers for stateless SBs or MDBs which might include existing timers, as well as new ones. Furthermore, they might remove all timers or parts of them from beans.

5.6.4. Limitations

The design of quiescence, as provided by *mKernel*, does imply certain limitations which are discussed within this section.

Consequent Interaction According to the discussion in section 5.6.1, consequent interactions must be permitted to pass a `BLOCKING` quiescence region to reach the `QUIESCENT` state. This is guaranteed for all consequent interactions inside a managed system. Nevertheless, there are situations conceivable where entities outside a managed system are accessed by a bean instance of a managed system. If such an external entity would itself access at least one element of the quiescence region directly or transitively during processing a request from inside the region, a deadlock might occur, because the external request would not be identified as consequent invocation. This is the case, because on performing external interactions all call chain information is lost. Hence, the foundation of the identification of consequent interactions would not be available during deciding whether an invocation should be blocked or not. Summarizing, the correct identification of consequent interactions is limited to those interactions which occur inside a managed system.

Transactions The EJB standard does not require that any information is provided to identify transactions in an EJB-based system such as unique transaction identifiers. Therefore, it is not possible to determine whether an externally initiated interaction belongs to a transaction which is already active in a quiescence region. Those interactions would get blocked even if they should be processed to finish an active transaction. Another scenario would be if a stateful SB instance with BMTD started a transaction during the processing of an invocation. This transaction should be subsequently committed during a second invocation. If this invocation is blocked, the transaction could not be finished. This scenario is similar to the first one. Nevertheless, it could not even be handled if transaction identifiers would be available, because for BMTD client transactions are not joined, but new ones are started. Hence, it would be theoretically possible that two independent client transactions lead to the starting and committing of a bean instance transaction. Therefore, an identifier for the client transaction would not help. These scenarios would not lead to deadlocks, but might result in transaction rollbacks in case of the removal of bean instances. Consequently, reconfiguration could not be performed seamlessly in case external clients are affected by transaction rollbacks.

If transactions span multiple interactions and these interactions are performed during the processing of a single interaction from inside another bean instance directly or transitively, the occurrence of the above scenarios might be avoided. For such a case the relevant beans must be added to the quiescent region even if they are not subject of reconfiguration with respect to their adjustment, replacement, or removal.

Constrained Interactions The EJB standard allows situations where constrained interactions might occur. An example of such a situation might be the direct use of a message queue as message receiver from inside the execution of an interaction. If the expected message would be sent during the execution of another interaction, it might be possible that this interaction is blocked which would result in a deadlock. Scenarios of constrained interactions can be – according to *Warren and Sommerville* [160] – assumed to be application specific and could not be identified without any further information in general. This aspect was left open for *mKernel*.

Data Inconsistency A reconfiguration might only be performed seamlessly if data exchanged between system elements and their external clients do not become inconsistent. This might, for example, be the case if certain identifiers are exchanged between SB instances and external clients. If these identifiers are converted during state transfer between instances or the transfer of underlying data sources, this conversion cannot be performed for the data held by external clients, because *mKernel* was not designed to control their state.

5.6.5. Application Example

For the example presented in this section it is assumed that the operating bank wants to charge a fee for bank transfers performed by their customers. Therefore, two enterprise bean implementations are adjusted, namely the `TxControllerBean` from the *Foundation* component and the `TransferControllerBean` from the *Transfer* component. For both of them a new simple environment entry type `fee` is integrated which covers the amount in cents which should be charged as fee for each transaction. All other fields of the corresponding classes remain unchanged. The same holds for the fully qualified class names. The implementations of both bean types are adjusted to charge a fee for each transaction if `fee` is set to a value greater zero and to track these as extra transaction in the particular account record[26]. All other bean implementations are adopted unchanged. In general, it is assumed that none of the configuration aspects are changed between the original and the replacing module types. Regarding the externally observable properties of the affected module types all required and provided interfaces of the constituent bean types are not affected by the adjustments stated above. This also implies that none of the Java interfaces were changed. Furthermore, no changes of the underlying database and its representation were necessary. Consequently, only the application logic of two enterprise bean types is changed and two new fields are added.

To integrate the adjustments into a managed system, the two modules *Foundation* and *Transfer* of the *Managed Layer* must be replaced with adjusted releases. Due to the absence of Java interface changes there are no changes required regarding the *Access Layer* modules. Furthermore, the underlying data source can be directly adopted.

The following discussion is kept generic, that is, it is not limited to the concrete situation presented above, but might be applied to similar scenarios. For those the requirements regarding the absence of interface and state changes must also be fulfilled. As preparation for the reconfiguration it is assumed that the archives of the replacing *Managed Layer* modules are integrated into the system according to the discussion in section 5.2. It is

[26] In this context, the value of `fee` set for `TxControllerBean` is used for transactions executed if the source and target account of a transaction are hosted by the considered bank while the value of `fee` for `TransferControllerBean` is used for outgoing transactions with a target account hosted by a different bank.

also assumed that – in accordance with the presentation in section 5.3 – `EjbModules` are created from them, environment entries are set, and the modules are connected internally, as well as to their environment, that is, required interfaces of the replacing modules are connected to matching provided interfaces of the system. In this context, no connections to modules to replace are created, and incoming connections are only established from beans belonging to replacing modules. Finally, it is assumed that the replacing modules are transferred to the state `DISTRIBUTED`, but not yet started. Consequently, all constituent beans of the replacing modules are in state `ACTIVATABLE` regarding their required references while no bean outside the replacing modules does rely on their `EjbInterfaces`. The source code, building the foundation of this section, can be found in appendix C.

For the following discussion of the reconfiguration proceeding five data structures are used as foundation:

- `om`: This `java.util.Set` holds representations of the original modules which should be replaced.
- `rm`: In this `java.util.Set` representations of the replacing modules are covered.
- `os`: The elements of this `java.util.Set` consist of SB representations belonging to the original modules.
- `rs`: To easily identify the session beans of the replacing modules, this `java.util.Map` contains the SB names as keys and their representations as values.
- `ri`: Analog to `rs`, this `java.util.Map` allows the identification of the `EjbInterfaces` of the replacing SBs through submitting the corresponding `JavaInterfaceType`. This unambiguous mapping is possible, because each affected `JavaInterfaceType` is provided by exactly one `EjbInterface` within the two modules.

The set `os` and the two maps `rs` and `ri` are used to facilitate the identification of their values. They do not cover any information which could not be obtained through navigation along the associations of elements of om and rm. As preparation for the reconfiguration the data structures are constructed and filled with the corresponding entries. In addition to the data structures, a connection to the managed system is established through a Container reference c. Finally, a reference to the QuiescenceRegion q is used for inspection and manipulation purposes.

After finishing preparations, the original reconfiguration starts with the definition of a quiescence region and its transfer to the state TRACKING. This can be reached through execution of the following method define-Region depicted in listing 5.13.

```
1 public void defineRegion(){
2  Set<EjbModule> qm = new HashSet<EjbModule>();
3  qm.addAll(this.om);
4  qm.addAll(this.rm);
5  this.q = c.declareQuiescenceRegion(qm, null);
6  this.q.track();
7 }
```

Listing 5.13: Definition of a QuiescenceRegion and Transfer to the TRACKING state

In the lines 2 to 4 the set of modules which should be considered by q is created. It consists of all modules affected by the reconfiguration, that is, the modules to replace and the replacing ones. The integration of the set of replacing modules into the reconfiguration set is not necessary for the case study. Nevertheless, it has the advantage that no life cycle calls would be forwarded to bean instances of those modules during reconfiguration. Therefore, no connections would be established which leads to performance savings. In general, the execution of invocations should be avoided on replacing modules before reconfiguration is finished. Otherwise, there might arise situations where these might cause inconsistencies either through interactions with an underlying database which was not adjusted or transferred, or through interactions with other bean instances which directly or transitively affect the quiescence region. This might even lead to deadlocks[27]. Such a case cannot occur in the current scenario. The first parameter during the declaration of a quiescence region in line 5 requires a `java.util.Set` of `EjbModules` which should be transferred to quiescence as a whole during a reconfiguration. As second parameter a `java.util.Set` of `EnterpriseBeans` is expected which should be brought to quiescence. For the example presented here no individual beans are considered which should be treated independently from their modules. Therefore, `null` is submitted for the second parameter. The activation of tracking performed in line 6 should be performed an adequate timespan before the following steps, as argued before, to collect information about references which do exist within a system. A delay which conforms with the timeout period of connections as configured within a container might be an appropriate choice, because if references are not used within this timespan, clients might also be confronted with connection losses even if no reconfiguration is performed. This delay is assumed of being given before the following steps are executed for this example.

After the tracking period quiescence must be reached for the region to perform the original reconfiguration. This transition can be initiated through execution of the method in listing 5.14.

```java
public boolean reachQuiescence(){
 this.q.block();
 return this.q.waitForQuiescence();
}
```

Listing 5.14: Reaching Quiescence

[27] An example of such a situation would be if an instance of a replacing bean performs an interaction with an arbitrary instance of another bean during construction. If this instance itself is connected to an instance belonging to a quiescence region in state `BLOCKING` or `QUIESCENT` and tries to access this instance, this invocation would get blocked.

As first step `q` must be transferred to the `BLOCKING` state. Afterwards, in line 3 the method `waitForQuiescence` is used to wait for the region to be internally transferred to the `QUIESCENCE` state. The method returns a `boolean` value indicating whether quiescence was reached (`true`) or another state transition occurred which prevents the reaching of quiescence (`false`). This might be the case if another autonomic entity destroys the region before quiescence is reached. For the remainder of this section it is assumed that quiescence has been reached.

In order to perform the seamless reconfiguration, three steps must be executed before deactivating the replaced modules and releasing the region. These are the transfer of state for stateful SB instances, the transfer of timers, and the manipulation of the architecture. These might be performed in arbitrary order. For this example the execution is presented in the aforementioned order.

Through the execution of listing 5.15 the state of original stateful SB instances is transferred to replacing counterparts. Additionally, references to the original instance within the system are replaced with references to the replacing ones. In this context, *mKernel* blocks invocations on these new references until the region is released.

```
public void transferState(){
 for(HoldingReference ob:this.q.getReferences()){
  SessionBean s = this.rs.get(
      ob.getSessionBean().getType().getEjbName());
  HoldingReference rb = this.c.createReferenceTo(s);
  rb.setState(ob.getState());
  if(s.getSessionBeanType().getEjbName().
    equals("TxControllerBean")){
   rb.setFieldValue("fee", new Long(10));
  }
  ob.replaceWith(rb);
 }
}
```

Listing 5.15: State Transfer between stateful SB Instances

The method iterates over all holding references `ob` of the quiescence region (lines 2 to 12). In a first step the session bean `s` of which a replacing instance should be created is selected from the map of replacing session beans in lines 3 and 4. Afterwards, in line 5 an instance of `s` is constructed through invocation of the method `createReferenceTo` on the `Container` reference which returns a reference `rb` to the representation of the new instance. Afterwards, the original state transfer is performed in line 6 through invoking `setState` on `rb` with the result returned from the invocation of `getState` on `ob`. This is only possible, because all fields of the SB type of `ob` are also present in the SB type of `rb` and the fully qualified class names are the same. While the previous lines are generic and could be applied to other scenarios, the following four lines 7 to 10 are specific for the case study. To avoid the execution of method invocations in a `QUIESCENT` region, life cycle calls are blocked by *mKernel*. Furthermore, dependency injection is not performed to avoid the need to construct new instances of referenced session beans. Therefore, the new field `fee` must be set explicitly on each instance of the replacing `TxControllerBean`. Within this listing the transfer fee is set to ten cent. Finally, in line 11 the replacement of all references to the SB instance represented by `ob` with the instance represented by `rb` is performed through the invocation of the method `replaceWith` upon the original reference, submitting the new reference which should be used instead of the original one after finishing the reconfiguration.

If no changes regarding the application logic for timers is given between the original beans and the replacing ones, timers can be directly transferred without any needs for adjustments. In fact, for the scenario presented here the affected bean (`StandingOrderControllerBean`) was not even changed. The method `transferTimer` in listing 5.16 on page 208 performs this direct timer transfer on execution. The method iterates over all original session beans `os` (lines 2 to 8). If a stateless session bean `o` is given (lines 3 to 7), the name of the corresponding SB type is identified in line 5. This name is used to find the replacing SB in the lines 4 and 5. Finally, the timers extracted from `o` are set on the replacing bean `r` in line 6.

```
1 public void transferTimers(){
2  for(SessionBean o:this.os){
3   if(!o.getSessionBeanType().isStateful()){
4    SessionBean r = this.rs.get(
5      o.getSessionBeanType().getEjbName());
6    r.setTimers(o.getTimers());
7   }
8  }
9 }
```

Listing 5.16: Transfer of Timers

Internally, *mKernel* uses an instance of `r` to set the timers. For this instance no dependency injection is simulated and no *PostConstruct* method is invoked to prevent undesired side effects. Both of them are caught up if necessary on arrival of the first client interaction after the end of reconfiguration. The same would hold for MDB instances which are not considered in this example.

The manipulation of the system architecture might be performed analog to the discussion in section 5.3.2. In this context, all incoming connections of the elements of `os` must be reconnected to matching values of `ri`, as depicted in listing 5.17 on page 209.

The method iterates over all original session beans `s` (lines 2 to 11). For each of them the provided `EjbInterfaces` `i` are handled in a nested iteration in the lines 3 to 10. From each `i` the connected `EjbReferences` `r` are addressed for reconnection in a third iteration in the lines 4 to 9. Only if the corresponding `EnterpriseBean` is not part of the set of original SBs, it must be reconnected (lines 4 to 8). Although it would not cause any problems to also reconnect the `EjbReferences` of the original session beans, this is not necessary, because the corresponding modules are deactivated and removed from the system before the reconfiguration is finished. The original reconnection is performed in the lines 5 to 7 through identifying the provider of the same `JavaInterfaceType` as provided by `i` inside `ri`. To this provider `r` is connected through invocation of the method `connectTo`, as presented in section 5.3.2.1.

```
1  public void replaceConnections(){
2   for(SessionBean s:this.os){
3    for(EjbInterface i:s.getEjbInterfaces()){
4     for(EjbReference r:i.getConnectedEjbReferences()){
5      if(!this.os.contains(r.getEnterpriseBean())){
6       r.connectTo(this.ri.get(i.getEjbInterfaceType().
7          getJavaInterfaceType()));
8      }
9     }
10   }
11  }
12 }
```

Listing 5.17: Reconfiguration of System Architecture

To finish the reconfiguration, the original modules must be stopped, and the region must be released. Afterwards, the elements of `om` can be undeployed and destroyed. Listing 5.18 covers the corresponding source code.

```
1  public void finish(){
2   for(EjbModule m:this.om){
3    m.stop();
4   }
5   this.q.release();
6   for(EjbModule m:this.om){
7    m.undeploy();
8    m.destroy();
9   }
10 }
```

Listing 5.18: Module Removal and Region Release

The deactivation of the original modules (lines 2 to 4) must be performed before releasing the region (line 5), because the release of q implies that life cycle calls are not blocked anymore. Therefore, a release of q before module deactivation might result in attempts of bean instances belonging to original modules to interact with the underlying database, for example, during destruction. This in turn might lead to inconsistencies, because the database might already be accessed by bean instances belonging to the replacing modules through different persistence contexts which are not kept consistent across module boundaries. The undeployment and destruction of the original modules (lines 6 to 9) might be performed before releasing `q`. For this case clients would have to wait for those operations which would result in avoidable delays for them.

Finally, the quiescence region must be destroyed. This can be performed through invocation of the `destroy` method upon the region. The reference exchange, as declared in the end of listing 5.15 on page 206, is performed during usage. The destruction of the quiescence region also implies the removal of these declarations. Therefore, it should be performed with a sufficient delay to avoid broken connections, as already argued for the delay between the transfer from the `TRACKING` to the `BLOCKING` state of the region in the beginning of reconfiguration.

5.7. Summary

Within the previous sections the different aspects of the *mKernel* meta model and the corresponding API were discussed. In combination, they establish a comprehensive foundation for the autonomic management of EJB-based enterprise systems through the provision of a rich set of sensors and effectors.

The *Type Level* of the meta model represents a repository of module types available for deployment into a managed system. It supports the inspection of the different configuration aspects of EJB components such as required and provided interface types and SEE types, as well as transaction and security settings. Through the opportunity to request the original ejb-jar file of each module type extensibility of the meta model on *Type Level* is given. In order to manage the content of the repository, new module types can be integrated and deprecated ones can be removed. Through the explicit addressing of Java interface types, the deployment of components, as well as the reconfiguration of an existing architecture might be planned in a type safe way, as presented in section 5.2.5.

The *Deployment Level* of the meta model addresses the architecture of concrete systems and the configuration of the constituent EJB modules. In this context, all aspects of the EJB standard are addressed. The inspection of a system architecture is supported through various elements of the meta model supporting detailed analyses and the identification of adjustment demands such as the different types of states or the navigation along representations of established connections. Additionally, the opportunities to inspect the configuration of modules and beans support further investigations. The enforced type safeness of connections facilitates the avoidance of configuration errors. Regarding compositional adaptation *mKernel* reaches a level of configuration freedom which goes far beyond that of the EJB standard. Through the opportunity to reconnect enterprise beans at runtime dynamic adaptation becomes possible. For parameter adaptation simple environment entries might be set at runtime which is also not possible solely based on the EJB standard. Mapped names, and security and transactions settings might also be manipulated based on the API before module deployment. Additionally, the opportunity to integrate custom artifacts into ejb-jar files supports extensibility of *mKernel*. Through associations between *Type Level* and *Deployment Level* elements *mKernel* allows various analyses such as the identification of alternative implementations in case parts of the system should be replaced.

The *Instance Level* supports investigations regarding interactions within a managed system in an ex-post manner. Through the provided information detailed analyses are enabled, for example, regarding exceptions, transaction rollbacks, or performance aspects. Furthermore, relations to other levels of the meta model are established such as the corresponding

EnterpriseBean of an EnterpriseBeanInstance on *Deployment Level* or the MethodSpecification belonging to a JavaInterfaceType for a given BusinessCall. Call chains and call histories allow detailed analyses regarding observed interaction scenarios. Call histories might provide helpful insight into interactions of a single bean instance. In this context, state transitions might, for instance, be observed through LifecycleCal-ls while the other call types support analyses of interactions with clients or the occurrence of timer callbacks. Call chains allow the analysis of a concrete invocation context regarding those invocations which led to the considered one, as well as invocations which resulted from it. Finally, the establishment of opportunities to obtain context information supports the development of self-managing entities.

The notification facility of *mKernel* allows autonomic entities to obtain information about state changes on *Type Level* and *Deployment Level* of a managed system in a push-oriented fashion. Therefore, they are disburdened from regular inspection of a system and from potentially complex comparisons of investigation results with former observations to identify relevant changes. Regarding inspection aspects of autonomic entities, their realization should be facilitated and performance demands should be reduced. Finally, timeliness of propagation is enhanced, because notifications are published directly after the corresponding change execution.

Dynamic adaptation is supported by *mKernel* through seamless reconfiguration based on the concept of *Quiescence*. In this context, the API provides fine-grained opportunities to control the process of reconfiguration in detail. This does not only cover controlling the life cycle of a quiescence region, but also opportunities for state extraction, manipulation, and injection for instances of stateful session beans. Furthermore, the transfer of timers is supported for stateless SBs and MDBs. In combination with the opportunities provided by the *Deployment Level* of the API, various opportunities for adaptation of a system at runtime are established. These are not limited to the replacement of modules, as presented in the example in section 5.6.5. Instead of that, they might, for instance, also be used to perform consistent adjustments of SEEs or to change the state of existing stateful SB instances. The limitations regarding seamless reconfiguration mainly result from the absence of control over the interaction partners of a managed system.

6. The mKernel Realization

In the previous chapter the *mKernel* meta model was discussed. The corresponding API represents the management interface for autonomic entities consisting of various sensors and effectors. Consequently, the previous chapter presented the design of a black-box-view on managed systems. In this chapter the underlying realization of this view is presented focusing on general concepts and architectural aspects. In section 6.1 the interacting parts for system management are presented. Afterwards, in section 6.2 the preprocessing of components is discussed regarding the general tasks to perform. Subsequently, four tools which are used by different parts of *mKernel* are shortly introduced in section 6.3. Finally, section 6.4 summarizes this chapter.

6.1. System Management

Basically three main elements participate in the administration of a managed system. These are the *mKernel API*, the *Container Plugin*, and the *Managed Modules*. The elements and their relationships are depicted in figure 6.1 on page 216 including the interfaces which build the foundation for interactions among them.

The following discussion addresses each of these elements separately highlighting their particular tasks and the concepts for their realization. Section 6.1.1 starts with a presentation of the *Container Plugin*, because it is the central element of system management. Section 6.1.2 focuses on the internals of *Managed Modules*. Finally, section 6.1.3 illustrates relevant aspects of the *mKernel API* realization with respect to its internal architecture, information caching, and interactions with a managed system.

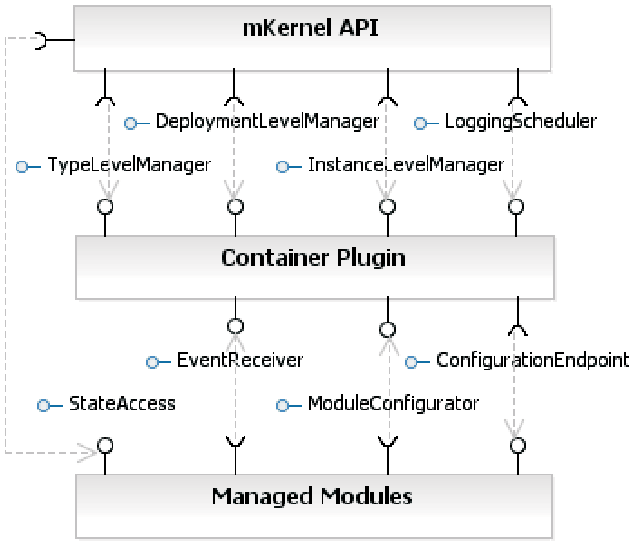


Figure 6.1.: Management Architecture

6.1.1. Container Plugin

The *Container Plugin* is realized as an EJB module consisting of a set of enterprise beans. To fulfill its tasks for supporting system management, its target container must be prepared and an underlying database must be created. Besides the integration of the plugin module the preparation of a container covers the creation of the JMS topic for publishing notifications, as discusses in section 5.5, including a corresponding connection factory. Additionally, one JMS queue and one JMS topic with corresponding factories are needed internally for the establishment of connections between message-driven beans in a managed system. Finally, one JMS queue is needed for transferring response messages from managed MDBs to managing entities.

The installation of a *Container Plugin* and the creation of all necessary resources can be automated through the execution of a command line script. It internally makes use of the *asadmin* tool [3] being part of the *GlassFish Application Server*. Although the script is limited to the execution in combination with a specific EJB container implementation, it does not restrict the application of *mKernel* within other containers. For those containers the creation of resources and the deployment of the plugin itself would have to be performed without script support.

Figure 6.2 depicts the internal structure of a deployed *Container Plugin*.

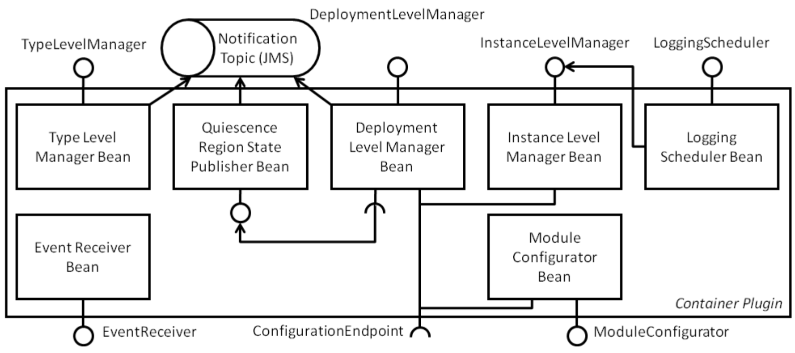


Figure 6.2.: Container Plugin Overview

The figure covers the constituent enterprise beans and their internally established connections, as well as their externally required and provided interfaces. Furthermore, connections to the *Notification Topic* are shown. This JMS topic is used for publishing notifications of *Type Level* and *Deployment Level* state changes, as well as changes regarding the state of a quiescence region (see section 5.5). The figure does not show the connection to the underlying database. Furthermore, the JMS queue and JMS topic needed for message forwarding from *Access Layer* MDBs to *Managed Layer* MDBs are not depicted in the figure, because they are not directly accessed by the container plugin. Finally, the JMS queue for reply messages to management instructions submitted to *Managed Layer* MDBs is also not covered in the figure, because the *Container Plugin* does not interact with this queue.

The set of beans can be divided into two groups. The first group consist of the five beans at the top. They provide their functionalities to the API and to managing entities. The two beans at the bottom belong to the second group. They interact directly with the managed modules for information collection and configuration submission. All enterprise beans shown in the figure are realized as stateless session beans.

Type Level Manager Bean The *Type Level Manager Bean* is responsible for all *Type Level* aspects of a managed system. For this purpose it provides the `TypeLevelManager` interface to the API. The bean exposes different methods to request information about *Type Level* elements of the managed system such as the set of managed module types. Furthermore, its instances are responsible to accept new ejb-jar files and integrate them into the repository, or to remove deprecated module types.

In order to integrate a new module type, the *Type Level Manager Bean* receives two `byte` arrays which contain an original and a preprocessed archive. The preprocessed ejb-jar file covers – besides adjusted class files and extensions integrated during preprocessing – two deployment descriptors. The first one is compliant with the EJB standard and includes all information about the constituent enterprise bean types. It was constructed from the DD of the original ejb-jar file and the annotations inside the source code of the beans during preprocessing (see section 6.2). Consequently, this descriptor contains all needed information directly related to EJB. The second DD is specific to *mKernel* and covers additional information which can not directly be represented through elements of the former DD such as meta model aspects of `JavaInterfaceTypes`. In combination, the two descriptors provide the foundation for constructing the representation of new module types in the underlying database which might later on be used to provide *Type Level* information for the API, as well as during module deployment.

After finishing the creation or removal of an ejb-jar file the *Type Level Manager Bean* accesses the *Notification Topic* and publishes the corresponding information.

Deployment Level Manager Bean The administration of a system architecture is addressed by the *Deployment Level Manager Bean*. This bean covers aspects of managing the life cycle of modules including the publication of corresponding notifications on state transitions, the initiation of adaptation actions, the management of quiescence regions, and the provision of inspection and manipulation opportunities for the API.

The life cycle of modules is managed in accordance with the states presented in section 5.3.1. After each state transition the transferring instance establishes a connection to the *Notification Topic* and publishes a corresponding message. During the creation of a module the default configuration of the corresponding module type is adopted through the creation of entries in the underlying database. In the `EXISTS` state of the life cycle no actions are performed upon an ejb-jar file. In the beginning of module deployment the bean implementation creates a copy of the preprocessed ejb-jar file, extracts the DD, and adjusts it according to the settings performed by managing entities. This covers aspects of security, transaction management, mapped names of *Access Layer* beans, and MDB message selectors. Furthermore, the message selectors of *Managed Layer* MDBs are extended to allow their compositional adaptation which is discussed in section 6.1.2.2. Afterwards, *mKernel* specific settings are integrated into the DD. These cover, amongst others, the specification of the mapped names for the configuration endpoint and the integration of SEEs for *Deployment Level* identifiers which are needed for management purposes. Consequently, all aspects which must be specified during deployment are integrated into the DD. In contrast, simple environment entries and targets for required interfaces do not become part of the adjusted DD, because these are addressed by *mKernel* separately to allow dynamic adaptation and parameter adaptation at runtime. Finally, the DD is integrated into the archive copy which is itself deployed into the target container. To perform the deployment of a module, the transfer to the `STARTED` state, as well as its stopping and undeployment, the *Deployment Level Manager Bean* internally makes use of the corresponding parts of the JSR 88 API presented in section 3.4.1. Finally, during destruction of a module all entries in the underlying database are removed. This also covers the removal of all tracked information on *Instance Level*.

Parameter and compositional adaptation of managed components becomes necessary after manipulation instructions from the API are processed by an instance of the *Deployment Level Manager Bean*. While in the deployment state `EXISTS` all types of changes are allowed upon an affected module, in the other states – except `DESTROYED` – only SEEs, connection targets for required interfaces, and connections between MDBs can be manipulated on *Deployment Level*. Valid changes are reflected through corresponding entries in the underlying database. Additionally, information about configuration changes must be transferred to all affected modules, if possible[28]. In order to transfer configuration settings, the interface `ConfigurationEndpoint` is used. An implementation of this interface is part of each managed module, because the corresponding session bean is integrated during preprocessing. The particular mapped name is specified during deployment performed by the *Deployment Level Manager Bean*. Therefore, the endpoint of each module can be identified based on information kept by the container plugin, and a corresponding reference can be looked up in the global namespace.

The management of a quiescence region is also performed by the *Deployment Level Manager Bean*. In this context, instances of the bean are responsible to store and forward instructions of region state changes to

[28] There might be more than one module affected by a configuration change, for example, by the definition of a global rerouting. A configuration transfer is possible if an affected module is in the `STARTED` state.

affected modules. All interactions regarding timer management of stateless SB and MDBs, and state extraction and injection of stateful SB instances are performed directly between the API and the affected bean instances. To identify whether the `QUIESCENT` state is reached, instances of the bean directly interact with the affected modules and their access points through the `ModuleConfigurator` interface. In this context, quiescence for a `BLOCKING` region is reached when for all of the affected modules which are in the deployment state `STARTED` no interactions are currently executed upon bean instances which are part of the region. This is sufficient, because in such a situation no new interaction can enter the region until it is released or destroyed. This also holds in case the deployment state of one or many affected modules is changed during reconfiguration. When modules are transferred to the `STARTED` state, the quiescence region definition with the corresponding state immediately takes effect which results in the blocking of interactions with elements of the newly started module as target. During other deployment states no interactions with the constituent beans are possible at all. On each state change of a region the *Quiescence Region State Publisher Bean* is informed through the corresponding interface.

For the API the session bean exposes its functionality through the `DeploymentLevelManager` interface. This interface allows the inspection and manipulation of all *Deployment Level* aspects including the management of quiescence regions.

Quiescence Region State Publisher Bean Instances of this bean are informed by *Deployment Level Manager Bean*s about state transitions of a quiescence region, if defined. On arrival, a corresponding notification message is created and published through the *Notification Topic*. In case the received state is `BLOCKING`, a timer is started which is used to inspect the region state in regular intervals. When a state transfer to `QUIESCENT` is identified, this is also published through the topic mentioned above. Additionally, the timer for regular inspection is stopped. Although this proceeding implies a polling of the region state including interactions with the access points of the affected modules, it is considered of being an appropriate solution for all conceivable scenarios, because interactions with module access points for identifying quiescence are not resource intensive. Furthermore, the timeout interval duration might be adjusted to the concrete system demands. In case a release or destruction of a region is identified, the applied timer is also stopped, because for this case no direct transfer to the `QUIESCENT` state would be possible anymore.

Instance Level Manager Bean The *Instance Level Manager Bean* provides the interface `InstanceLevelManager` to the API and to the *Logging Scheduler Bean*. It is used to request and remove *Instance Level* information from the container plugin and the underlying database. Furthermore, the interface can be used to activate *Instance Level* information tracking.

Internally, the bean manages registrations for information tracking and initiates the transfer of settings to the affected module access points through the `ConfigurationEndpoint` interface. In order to correctly finish the collection of *Instance Level* information, the bean makes use of the container timer service. A timer is created on initiation of logging and expires in the end of the logging interval. On activation and expiration of the timer the affected modules are informed about the new logging settings. Furthermore, the database is updated regarding the activation or deactivation of logging.

Logging Scheduler Bean In order to schedule logging intervals, the *Logging Scheduler Bean* is used. It accepts scheduling instructions from the API and internally configures corresponding timers. On expiration of a timer an instance of *Instance Level Logging Manager Bean* is accessed and instructed to start logging. The scheduler can also be instructed to deactivate the scheduling of logging for previously defined targets.

Module Configurator Bean As discussed in the context of the *Deployment Level Manager Bean*, module configurations are transferred through the module access points based on the `ConfiguratorEndpointInterface`. This is performed in a push-oriented fashion, for example, when certain configuration aspects are changed. Managed modules do not store configuration information persistently, but keep them in main memory. This might lead to situations where configuration settings are lost, for instance, due to container restarts or crashes. A retransfer of configuration data can be initiated through invocations of the corresponding methods of `ModuleConfigurator` upon instances of the *Module Configurator Bean*. These internally make use of the module access points through the `ConfigurationEndpoint` interface, analog to the *Deployment Level* and *Instance Level Manager Bean*s. Consequently, the *Module Configurator Bean* is used by the managed modules for on demand configuration transfers.

Event Receiver Bean The *Event Receiver Bean* acts as access point for managed modules to transfer tracked *Instance Level* information to the *Container Plugin* based on the `EventReceiver` interface. On arrival of logging information the *Event Receiver Bean* creates corresponding entries in the plugin database which can later on be used by the *Instance Level Manager Bean*.

The push-oriented proceeding for information transfer is realized to avoid the need to access all modules of a managed system or at least a subset of them in regular intervals for requesting new invocation information. This would become necessary for an alternative pull-oriented approach because of the demand for collecting complete call chains starting at those modules inside which logging is activated for at least one bean. Nevertheless, it would not be sufficient to access only those modules, because call chains might reach across module boundaries. Although the system architecture might provide helpful information to limit the set of modules to access, the opportunity to transfer bean references as parameters or return values ought to be taken into account. In summary, such an alternative would demand for analyses which might lead to uncertain results and to unnecessary communication overhead. In contrast, the realized solution only requires interactions for the original transfer of logging information.

There do exist different beans with overlapping responsibilities inside the *Container Plugin*. An example might be the *Module Configurator Bean* in combination with the *Deployment Level Manager* Bean and the *Instance Level Manager Bean*. While the latter two beans are responsible for accepting and transferring parts of module configurations, the former bean must be able to transfer all aspects of module configurations. In case of overlapping responsibilities, the corresponding source code is encapsulated in special classes called *Task*s. These tasks might be used from inside all beans of the *Container Plugin* to fulfill their duties. Through this proceeding redundant bean source code is avoided while the administrative overhead of the container is minimized at the same time. Alternatively, it would have been possible to let the *Module Configurator Bean* rely on the *Deployment Level Manager Bean* for *Deployment Level* configuration transfer. For this scenario an additional inter-bean-interaction would be necessary which would lead to an avoidable performance overhead.

Transactions regarding the manipulation of plugin data are isolated through bean managed transaction demarcation. If an instance of the *Deployment Level Manager Bean* is, for instance, used for starting a managed module, this state transition and the corresponding manipulation of the plugin database are executed in a transaction which is separated from the publication of the transition through the *Notification Topic*. Therefore, the original transition is isolated from the potential client transaction which is suspended before the corresponding invocation. Furthermore, transactions of message receivers cannot have impact on the original state transfer. Summarizing, BMTD is used to avoid external impacts on critical processes inside the container plugin regarding transactions. Otherwise, consistency between the underlying plugin database and the system architecture might be violated.

6.1.2. Managed Modules

Managed Modules are those EJB modules which are controlled by an *mKernel* system. Internally, they consist of the original artifacts being the results of component development, as well as *mKernel* specific adjustments and extensions. Figure 6.3 depicts a schematic overview of the different elements of a module on *Managed Layer*.

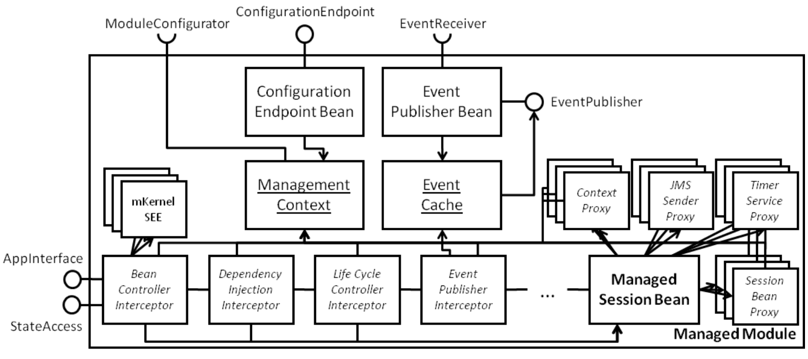


Figure 6.3.: Internals of a managed Module

In the figure the original enterprise beans are represented through a *Managed Session Bean*. Furthermore, the original interceptors attached to this bean are depicted through the three dots between the *Life Cycle Controller Interceptor* and the *Managed Session Bean*. All other elements in the figure are specific to *mKernel*. In general, the elements can be divided into two groups. The first group consists of the *Configuration Endpoint Bean* and the *Management Context*, as well as the *Event Publisher Bean* and the *Event*

Cache. These elements are needed for the configuration of a module as a whole and for the transfer of *Instance Level* events to the *Container Plugin*. The remaining elements belong to the second group. They are used in the context of managed bean instances for controlling their life cycle, interactions with their environment, and for collecting information needed for *mKernel*-based management.

The remainder of this section discusses the two groups separately, starting with the first group in section 6.1.2.1. Afterwards, section 6.1.2.2 addresses those aspects of managed components which are directly related to individual enterprise beans and, thus, belong to the second group.

6.1.2.1. Module Infrastructure

The infrastructure of a managed module consists of those elements which are not directly associated with a single bean, but address aspects of the module as a whole.

Management Context The *Management Context* represents the configuration center of a managed module. It holds all configuration information which is needed by instances of managed beans and their associated elements. This covers, amongst others, information about connections between enterprise beans and the configuration of a quiescence region, if defined. Furthermore, the *Management Context* is also responsible for configuration aspects regarding *Instance Level* information tracking. Finally, it holds identifiers of active invocations which can be requested by the different elements affected by a call to obtain context information.

In case the *Management Context* is accessed by an element for configuration discovery, and the corresponding information is not present, the context establishes a connection to the *Container Plugin* through the `ModuleConfigurator` interface and requests the publication of the corresponding information. The *Container Plugin* reacts with the submission of the corresponding configuration through the `ConfigurationEndpoint` interface, as discussed in the previous section. Such a situation might, for example, occur after a container restart or crash.

The functionality of the *Management Context* is realized through `static` fields and methods, because this makes the covered information available globally inside a *Java Virtual Machine* (JVM). Therefore, any element inside the JVM might request configuration data. In this context, the identification of the active invocation and the corresponding information such as the bean identifier, is based on the identifier of the active thread[29]. For an approach which would not be based on `static` methods and fields configuration information would have to be published along internal call chains during execution of an interaction. This was assumed of not being feasible, because a reference to a session bean instance might be used in any conceivable context, including through elements of the underlying JVM[30]. Therefore, the opportunity to publish configuration data along internal call chains would require a manipulation of the underlying *Java Runtime Environment* (JRE). This would – beyond a violation of the *COR-UCI* (Unchanged Container Implementation) requirement stated in section 1.3 – imply a serious performance overhead, because all interactions inside a JVM would be affected by information transfers. Furthermore, on configuration changes, these would also require information synchronization across all affected objects which was also assumed of not being feasible. Alternatively, configuration information could be obtained from the

[29] The identifier of the currently active thread can be requested through an invocation of `Thread.currentThread().getId()`.

[30] This might, for example, be the case if an interface defined by the Java API is implemented by a session bean and provided as business interface to clients. In such a case the connection to an SB instance might be established inside the bean source code, and the corresponding reference might be submitted to an instance of a class provided by the JVM. This instance might afterwards make use of the submitted reference in different contexts or even transfer it to other bean instances during interactions.

underlying database directly, either through accessing a central database or local caches for each module. This might require multiple database accesses for each client interaction which was assumed of leading to an unacceptable performance overhead. Therefore, a centralized solution based on locally cached configuration information inside the main memory was chosen. This solution requires write access to `static` fields and is therefore a violation of the EJB standard (cf. [58], p. 545). Nevertheless, the corresponding reasons for the limitation in the standard are not given for the *GlassFish Application Server*[31]. Consequently, the solution presented here was considered of being appropriate at least for the applied container. Summarizing, the chosen approach is feasible for storing and providing configuration information, but avoids the application of *mKernel* within environments in which the restriction of the EJB standard are relevant.

An alternative solution could be based on direct access to the *Container Plugin* database from inside the *Management Context* to avoid the need for `static` fields. This alternative was rejected for two reasons. First of all, such a solution would imply a high number of database interactions because of the different configuration settings relevant for each client interaction. Secondly, the caching strategy for the EIS-tier would have to be deactivated inside managed modules at least for accesses to the plugin database. This would become necessary, because configuration updates might occur anytime and should take effect directly. Therefore, each request for a configuration setting would require its lookup inside the corresponding database. This was assumed of inducing an unacceptable performance overhead.

Configuration Endpoint Bean This stateless session bean is part of each module and is used as endpoint for configuration transfer from the *Con-*

[31] There would arise problems if instances of beans belonging to the same module are not executed within the same JVM, because they would not share static fields. This is not given for the *GlassFish Application Server*.

tainer Plugin. In this context, it receives configuration updates through the `ConfigurationEndpoint` interface and forwards them to the *Management Context*. The particular mapped name for a module-specific endpoint inside the global namespace of a container is set by the *Deployment Level Manager Bean* during deployment of managed modules and is stored in the underlying database. Consequently, the *Container Plugin* is able to identify the corresponding *Configuration Endpoint Bean* for each of the managed modules and to submit the relevant information to the correct targets.

Event Cache The *Event Cache* is used as central, temporary storage for invocation related events to track on *Instance Level* for the corresponding module. Its implementation is analog to the one of the *Management Context* in that it relies on `static` fields for caching relevant events. The cache receives events from the different *Event Publisher Interceptor*s which are attached to individual bean instances as interceptors. Events are collected block wise from the cache in regular intervals by the *Event Publisher Bean*. To ensure the event collection through the *Event Publisher Bean*, the *Event Cache* internally stores whether the *Event Publisher Bean* will collect events in the future in a `boolean` field `tracking`. Initially, the corresponding value is set to `false`. When an event is received and no events will be collected, the cache contacts the *Event Publisher Bean* through the `EventPublisher` interface and instructs the publisher to start event collection. At the same time tracking is set to `true`. When the publisher stops collection of events, it informs the cache which resets `tracking`.

Event Publisher Bean The *Event Publisher Bean* is realized as stateless SB and makes use of the container timer service. If instructed to collect and transfer events, as discussed above, it creates and activates a timer. The corresponding timespan can be configured according to the demands of a concrete system. On timer expiration the bean removes the currently cached events from the *Event Cache* and transfers them to the *Container Plugin* through the `EventReceiver` interface if at least one event was found. If no event could be obtained from the cache, an internal counter for storing unsuccessful attempts for event collection is incremented, otherwise the counter is reset. When the counter reaches a configurable threshold, the timer is deactivated, and the cache is informed about stopping event collection performed by the publisher.

Event Cache and *Event Publisher Bean* are integrated into modules due to performance reasons. In an alternative design without *Event Cache* and *Event Publisher Bean*, *Event Publisher Interceptor*s could also submit relevant events directly to the container plugin through the `EventReceiver` interface. Such an approach would lead to an additional invocation for each interaction to track, but would not violate the EJB standard. Furthermore, each of these additional invocations would require the originally handled interaction to wait for the integration of the corresponding event into the plugin database. In combination, this would induce undesired delays for the execution of client interactions. A block wise transfer would not be feasible in this case, because timeliness of information transfer could not be guaranteed. This would be the case, because events would be transferred to the *Container Plugin* during the execution of one of the following client interactions. Their occurrence could not be predicted generally. Therefore and because no timer facility is provided for interceptors, there might arise situations where a relatively long timespan between the processing of an interaction and the submission of the corresponding event elapses.

As second alternative it would have been possible to store events directly inside the plugin database from inside the *Event Publisher Interceptor*s. Besides the need to wait for the integration of events during interaction processing, this might imply a great number of database connections, because each interceptor instance would require its own connection. Alternatively, the integration into the database could be assigned to a separate stateless SB which instances would reuse their database connections for multiple interceptor instances. This would reduce the number of required connections, but would induce additional interactions between interceptor instances and bean instances. Furthermore, database caching inside the Container Plugin and the corresponding EIS-tier respectively would be highly limited, because information updates must be received from the database at least if logging is activated and the corresponding meta model representations are requested.

Summarizing, a violation of the EJB standard was accepted due to the achieved performance gains. Nevertheless, *a standard compliant solution would also be possible.* Its integration would not induce any changes regarding the provided functionality and the external view on a managed system, but would result in a high performance overhead.

6.1.2.2. Managed Enterprise Beans

Enterprise beans managed by *mKernel* are attached with a set of interceptors for gaining control over the control flow during execution of interactions and for performing management tasks. Furthermore, all interactions with a container performed from inside the context of an interaction are intercepted through proxies, for example, to control the establishment of connections to SB instances. Finally, connections to SB instances themselves are also performed through proxies which allow connection rerouting.

Within this section the participating elements are presented with respect to their particular tasks. The discussion is organized according to the order in which the different elements are accessed for the first time during interaction execution.

Enterprise Bean Interfaces In figure 6.3 on page 225 two interfaces are depicted for the *Managed Session Bean*, namely `AppInterface` and `StateAccess`. The first interface `AppInterface` represents those interfaces which can be used to interact with instances of the corresponding bean to make use of the encapsulated functionality. For the discussion in this section it is not relevant whether the original interface on *Access Layer* or an extended version on *Managed Layer* is given. The second interface `StateAccess` is used by *mKernel* internally. For stateful SB instances on *Managed Layer* it provides methods for inspection and manipulation of the instance state. For *Managed Layer* stateless SBs timers might be requested and configured.

To inspect or manipulate the timers of MDBs, specially constructed JMS messages are used. For those messages certain properties are set which allow to identify that a management message is given. For the case that timers are requested, their representations are returned through a second message received by the original sender of the management message. The response message is sent through an *mKernel*-specific queue which is created for this purpose during installation of the *Container Plugin*.

Interceptors To each managed enterprise bean a set of *mKernel*-specific interceptors is attached. These interceptors are arranged before the set of original interceptors which were assigned to the enterprise bean based on the configuration of the original archive. Each member of the *mKernel*-specific set fulfills different tasks of bean instance management.

The *Bean Controller Interceptor* is responsible to create context information for interactions to process. This information covers, for example, identifiers of the affected module, bean, bean instance, and invocation. The first two identifiers are received from *mKernel*-specific simple environment entries (*mKernel SEE*) which are integrated during deployment. The instance identifier is generated during processing of the *postContruct* life cycle call. Finally, for each interaction a new unique identifier is created. With each interaction inside the managed system administrative information, for instance, regarding the corresponding call chain or instructions for information tracking, are submitted as parameters of invocations on SB instances or as message properties for MDB instances. This information is also integrated into the established context information. After finishing its creation the information is submitted to the *Management Context* for being accessible to other elements concerned with management aspects. Beyond context establishment the *Bean Controller Interceptor* also identifies management instructions arriving at a bean instance. Instructions such as the setting of a new state for a stateful SB instance, are not forwarded along the interceptor chain, but are realized directly through an invocation on the corresponding bean instance. Consequently, management instructions are always executed without being noticed by other interceptors or by the business logic of the affected bean instance. This is ensured, because the corresponding methods are integrated during preprocessing of the module, and the execution of direct invocations upon the target of an interaction do not pass interception methods of subsequent interceptors or the target bean instance itself. Before forwarding a life cycle invocation along the interceptor chain the interceptor analyzes whether the corresponding bean is part of a `BLOCKING` or `QUIESCENT` region. If this is the case, the forwarding of the invocation is omitted to avoid the occurrence of undesired interactions inside a quiescence region. Finally, the execution of state injection for stateful SB instances is stored internally in a field and becomes part of the context information during subsequent interactions.

For each enterprise bean an individual *Dependency Injection Interceptor* is generated and attached during preprocessing if at least one SEE or bean dependency is defined. On identification of a *postConstruct* invocation an interceptor instance requests configuration information regarding dependency injection from the *Management Context*. For SEEs the corresponding values are directly injected into the bean instance. For required interfaces references to SB instances are looked up and injected in accordance with the configuration requested from the *Management Context*. In this context, references are not directly inserted, but integrated into proxies to control interactions and to allow the rerouting of *Instance Level* connections. The underlying concepts of these proxies are discussed later on within this section. An interceptor instance internally stores whether dependency injection was performed for the corresponding bean instance. If this is not the case and an *aroundInvoke* method arrives at an interceptor instance, this is interpreted as release or destruction of a quiescence region. In this case dependency injection is caught up except for those cases where state injection for stateful SB instances was performed. Otherwise, the injected state or parts of it would be overwritten. The identification of such a situation is based on the information submitted by the *Bean Controller Interceptor*, as discussed above.

Life Cycle Controller Interceptor instances manage the life cycle of bean instances after the execution of seamless reconfigurations. In this context, they initiate the execution of necessary life cycle invocations after a quiescence region is released or destroyed, and the first interaction reaches an interceptor instance.

Instances of the *Event Publisher Interceptor* are responsible to forward interaction information to the *Event Cache* if logging is activated directly or indirectly. If logging is activated for the corresponding bean or module through the configuration obtained from the *Management Context*, direct activation is given. An indirect activation is determined from the information forwarded along interactions inside a managed system. It is given if for at least one call upward inside a call chain logging is activated directly.

Managed Bean *Managed Bean*s are divided into two groups according the layering of system architectures introduced in section 5.1. The following discussion addresses these two layers for session beans separately.

MDBs of the two layers are treated in combination because of their close relationships.

Managed Layer SBs are deployed instances of adjusted and extended bean types adopted from the original module type. To enable their management through *mKernel* they, amongst others, provide additional methods to inspect and manipulate the state of instances as well as for instance life cycle management. Newly integrated methods for state access are generated during component preprocessing and do not interfere with the original methods. Methods for life cycle management internally lead to an invocation of the corresponding methods of the original bean, if any exist. Consequently, they provide adapter methods for original life cycle methods which are needed to provide a unified interface to *Managed Layer* beans. Finally, the implementations of the extended methods providing access to the encapsulated business logic internally make use of the original ones. In this context, only the original parameters are forwarded while the parameters used to transfer management information are omitted. These are used by the *mKernel*-specific interceptors only. Consequently, the implementations of the extended methods are realized as adapter methods, too. Summarizing, there do not exist any conceivable differences for the original source code regarding its management through *mKernel*.

Access Layer SB types are generated during preprocessing of an ejb-jar file from the provided interfaces of the original SB types. Their implementations internally make use of proxies to *Managed Layer* SB instances at runtime to forward invocations. The design of these proxies is presented in the context of environment interactions discussed below. In addition to the provided interfaces of the original SB, *Access Layer* SBs expose an *mKernel*-specific interface which allows the injection of an internally used proxy.

MDB types belonging to the *Access Layer* are realized by an implementation being part of *mKernel* which forwards incoming messages to a specific JMS topic in case it is itself connected to a topic or to a specific JMS queue otherwise. The target destinations are created for *mKernel* during the installation of the container plugin. At these destinations all *Managed Layer* MDBs are registered as message receivers, depending on the *Access Layer* MDB they are connected to. This implies that a *Managed Layer* MDB which is registered at the *mKernel*-specific topic inside a managed container can only be reconnected to those *Access Layer* beans which are themselves associated with a topic. The same holds for *Managed Layer* MDBs connected to the *mKernel* queue and their reconnection to *Access Layer* MDBs registered at a queue. Only if the corresponding *Managed Layer* module is in state `EXISTS`, a reconnection from a queue to a topic is possible for an MDB, because in this case there does not exist a registration at a destination inside the container which cannot be changed after module deployment.

Figure 6.4 depicts an exemplary part of a system architecture with two *Access Layer* MDBs and two *Managed Layer* MDBs for the case of message routing through queue connections.

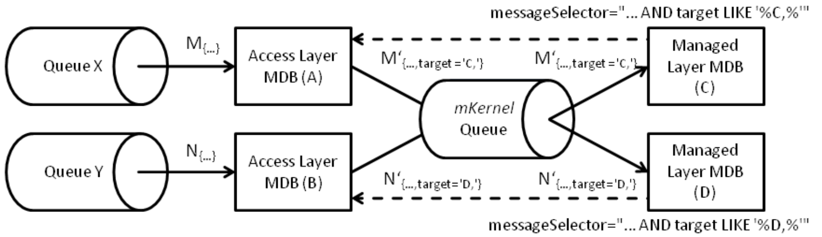


Figure 6.4.: Connection Structure for MDBs

Connections declared through the *mKernel API* are represented through dashed lines within the figure. *Access Layer MDB A* is registered as message receiver at *Queue X* through its mapped name, and *Access Layer MDB B* is registered at *Queue Y*. All *Managed Layer* MDBs are registered at the *mKernel Queue*. Such a connection is established during deployment for each MDB on *Managed Layer* which is connected to a queue listener on *Access Layer*. On arrival of a message *M* at *X* it is forwarded to an instance of *A*. As first step during processing of an incoming message the instance of *A* analyzes whether the further delivery must be delayed. This would be the case if at least one of the associated *Managed Layer* MDBs is affected by a quiescence region in state `QUIESCENT`, or if at least one of them is part of a region in state `BLOCKING` and the call chain leading to the sending of *M* did not already pass the region. If this is the case, the further processing is delayed until the quiescence region is destroyed or released. During further processing the properties of *M* are analyzed by the instance of *A* regarding the existence of management information at first. This is given if the message was sent during execution of an interaction inside the managed system. If management information is found, it is adopted and adjusted according to the current context. Otherwise, new management information is constructed. The new or adjusted information is integrated into *M*. Additionally, the message properties of *M* (shown in brackets) are extended by the instance of *A* with the identifiers of the connected *Managed Layer* MDBs (*C*). Finally, the adjusted message (*M'*) is send to the *mKernel Queue*. Each original message selector (*messageSelector*) of a *Managed Layer* MDB is extended with its own identifier through `AND` concatenation during deployment. For the case depicted in figure 6.4 on page 236 it is assumed that the different message selectors of the *Managed Layer* MDBs match with the original properties of *M*. Due to the integration of target identifiers into the properties of *M'* and the selector adjustments, *M'* can only be received by an instance of *C*, but not by an instance of *D*. Analogously, a message *N'* sent by an instance of *Access Layer MDB B* could only be received by an instance of *D*. In case of a reconnection of a *Managed Layer* MDB to another *Access Layer* MDB, the affected *Access Layer* beans would be informed about the change and would react accordingly with respect to the extension of incoming messages with target MDB identifiers. This proceeding allows the reconnection of MDBs without the need to change the configuration inside underlying container.

Environment Interaction To gain control over the different aspects of bean instances, connections to their environments are intercepted by elements of *mKernel*. All of the discussed proxies are integrated transparently during preprocessing. *mKernel* does not require that those proxies are used explicitly in the source code of enterprise beans. Consequently, there do not arise any additional requirements during component development.

Within the EJB standard interactions with the container are envisioned through so-called **Context**s. In order to interact with the global namespace, for example, to obtain references to session bean instances or to request references to JMS destinations, a reference to a `javax.naming.-Context` might be used. Moreover, session bean instances might make use of references to a `javax.ejb.SessionContext` to obtain references from their local namespace or, for stateless SBs, to request a reference to their timer service. A corresponding context[32] is provided for MDBs. Finally, the interface `javax.interceptor.InvocationContext` exposes methods for inspection and manipulation of different aspects during interception of an interaction. Invocations on all of those contexts are intercepted and manipulated by *mKernel*, if necessary. Each lookup in the global or local namespace is intercepted to realize rerouting according to the configurations performed through the *mKernel API* and to integrate proxies for references to SB instances (see below). Furthermore, the establishment of references is delayed in case this would affect a quiescence region unless it should be created from inside the context of an interaction which has already passed the region. The results of inspections performed through an `InvocationContext` are manipulated to ensure transparency of management aspects. It is, for instance, possible to re-

[32] `javax.ejb.MessageDrivenContext`

quest an array of `java.lang.Objects` covering all parameter values of the currently processed invocation upon an SB instance. This array would – if the corresponding request would be processed without interception by *mKernel* – contain the original parameter values, as well as those which are specific to *mKernel*. To make *mKernel* based management transparent for interceptors, the original array is replaced with a new one not covering the *mKernel*-specific values. An analog proceeding is performed for the case of parameter value manipulation through the `InvocationContext` interface.

Connections to session bean instances are intercepted by *mKernel* through so-called **Session Bean Proxies**. Their main tasks are to provide an adapter from the original interfaces to the extended ones, and to support rerouting and seamless reconfiguration. In order to provide the adapter functionality, instances of SB proxies expose the original interfaces of referenced SB instances to their clients. Internally, they request information to transfer along call chains from the *Management Context*. Afterwards, they use this information and the submitted parameter values to perform invocations to their referenced SB instances through the extended interfaces used by *mKernel*. If *default* rerouting should be performed, that is, existing references should be rerouted, this is supported by an SB proxy through the establishment of a connection to an instance of new the target. On each incoming invocation a proxy instance analyzes the configuration of a quiescence region, its state, and the call chain of the currently processed invocation, if a region is defined within the system. If necessary, the invocation is blocked until the region is released or destroyed.

The blocking of invocations is realized by the *Management Context*. The underlying implementation does not make use of thread synchronization primitives, because this is forbidden by the EJB standard (cf. [58], p. 545). Instead of that, instances of `java.util.concurrent.ArrayBlockingQueue` are applied. For each blocked invocation such an instance is constructed and integrated into a data structure for keeping track of blocked invocations. Afterwards, the method `take` is invoked by the affected proxy upon the queue instance. This invocation blocks until an element is available inside the queue. On release or destruction of a quiescence region the *Management Context* inserts an element into each blocking queue which results in the unblocking of the `take` invocations. A permanent blocking of invocations is avoided through an additional `ArrayBlockingQueue` inside the *Management Context*. Otherwise, it would be possible that interleaving requests on the *Management Context* lead to an omission of an invocation release. An exemplary scenario is depicted in figure 6.5 as sequence diagram. The included interactions are simplified versions and do not directly represent the original source code. Instead of that, they are intended to provide a schematic insight into the avoided problem.

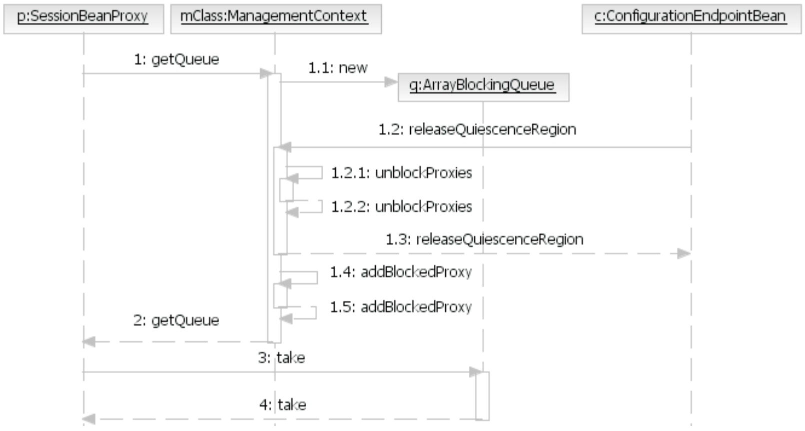


Figure 6.5.: Scenario of permanent Blocking without Synchronization

In this scenario the proxy to block `p` performs a `getQueue` invocation (1) to obtain a queue reference (`q`) from the *Management Context* (`mClass`). During execution of this invocation the requested queue is created first (1.1). Afterwards, it is added to the set of queues to unblock on release or destruction of a quiescence region. This is depicted through the invocation of `addBlockedProxy` (1.4). During unblocking of proxies due to the release of a region (1.2.1) elements are added to all queues contained in the set of queues to unblock. If unblocking of queues occurs between the creation of a new queue and its integration, as depicted in the figure, no element would be added to this queue during unblocking, because it is not part of the set of queues yet. Therefore, the corresponding proxy would not get unblocked from the subsequently performed `take` invocation upon `q` (3). As no quiescence region is active after the unblocking, this might result in a permanent blocking of the affected proxy. To avoid this and comparable scenarios, an additional queue is used by the *Management Context* which is constructed with a capacity of one. Furthermore, the queue is initialized as fair queue meaning that invocations upon the instance are processed in first-in-first-out order, if possible. In the beginning of an execution of `getQueue` and `releaseQuiescenceRegion` an element is added to the queue through invocation of the `put` method. This method blocks if there is no space left inside the queue. Invocations of `releaseQuiescenceRegion` could not interleave with invocations of `getQueue` because of the capacity of one. In the end of an execution of `getQueue` and `releaseQuiescenceRegion` the previously inserted element is removed again freeing the critical section for other invocations.

The presented solution does not violate the EJB standard with respect to the usage of thread management primitives. Nevertheless, the queues for blocking proxy invocations are stored within a `static` field by the *Management Context*. Therefore, the same case of standard violation is given, as already discussed earlier within this section. An alternative approach would have been to register each proxy to block as message receiver at a JMS topic and to perform a blocking request for receiving a message from inside the proxy source code. In case of quiescence region release or destruction, a message could be send to the topic to release all blocked proxies[33]. Depending on the concrete reconfiguration context and the af-

[33] To avoid the permanent blocking problem, the receiving could be performed with a time-

fected system, a great number of proxies might require to be blocked. This would result in the same number of topic registrations which might lead to an unacceptable administrative overhead for the underlying container. Therefore, this alternative was rejected.

According to the EJB standard it is valid that enterprise bean instances transfer references to SB instances during interactions. This also covers the transfer of references to external clients, for example, as return value of a method invocation. In the context of *mKernel*, situations are conceivable where a *Managed Layer* SB instance returns an SB proxy as result of a method invocation. If this proxy is subsequently transferred to an external client, the proxy to transfer must pass an *Access Layer* bean instance, because clients do access *Managed Layer* SBs only indirectly through these instances. The container serializes the proxy before transferring it to the *Access Layer* bean instance and deserializes it again on arrival. Based on the interaction context information provided by *mKernel*, the proxy would determine that it is deserialized in the context of an *Access Layer* SB bean instance. For this case, it initiates its own replacement with a direct reference to an instance of an *Access Layer* SB bean which provides the same Java interface. The exchange itself is performed in four steps.

1. The proxy searches for a matching *Access Layer* SB which provides the required Java interface and is connected to the same *Managed Layer* SB as the reference encapsulated in the proxy instance. If no such bean is given, this can be interpreted as insufficient configuration of the system, because the session bean instance which is referenced by the proxy should not be exposed outside the managed system.
2. A reference to the *Access Layer* SB is created using the global namespace.

out. After its expiration a blocked proxy could interact with the *Container Plugin* to find out whether the region does still exist. If this is not the case, the proxy could continue processing.

3. The proxy to the *Managed Layer* SB instance is injected into the *Access Layer* SB instance. This step is only necessary if a stateful SB instance is referenced, because all references to instances of a single stateless SB are equivalent. In order to inject the proxy, the corresponding interface provided by each *Access Layer* SB is used.
4. The reference to the *Access Layer* SB instance instead of the proxy instance is returned as result of deserialization.

The *Access Layer* SB instance through which the reference is passed as return value does not use it, but only forwards the reference to the client. Therefore, no interactions between bean instances would take place inside a managed system directly without proxies for the presented solution.

A second scenario would be the case that a bean instance tries to transfer a proxy to an externally used interaction partner, for example, as parameter value during a method invocation. This case is identified by affected proxies through analyzing information obtained from the *Management Context*. If such a situation is given, the proceeding for replacing the proxy with a reference to an *Access Layer* SB instance is the same as discussed above.

Both replacement scenarios discussed above demand the substitution of proxy objects with reference objects. From the point of view of a client, the provided interface would not exhibit any difference. Nevertheless, the EJB standard forbids object substitution during serialization due to security reasons without further explanations (cf. [58], p. 547). Consequently, the above stated approach represents a violation of the EJB standard because of the integration of object substitution into the behavior of proxies during serialization and deserialization. An alternative to the above stated proceeding would be, to transfer proxy instances to clients and to replace the internally used reference to a *Managed Layer* SB instance with a reference to an *Access Layer* instance similar to the steps discussed above. Furthermore, proxies would have to be able to internally make use of original interfaces. This is already given, because *mKernel* was designed to also allow interactions initiated by managed bean instances of which the target is an unmanaged SB instance. For the corresponding references, proxies are also integrated. The transfer of proxy instances to external clients would induce three major disadvantages which led to a rejection of this alternative approach:

1. The transfer of proxy instances would require the availability of the corresponding class files inside the client environment. This would result in the demand for clients to integrate the classes needed for proxy usage through class loading anyway such as the integration of an additional library into their class path. Consequently, the application of *mKernel* in a managed system would state additional requirements for clients affected by proxy transfers and would therefore not reach the level of transparency as the applied approach.
2. The proceeding for replacing a reference to a *Managed Layer* SB instance with a reference to an *Access Layer* SB instance would require access to management functionality from inside the client environment. This was assumed of inducing very much higher security risks than the execution of nearly the same source code inside a controlled container environment during serialization or deserialization.
3. Finally, the replacement of a reference requires certain interactions with a managed system. If these are performed inside the container based on information being available inside the affected module, this causes much lower performance overhead than the corresponding interactions between clients and a managed system which are potentially performed across container boundaries from inside a web container, or even over a network infrastructure.

In addition to the above stated disadvantages, it must be highlighted that classes which contain source code for object substitution are only used inside a managed container. They are not exposed to clients and therefore might not be the source of security attacks. Therefore, the potential security risks, which were provided as reason for forbidding the usage of object substitution during serialization or deserialization, are not evident for the chosen approach.

Beyond attempts to transfer proxies during method invocations it is also possible that a bean instance tries to transfer a proxy as part of the content of a JMS message. For this case it cannot be determined in general if the receiver of the message would be a managed MDB or an external interaction partner, for instance, if managed MDBs and other JMS receivers are registered at the same queue or topic. For the design of *mKernel* it was decided to transfer proxies through JMS destinations, although it would also have been possible to replace them with references to *Access Layer* SB instances, analog to the proceeding presented above. The reason for this decision was that a transfer of references would lead to a loss of control and information richness. If references would be transferred to managed MDBs, information about call chains would get lost in case of invocations performed through the *Access Layer*. This would be the case, because the original interfaces do not support the transfer of management information. Therefore, *Instance Level* information would not reflect interactions correctly, because invocations performed by *Managed Layer* SB instances through *Access Layer* instances would be interpreted as externally arriving. For the same reasons seamless reconfiguration might become limited, because for an invocation performed through the *Access Layer* it cannot be identified if the corresponding call chain has already passed a quiescence region. Therefore, a deadlock might arise although the corresponding interactions would solely be performed within a managed system. Finally, direct references could not be rerouted which would limit opportunities for dynamic adaptation[34].

In addition to the previous scenarios, it is also conceivable that an ex-

[34] Nevertheless, a manipulation of one single line of the *mKernel* source code would be sufficient to change the behavior of a system in a way that proxies are replaced with references to *Access Layer* SB instances in the context of JMS based messaging.

ternal interaction partner transfers a reference to an *Access Layer* SB instance back to the managed system through the same channels as discussed above. Such a reference does not necessarily need to be passed as parameter value or message content on top level, but might reside deeply within an object hierarchy associated with a parameter value or message content. Therefore and because enterprise bean instances are not allowed to use reflection for the inspection of object hierarchies (cf. [58], p. 545), it is not possible to investigate whether a relevant reference is submitted in general. In case a reference to an *Access Layer* SB instance is submitted to the managed system and is used by *Managed Layer* SB instances, this would not limit their ability to interact as desired. Nevertheless, system management would be affected the same way as discussed above.

Invocations for sending a message to JMS-based destinations are intercepted by so-called **JMS Sender Proxies**. These proxies enrich an outgoing message with management information, for instance, regarding the current call chain or activated logging. This information is integrated through additional message properties. Therefore, the original content and the original properties of a message are not affected by these extensions.

Finally, **Timer Service Proxies** intercept attempts of stateless SB or MDB instances to interact with a timer service. Internally, a set of blocked timers is managed with the aid of the *Management Context* for the case that a quiescence region requires the blocking of timers. On arrival of an invocation for the creation of a new timer it is first analyzed whether a quiescence region does exist within the system, covering the currently considered bean. If this is the case and the region is in state `BLOCKING` or `QUIESCENT`, the creation of the timer is avoided, and the necessary information for later activation of a matching timer is constructed and integrated into the set of blocked timers. Otherwise, the invocation is forwarded to the original timer service. In addition to the interception of timer creation, *Timer Service Proxies* are also used by managed bean instances for deactivating existing timers when a relevant quiescence region is transferred to the BLOCKING state. To realize timer deactivation a proxy requests all active timers for the corresponding bean from the original timer service, extracts the relevant information for reactivation, and cancels their execution. Afterwards, entries for all canceled timers are integrated into the set of blocked timers.

6.1.3. Application Programming Interface

The external view on the *mKernel API* was introduced as interface to a managed system for autonomic entities in chapter 5. In this section an overview of the API implementation is presented including the main elements and the relationships between these elements. This section does not aim to provide a comprehensive and detailed insight into all API aspects, but presents the general design decisions. Figure 6.6 depicts an extract of the API implementation covering the relations between selected elements. These can be applied analogously to the remaining elements of the API implementation.

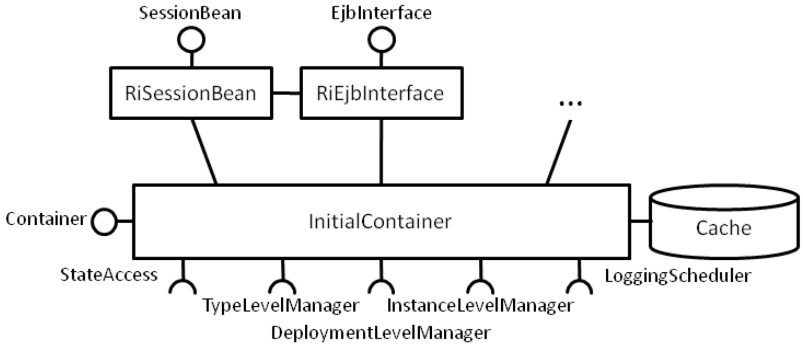


Figure 6.6.: Overview of API Implementation

The figure shows representatives of the main elements of the API and relationships among them, as well as the required interfaces which must be provided by a managed system. These are used by the API as foundation for inspection and to perform management actions.

The class `InitialContainer` is used as mediator between the elements of the API implementation and the managed system. It provides the `Container` interface as access point to the managed system for autonomic entities. A reference to an instance of `InitialContainer` is returned on invocation of the `static` method `getNewContainer` on `ContainerFactory` (see chapter 5). Instances of the `InitialContainer` class interact with the *Container Plugin* through the interfaces `TypeLevelManager`, `DeploymentLevelManager`, `InstanceLevelManager`, and `LoggingScheduler`. Furthermore, an `InitialContainer` interacts with session bean instances through the `StateAccess` interface, for example, during seamless reconfiguration for requesting timer information or for accessing the state of a stateful SB instance. Finally, it establishes connections to instances of managed MDBs through the *mKernel*-specific JMS queue or topic which are also used for forwarding messages from *Access Layer* to *Managed Layer*. In this context, nearly the same proceeding is performed as the one discussed in the previous section. To only interact with an instance of the desired MDB, the target identifier of that bean is set as value of a message property. A corresponding part is integrated into the message selector of each *Managed Layer* MDBs and connected to all remaining selector parts through `OR` concatenation. Consequently, *Managed Layer* MDBs would either receive messages sent by *Access Layer* MDBs based on their extended original message selector, as discussed in the previous section, or they would accept management messages explicitly addressed to them. These are not processed by MDB instances, but by *mKernel*-specific interceptors.

In order to receive results of management messages such as timer information, instances of `InitialContainer` make use of a JMS queue which is created during container plugin installation for this purpose. At this queue an `InitialContainer` registers with a special message selector before sending a message to a managed MDB. In response to the received message and after execution of the corresponding actions the MDB instance sends a response message to the queue the `InitialContainer` is listening to, containing information about the results of execution. Within this message matching message properties ensure that only the requesting `InitialContainer` receives that message.

Each interface of the API is implemented through a corresponding class, as depicted in the upper part of figure 6.6 on page 247 such as the class `RiSessionBean` which implements the interface `SessionBean`. In this context, the prefix `Ri` of the class name indicates that it is a member of the API reference implementation. Between instances of API classes different associations might exist such as associations between a session bean representation (`RiSessionBean`) and its provided interfaces (`RiEjb-Interface`).

Instances of API elements are received either as results of invocations on an instance of `InitialContainer` or from invocations on other API element instances. An example of the former type of invocations is an invocation of `getEjbModules` which delivers a set of representations for all modules within a managed system. From each element of this set all included `EnterpriseBeans` might be requested through an invocation of `getEnterpriseBeans` which is an example of the second type of invocations.

All interactions with a managed system are performed through an instance of `InitialContainer`. If, for example, all `RiSessionBeans` for a session bean type are requested from an `RiSessionBeanType` through an invocation of `getSessionBeans`, this request is forwarded to an `Initial-Container` which performs the original interaction with the container plugin. As results of interactions with the plugin instances of API elements are returned. Consequently, all instances of API elements pass an instance of `InitialContainer`. This allows the container to inject a reference to `this` into each received element which might require a mediated interaction with the *Container Plugin* later on. To avoid the need for traversal over object hierarchies, references to an `InitialContainer` are forwarded to sub elements on injection. This would, for example, be performed by an instance of `RiSessionBean` upon each of its associated `RiEjbInterfaces` on container injection. Due to the different navigation opportunities there would exist the threat of endless loops[35]. These are avoided through a test whether a container reference was injected earlier. If this is the case, no actual injection is performed, and no reference is forwarded.

mKernel is developed to enable concurrent system management performed by independent autonomic entities. Therefore, it would be possible that multiple entities perform changes of a system and thus influence their views on the system mutually. In order to keep the reflective nature of the meta model, only immutable associations are established during information transfer from a *Container Plugin* to an `InitialContainer`. An example of an immutable association would be the relation between a session bean and its provided interfaces. A mutable association is, for instance, given for a connection between a required and a provided interface, because it might be changed through management actions. Additionally, mutable properties of element instances such as the deployment state of a module or the value of a SEE, are also not set for transferred representations. Instead of that, these aspects of a system representation are always requested from a *Container Plugin* if accessed through method invocations upon instances of API elements.

Internally, instances of `InitialContainer` make use of a *Cache* for reducing interactions with a *Container Plugin*. In this cache, amongst

[35] An `RiEjbInterface` might, for instance, be obtained as part of an `RiSessionBean` or as connected interface for a required interface. Because of the former navigation opportunity the SB representation must forward the injection of a container reference to the `RiEjbInterface`. For the latter case an `RiEjbInterface` must forward the injection to the corresponding SB representation. Without any further restrictions for injection forwarding both cases in combination would result in an endless loop.

others, mapping from unique identifiers to bean representations or from identifiers to module representations are stored. The cache size for each type of elements is limited, but might be configured. Entries of the cache are removed following the *least-recently-used* strategy. The cache is used to perform local lookups before interacting with a *Container Plugin*, for instance, if a representation of a module or enterprise bean is requested by its identifier. Additionally, the cache is used to reduce the size of transferred object hierarchies from a *Container Plugin* to an `InitialContainer`. This is, for instance, performed during navigation along call chains and call histories on *Instance Level*. For this case only *Instance Level* information is transferred while associations to the *Deployment Level* and the *Type Level* are not resolved. For those associations only identifiers of the corresponding elements are kept inside the transferred elements. If these associations are used for navigation, for example, to request a reference to the corresponding enterprise bean of an `EnterpriseBeanInstance`, first of all the cache is accessed based on the covered identifier. Only if it could not deliver a result for the bean identifier, the *Container Plugin* is accessed.

In order to support embedded inspection as discussed in section 5.4.4, the corresponding class `CallContext` makes use of the *mKernel*-specific simple environment entries which are attached to each bean during deployment. Additionally, a context instance interacts with the corresponding bean instance for state inspection and manipulation. For this purpose the current `javax.interceptor.InvocationContext` must be submitted as parameter to the corresponding method of `CallContext`, because it allows to request a direct reference to the bean instance. For this reference it is ensured during preprocessing that it provides the `StateAccess` interface. This also holds for MDBs, because the interface is needed by the *mKernel* interceptors. The interface specifies methods which are sufficient for providing the inspection and manipulation functionality of `CallContext`.

6.2. The mKernel Preprocessing Tool

The preprocessing tool of *mKernel* accepts EJB compliant ejb-jar files as input and constructs manageable archives as output. The emitted archives can be integrated into a managed system as module types through the API, as discussed in section 5.2.4. The discussion of the preprocessing tool is divided into three parts. First of all, the architecture of the tool is presented in section 6.2.1. Afterwards, section 6.2.2 discusses the major tasks during preprocessing. Finally, section 6.2.3 presents alternatives for the construction of *Access Layer* archives against the background of management aspects.

6.2.1. Tool Architecture

The tool is designed as extensible and modular foundation for the processing of ejb-jar files. Its basic building blocks and the artifacts needed for its execution, as well as the results of processing are depicted in figure 6.7 on page 253. The gray shaded elements of the figure represent artifacts which are created or manipulated during preprocessing.

Internally, the preprocessing tool consists of two major elements, namely the *Processing Controller* and the *Target Representation*. In combination, these two elements provide the infrastructure for so-called *Processing Modules* which encapsulate the different functionalities required for successful preprocessing.

Processing Modules are provided through Java classes which realize the interface `ProcessingModule`. This interface demands for the implementation of two methods, one for configuring a concrete instance and one for starting its execution. Furthermore, a `public` standard constructor must be provided by each module implementation which allows the creation of an instance through invocation of the `newInstance` upon a corresponding `java.lang.Class`. To include a module during preprocessing, the module class itself and all required utility classes must be integrated into the class path of the tool, for instance, through packing them into the preprocessor archive. The configuration of the tool regarding modules to execute during preprocessing of an ejb-jar file is provided through a so-called *Module Configuration*. This file-based configuration of the tool consists of FQNs of modules to be incorporated during preprocessing. Therefore, the FQN of each module which should be executed during preprocessing must be integrated into that configuration file. A basic configuration is provided as part of *mKernel*. An extension of the tool, for instance, to integrate new features during preprocessing, would require the development of a corresponding module, its integration into the tool archive, and the addition of a corresponding entry into the *Module Configuration*. Consequently, it is possible to extend the tool for specific AC domains with respect to the preprocessing of ejb-jar files.

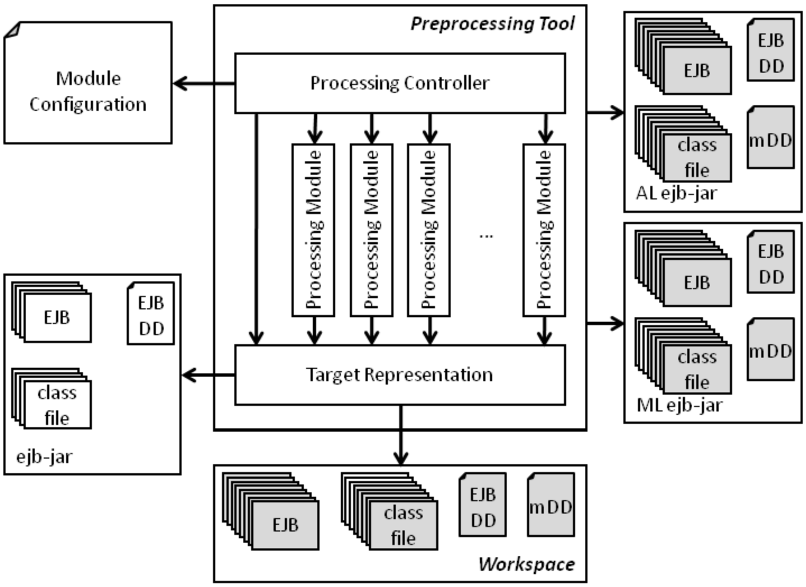


Figure 6.7.: Architecture of the Preprocessing Tool

On execution of the preprocessing tool the *Processing Controller* is instantiated first, submitting the relevant information such as the FQN of the ejb-jar file to process and additional elements for the class path which should be considered during preprocessing. These additional elements might, for instance, be container specific extensions which do not need to be part of the ejb-jar file to process, but are available for all deployed modules inside a target container. The controller first of all analyzes whether the submitted information is correct, that is, if the ejb-jar file to be processed and the class path extensions do exist. If this was successful, it instantiates the *Target Representation* which provides access to the original ejb-jar file and to a *Workspace* directory which builds the foundation for the archive to construct as result of preprocessing. Afterwards, the controller instantiates and configures all modules listed within the module configuration. If none of the instantiations and configurations threw an exception, the foundation for the original preprocessing is given. Finally, the execution of the modules is initiated in the order the modules are found in the *Module Configuration*.

The *Target Representation* encapsulates the original ejb-jar file, as well as the processing *Workspace*. In this context, it enables processing modules to access the covered elements without the need to know their concrete location in the underlying file system. Furthermore, the *Target Representation* provides opportunities for inspection and manipulation of preprocessing aspects which are not reflected in the original ejb-jar file or the *Workspace*. Examples of these are a representation of the processed archive which is held in main memory or newly created classes which are not yet written to the *Workspace*.

No inspection or manipulation of the original ejb-jar file or the target ejb-jar file are performed by the constituent elements of the tool. Instead of that, all steps from the extraction of the archive up to the packing of the emitted files are realized through the different processing modules. This also covers the establishment of the archive representation inside the *Target Representation*. Consequently, no element of the tool infrastructure does perform any actions upon the source or targets of preprocessing.

To facilitate the development of processing modules beyond the provision of unified access to the *Workspace* and the original archive through the target representation, different utility classes are part of the tool. These support, amongst others, the identification, extraction, and integration of jar file elements, as well as the generation of *Type Level* identifiers and analyses of bean methods.

6.2.2. Preprocessing Tasks

The preprocessing tool might be used through a command line script. Internally, the script prepares the environment for the tool with respect to the identification of necessary environmental information and afterwards starts the execution of the tool. As discussed in the previous section, the actions performed during preprocessing of an ejb-jar file are determined by the *Module Configuration*. Each of the listed modules is responsible for performing a special task. Consequently, the tool might be extended through the integration of new modules. Furthermore, certain actions might be excluded through removing the corresponding entry from the *Module Configuration*.

For the construction of the two archives which are emitted as results of preprocessing some of the tasks to perform are equal while others are specific for one of the two archives. Therefore, the tool requires two configuration files which are used as foundation for the construction of the particular archive. In the remainder of this section, the different tasks to perform for the construction of a *Managed Layer* ejb-jar file are discussed first. Afterwards, the differences regarding the generation of an *Access Layer* archive are highlighted. The discussion does not present the concrete preprocessing modules in detail, but gives an overview of different groups of tasks to perform.

6.2.2.1. Managed Layer Archive Construction

The construction of *Managed Layer* ejb-jar files requires the addressing of all aspects discussed in section 6.1.2.2. The processing modules which fulfill these demands can be divided into six groups, namely *Infrastructure Preparation, View Creation, Bean Manipulation, Environment Encapsulation, Interceptor Integration*, and *Target Archive Construction*. These groups are introduced in the remainder of this section.

Infrastructure Preparation The preparation of the infrastructure includes two steps. First of all, the foundation of the *Workspace* must be established. This is performed through the extraction of the original archive. Secondly, a class loading infrastructure must be constructed which allows the inspection and manipulation of the class files from the *Workspace*. In combination, these two steps lay the necessary foundation for the subsequent actions to perform. All actions performed by modules of this group do not address any aspects which are specific for the source archive.

View Creation Based on the established workspace and the class loading infrastructure, the modules of this group inspect the artifacts of the processed archive and construct a comprehensive view of it. For this purpose two deployment descriptors are used. The first one is given through the original DD extracted from the archive. The second one is constructed from the annotations identified in the class files being part of the archive. These two DDs are afterwards merged according to the rules defined in the EJB standard. In this context, declarations inside the original DD are prioritized over specifications contained in annotations if both address the same aspects of an ejb-jar file. Beyond the construction of the comprehensive DD different parts of the *Target Representation* are initialized such as aspects of identified Java interfaces and the corresponding inheritance hierarchies.

The modules of this group are responsible to lay the archive-specific foundation for further processing. At the same time no manipulation is performed on any artifact of the original ejb-jar file.

Bean Manipulation The manageability of enterprise beans is established by this group of modules. Consequently, all of the performed actions directly relate to interactions between bean instances through required and provided interfaces. In contrast, interactions with MDBs based on JMS are not addressed by this group of steps.

As preparation for bean manipulation all interfaces contained within the original archive are taken as foundation for the generation of new interfaces supporting *mKernel* management. In this context, each provided method of business interfaces[36] is extended with additional parameters for management information transfer. The names of the new interfaces are derived from the names of the original ones attached with an *mKernel*-specific suffix. For local home and home interfaces replacing interfaces are created, too. Within these interfaces, the return types of methods for the creation of SB instances are adjusted to provide access through `StateAccess`. In this context, an extended local or remote business interface is constructed for each original local or remote interface. Afterwards, the new interface is declared to be provided by the affected SBs to ensure that the corresponding functionality is still accessible. The `StateAccess` interface specifies – besides the management endpoint discussed before – a method for requesting references to the target bean based on any provided interface. Consequently, it is possible to perform management actions and to obtain any reference which might be needed by an SB proxy through a `StateAccess`-based reference. Therefore, proxies are also enabled to obtain references based on those interfaces they require for providing the functionality exposed to clients. The replacement of the original

[36] Local business and remote business interfaces.

local or remote interface with a corresponding generated business interface inside the *Managed Layer* cannot be recognized by bean instances making use of proxy instances, because these still expose the original interfaces to their clients (see below). It is not sufficient to consider only those Java interfaces which are used for required or provided interfaces of beans, because the remaining Java interfaces might build the foundation for connection establishment through lookups within the global namespace or during reference transfer between bean instances. Additionally, it might still be possible that Java interfaces from external libraries or the JRE are also used as foundation for connections. Nevertheless, only the Java interfaces being part of the processed archive are considered for extension. Otherwise, all Java interfaces within the class path – including those of the JRE – would require to be extended and integrated into the constructed archive. This would result in an indefensible storage overhead. Therefore, all Java interfaces which are used as foundation of connections between bean instances must be contained within the processed archive.

After the generation of Java interfaces all SB classes are extended to provide the relevant new interfaces. For the extended interfaces new methods are integrated into the classes which internally delegate an invocation to the corresponding methods originally provided by the SB. The additional parameters are ignored, because they are only needed by *mKernel*-specific interceptors, as discussed in section 6.1.2.2. The methods of MDB classes for receiving messages do not need to be addressed by the preprocessing tool, because management information is transferred through message properties which do not result in any adjustment demands on receiver side. Afterwards, all bean classes are extended to provide the `StateAccess` interface.

SB proxy classes are generated for all Java interfaces contained within the original module. Depending on the type of exposed interface, each generated proxy class extends a corresponding basic implementation being part of *mKernel*. These implementations provide methods which are needed as foundation for realizing the behavior discussed in section 6.1.2.2. Each proxy class implements one of the original interfaces. For each of the corresponding methods an implementation is generated which internally makes use of the provided functionality of the particular super class, for example, to block an invocation if necessary for ensuring quiescence or to replace the underlying reference. In order to forward method invocations, the necessary management information is requested from the *Management Context*, and a corresponding invocation making use of the extended interface is performed on the reference to the *Managed Layer* target.

Finally, the EJB DD is adjusted regarding declarations of provided interfaces of SBs. In this context, the newly created interfaces are used to replace the original ones. Furthermore, *mKernel*-specific interfaces are also declared to be provided.

This group of modules mainly addresses the establishment of a foundation for supervising and controlling relationships and interactions within a managed system.

Environment Encapsulation In order to encapsulate interactions with the environment of bean instances, corresponding proxies must be integrated into the beans. The general proceeding of proxy integration is based on the replacement of references during their obtainment or on the interception of interactions. In this context, proxies for the global naming context are, for instance, integrated through replacing each constructor invocation of `javax.naming.InitialContext` with the construction of a corresponding context proxy. For context proxies which allow the obtainment of references to session beans, mappings from original interfaces to their *mKernel*-specific counterparts are generated which support the instantiation of matching SB proxy instances for received references.

Through this group of modules control over interactions with the container environment is ensured.

Interceptor Integration For the basic integration of the *mKernel*-specific interceptors it is sufficient to adjust the DD accordingly, that is, to integrate a declaration for the particular interceptor and specify that the interceptor should be attached to each affected bean.

While the other interceptors are attached without changes to each considered bean, dependency injection interceptors are specific to each affected bean. If no dependency injection is required for a bean, no corresponding interceptor is attached at all. Otherwise, an individual interceptor is generated considering all identified injection declarations. Afterwards, the interceptor is attached to the bean. Finally, all affected injection declarations are removed from the bean class file, as well as from the DD to prevent container-based dependency injection.

Target Archive Construction In the end of preprocessing the target archive is constructed. As a first step the two *mKernel*-specific SBs presented in section 6.1.2.1 are integrated into the DD. These are the *Configuration Endpoint Bean* and the *Event Publisher Bean*. Afterwards, all classes which were generated or manipulated during preprocessing are written into class files inside the workspace. Moreover, class files which are part of *mKernel* and which are needed inside the target archive such as the class files of the *mKernel*-specific beans, are extracted from the tool archive into the workspace. Subsequently, the resulting DD (*EJB DD*) and the *mKernel*-specific DD (*mDD*) are written to the workspace. At this point of preprocessing all artifacts of the target archive do exist within the workspace. Finally, the content of the workspace directory is packed into the target ejb-jar file, and the directory is removed from the file system.

6.2.2.2. Access Layer Archive Construction

For the construction of an *Access Layer* ejb-jar file the proceeding is similar to the one for a *Managed Layer* ejb-jar file. The main difference during the construction of *Access Layer* archives is caused by the fact that the constituent beans of the original archives are substituted with *mKernel*-specific implementations which act as proxies for external clients. Therefore, the second group of modules (*View Creation*) is extended while the remaining groups only require a subset of the modules applied for the construction of *Managed Layer* ejb-jar files. In the remainder of this section the differences of the construction proceeding are highlighted.

Infrastructure Preparation This group consists of the same set of modules applied to *Managed Layer* archives.

Bean Replacement After the creation of the comprehensive DD and the internal representation each of the identified SBs is replaced with its *Access Layer* implementation. The replacing bean extends a generic implementation being part of *mKernel* which encapsulates general management functionalities. Furthermore, method implementations are generated for each of the originally provided methods. For MDBs it is sufficient to replace the original implementation with a generic one, because the content of messages is irrelevant on *Access Layer*. Therefore, messages do not require individual treatment. Finally, the original bean classes are removed from the workspace. Moreover, the corresponding entries in the DD are adjusted to refer to the *Access Layer* implementations. For SBs the management interface for setting a reference in the context of reference transfer to clients is added to the set of provided interfaces.

Summarizing, this group of steps is derived from the original *View Creation*. It is extended with modules for replacing the original beans with matching implementations for the *Access Layer*. This requires the generation of bean implementations, and manipulations of the workspace and the target representation.

Interface Manipulation This group consists of a subset of modules being part of the *Bean Manipulation* group applied for *Managed Layer* archive construction. All aspects of bean manipulation are excluded from the group, because all requirements of *Access Layer* beans are already reached during bean replacement. In this context, only the generation of proxies is performed for identified interfaces, because these proxies are used by *Access Layer* SB instances for forwarding requests.

Environment Encapsulation This group of modules is applied to *Access Layer* archives the same way as to *Managed Layer* archives. Only the newly integrated beans are excluded from the modules, because their source code shows the desired behavior for environment interaction. This is done, because transferred objects might try to interact with their environment during deserialization or serialization. Nevertheless, the corresponding behavior at runtime might lead to undesired side effects, for example, because the entries of the local namespace of the original target would not be available within *Access Layer* beans. Furthermore, interactions could be performed more than once, because the corresponding object would be deserialized twice, one time at the *Access Layer* and one time at the *Managed Layer*. This could not be avoided in general, because a generic manipulation of serialization and deserialization methods was considered of not being possible.

The remaining two groups *Interceptor Integration* and *Target Archive Construction* are applied to *Access Layer* modules the same way as to *Managed Layer* modules.

6.2.3. Access Point Distribution

Regarding the generation of *Access Layer* ejb-jar files, the preprocessing tool is not limited to the construction of archives based on enterprise beans providing business logic. In contrast, it would also be possible to create an ejb-jar file consisting of MDBs or SBs for which it is only specified that they provide one or many interfaces. The corresponding implementation could be left empty as long as the requirements of the EJB standard are fulfilled. Such an archive might be submitted to the preprocessing tool which generates the corresponding *Access Layer* and *Managed Layer* archives. While the *Managed Layer* archive would not provide any meaningful functionality, the *Access Layer* archive might be used for establishing an *Access Layer* architecture which deviates from the one on *Managed Layer* with respect to the distribution of beans upon modules. This might be helpful, for instance, to establish the *Access Layer* architecture, because it provides a high degree of freedom for the distribution of client access points upon modules. As one extreme it would be possible to construct archives which consist of only one enterprise bean providing one single functionality. The other extreme would be the integration of all external access points of a system into one single archive. While this would not make any difference for client interactions, it might have certain impacts on system management. Regarding reconfiguration the first extreme enables an isolated reconfiguration of single access points while demanding for the management of a potentially high number of modules on *Access Layer*. In contrast, the second extreme would lead to one single module on *Access Layer*, but would result in a temporal unavailability of all client access points even if only parts of the access points should be replaced. This is the case, because all access points would be affected by deployment actions.

6.3. Applied Tools

In order to realize the different parts of *mKernel*, four tools were applied. In particular, tools are used for XML processing, for performing deployment operations, for byte code inspection and manipulation, and for source code generation. This section gives an overview of the applied tools. It is not intended to provide an insight into each of the tools, but only sketches the particular application context.

Deployment Operations The execution of deployment operations within the container plugin is based on the corresponding API of the *JSR 88* [64]. It was chosen, because it supports all functionalities needed by *mKernel*. Furthermore, it is an integral part of the *Java EE 5* specification [140] and is therefore deeply integrated into a broader context building the foundation for Java-based enterprise systems. JSR 88 itself was already discussed in section 3.4.1.

XML Processing For the processing of deployment descriptors the *Java Architecture for XML Binding 2.0* (JAXB) [154] is used. It allows, amongst others, the construction of specific APIs out of *XML Schema* definitions [22, 153]. The generated classes allow the creation, inspection, and manipulation of XML files based on schema-specific objects from inside Java source code. In this context, APIs were generated for the schemata of EJB deployment descriptors and for *mKernel*-specific descriptors. The generated classes for both schemata are used during preprocessing for inspecting the original EJB DD, and for the construction of the comprehensive EJB DD and the creation of the *mKernel*-specific DD. Furthermore, the APIs are also used inside the container plugin for obtaining information about a submitted ejb-jar file. Finally, the API for EJB DDs is used inside the *Container Plugin* when adjusting the descriptor during module deployment. JAXB was released as JSR 222 and became part of the *Java Platform, Standard Edition*, version 6.0.

Byte Code Inspection and Manipulation The inspection and manipulation of class files, as well as the construction of new classes and interfaces are performed with the aid of the *Java Programming Assistant* (Javassist) [5,48,49]. *Javassist* provides, amongst others, an API for Java byte code inspection and manipulation based on Java language constructs. This enables, for example, the creation of methods for classes based on submitted Java source code or the construction of completely new classes. Additionally, facilities for in depth access to the underlying byte code are also provided. In combination, the *Javassist* API represents an easy to use foundation for byte code engineering which provides all opportunities needed in the context of *mKernel*. *Javassist* is used by different processing modules during preprocessing of ejb-jar files. The tool would also allow the dynamic generation and instantiation of classes at runtime and would therefore theoretically be usable within deployed modules such as the dynamic generation of proxies. For the realization of *mKernel* this was no alternative to the applied static approach, because it would violate the EJB standard. The development of *Javassist* started as independent project. Meanwhile, it became a sub-project of the *JBoss Application Server* [6].

Source Code Generation Finally, the generation of Java source code is based on the *Velocity Engine* of the *Apache Velocity Project* [10]. The *Velocity Engine* supports the processing of templates from inside Java source code. The corresponding template language allows the definition of templates which might, amongst others, refer to submitted Java objects and to iterate over Java-based collections. Furthermore, the opportunity to define branch definitions allows the conditional integration of template sections into the output. In the context of preprocessing, templates build the foundation for the generation of Java source code which is used as input for byte code manipulation performed with the aid of *Javassist*.

The tool application does not induce any violations of the EJB standard. *Javassist* and the *Velocity Engine* are applied during preprocessing. In this context, standard compliant ejb-jar files are constructed, and no elements of the tools become part of these files. Regarding *Javassist* it would theoretically be possible that the tool internally generates byte code which makes use of language constructs such as thread synchronization which would represent a violation of the EJB standard. To guarantee that this is not the case, an in depth analysis of the tool source code would be necessary. This was not performed during the development of *mKernel*. Instead of that, decompilations and analyses of many examples of generated and manipulated class files were performed. None of these files showed invalid source code constructs. Therefore, the application of *Javassist* was assumed of leading to valid byte code for the integration into ejb-jar files. The API of JSR 88 is explicitly designed for the management of Java-based enterprise systems. Its usage from inside an EJB container is not forbidden by the EJB standard. Finally, the usage of JAXB during preprocessing does not even induce any changes of byte code. Furthermore, the generated XML files are valid with respect to the underlying schemata. For the application of JAXB inside a managed container no file access or other invalid operations are necessary. Therefore, the usage of JAXB does also not induce any violations of the EJB standard.

6.4. Summary

Within this chapter an overview of the internals of *mKernel*-based system management was presented. The discussion did not aim to provide an insight into implementation details, but focused on the presentation of general concepts, design decisions, and applied techniques.

Section 6.1 first provided an overview of the main parts of the management infrastructure, as well as their tasks and relationships for realizing the management of EJB-based enterprise systems. Afterwards, the different parts were discussed separately with respect to their realization. In this context, an insight into the internal structure of each part was given highlighting how the different tasks are provided through the constituent elements. This also covered the presentation of relevant processes regarding interactions for reaching the desired behavior in case these were necessary for the understanding of design decisions.

Within section 6.2 the preprocessing of ejb-jar files was presented. The discussion started with an overview of the architecture of the preprocessing tool being part of *mKernel*. In this context, so-called *Processing Module*s were introduced which contribute to the construction of manageable archives through the provision of core functionalities needed for preprocessing. The subsequent discussion of the proceeding for manageable archive generation was divided according to the layer for which the particular archive is constructed. The discussion did not go into detail for each of the applied modules. In contrast, it focused on the presentation of groups of modules which in combination fulfill a certain part of the overall tasks during preprocessing. In this context, the differences regarding the construction of *Access Layer* ejb-jar files and *Managed Layer* ejb-jar files were highlighted. Furthermore, alternatives for the construction of *Access Layer* archives and their implications for system management were discussed.

Section 6.3 provided an overview of the applied tools which were used by different parts of *mKernel*. In this context, the intended usage area of each tool was presented and its particular application in the context of *mKernel* was discussed. Finally, it was argued why the application of each of the tools does not induce any violation of the EJB standard.

In section 6.1.2 two violations of the EJB standard were pointed out, namely the use of `static` fields, and the application of object substitution during SB proxy serialization and deserialization. For both of these violations alternative realizations of the corresponding concepts were presented, and it was argued why a violation of the standard was accepted for the realization of *mKernel*. In this context, it must be highlighted that the decisions for standard violation were based on detailed analyses and investigations regarding alternatives. Nevertheless, the replacement of the realized solutions with standard compliant substitutes would be possible with low expenditures.

7. Application

The previous chapters presented the approach for supporting autonomic management of enterprise systems based on the EJB standard. Chapter 5 discussed the externally provided sensors and effectors for autonomic entities which are organized in a meta model and are realized through a corresponding API. Afterwards, chapter 6 provided an insight into the realization of *mKernel* with respect to the overall architecture and the constituent elements. In this context, conceptual aspects were presented and the underlying design decisions were discussed. In combination, the chapters provided the black-box-view and white-box-view of the AC infrastructure.

While the previous chapters considered the realization of the thesis subject directly, this chapter discusses the application of *mKernel* for two projects. In this context, the AC infrastructure was used as foundation for addressing certain objectives of AC. The goal of the first project was to support *Self-Configuration* through the realization of a stepwise approach for defining complex reconfiguration strategies. These can be reused for different reconfiguration scenarios. The second project aimed to support *Self-Protection* and *Self-Healing* through applying the concept of *Design by Contract* [116] for supervising the behavior of system elements at runtime. In this context, erroneous or malicious behavior can be discovered, and immediate counteractions can be defined. Furthermore, enhanced analyses of relevant incidents are supported.

The remainder of this chapter is structured as follows: Section 7.1 presents the project for addressing *Self-Configuration* through reconfiguration strategies. Afterwards, section 7.2 discusses the approach for supporting *Self-Protection* and *Self-Healing* through application of the *Design by Contract* concept. Finally, section 7.3 summarizes the project results and evaluates the application of *mKernel* as foundation for the two projects. The presentation of the two projects does not go into details regarding their realization, but only provides conceptual overviews.

7.1. Support for Self-Configuration

The goal of the project presented in this section was to design and implement a framework which supports the definition, instantiation, and execution of reconfiguration strategies on top of the *mKernel API*. In this context, the project concentrates on dynamic adaptation regarding the replacement of an existing module with a new one. The project was realized during a diploma thesis by *Vogel* (cf. [156]) and was supervised by the author of this thesis. The results of the project were published in the *Proceedings of the International Conference on Software Engineering and Knowledge Engineering 2008 (SEKE'08)* (cf. [157]) and presented in a talk at the *SEKE'08* conference.

In the remainder of this section the project is discussed as follows: The motivation of the project is presented in section 7.1.1. Afterwards, section 7.1.2 provides a sketch over the design and realization of the approach[37].

7.1.1. Motivation

In section 1.2.2 the life cycle of enterprise systems was introduced. In this context, the *Management Phase* represents that phase of a life cycle during which a system is productively used. Over time, the reconfiguration of a system might become necessary due to various reasons such as changes in the environment which demand for the integration of new functionali-

[37] All details have been explicitly left out here. Please refer to the conference paper [157] and the diploma thesis [156] for further details.

ties, the removal of deprecated elements, or the need to replace erroneous ones. These adaptations might – depending on the concrete scenario – become highly complex, error-prone, and time consuming tasks, for example, due to the number and complexity of affected elements, the corresponding configuration demands, and external requirements to fulfill.

The *Self-Configuration* objective of autonomic entities addresses their capabilities to perform parameter and compositional adaptation autonomically based on user requirements (see section 2.1.1.1). Therefore, they should ideally be able to deduce the necessary actions without further interaction demands and finally execute them autonomically. Consequently, administrators should be enabled to concentrate on the specification of goals and should be disburdened from realization details. An exemplary goal might be the demand to replace an existing deployed component with a new revision of the corresponding component.

For the planning and execution of self-configuration there do exist certain activities which might recur in different reconfiguration scenarios. Examples of these activities are the deployment and configuration of a replacing module, or the rerouting of incoming connections from a module to replace to the replacing one. All these activities are supported through the *mKernel API* in a model based way. Nevertheless, their usage within the implementation of different autonomic entities might require the repeated writing of very similar source code. Therefore, it was considered meaningful to develop a framework on top of *mKernel* which supports the development, integration, and application of reusable building blocks for reconfiguration planning and execution. Furthermore, the assembling and realization of comprehensive reconfiguration procedures as a whole should also be supported. The last aspect was inspired by the work of *Rosa et al.* who identified three so called *Reconfiguration Strategies* and applied them in their system for the composition of message oriented services, called *RAppia* [129].

7.1.2. Overview

The framework realized as result of this project supports reconfiguration planning and execution in a stepwise approach on top of the *mKernel API*. Figure 7.1 depicts the corresponding concepts and their relationships.

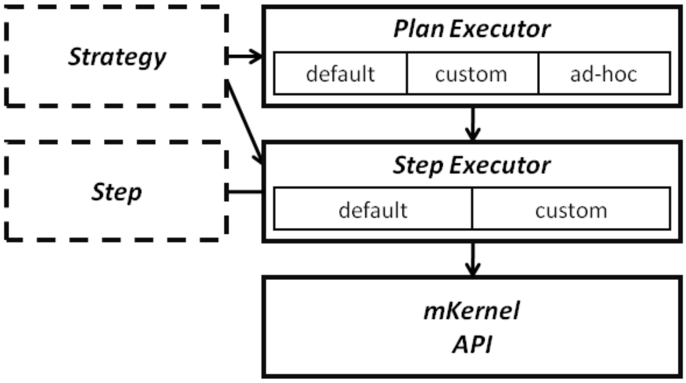


Figure 7.1.: Reconfiguration Framework Concepts

As foundation the *mKernel API* provides the infrastructure for the realized framework. On the next higher level so called *Step Executors* encapsulate delimited functionalities which can be integrated into complex reconfiguration procedures. To allow a structuring of executors, the concept of *Steps* establishes a classification schema which supports the grouping of *Step Executors* according to the particular task they could fulfill during reconfiguration execution. This classification should facilitate the identification of *Step Executors* which might be able to fulfill similar tasks during reconfiguration and to select an appropriate executor for the particular needs. As top-level concept so-called *Strategies* represent comprehensive procedures which define templates for performing an entire reconfiguration within a managed system. These *Strategies* might be instantiated into *Plans* which are not depicted explicitly in the figure. After their configuration these plans can be executed with the aid of so called *Plan Executors* which internally make use of the *Step Executors* referenced within the corresponding *Strategy*.

The two elements on the left hand side of figure 7.1 on page 272 – *Strategy* and *Step* – do not provide any functionalities during reconfiguration execution. Therefore, they are depicted with dashed lines. For the other two elements of the framework – *Plan Executor* and *Step Executor* – different realizations are conceivable which might be applied in different reconfiguration scenarios. As part of the framework, for both of them *default* implementations are provided as proof-of-concept. Additionally, the framework allows the integration and application of *custom* realizations, both for *Plan Executors* and *Step Executors*. Finally, on top level it is also possible to make use of an *ad-hoc Plan Executor*. Such an executor does not require the existence of a corresponding *Strategy*, but might be configured directly.

In the following, the core elements of the framework are presented bottom-up. Therefore, section 7.1.2.1 starts with a short overview of the identified groups of *Steps* and afterwards discusses the requirements for making use of *Step Executors*. In section 7.1.2.2 *Strategies* and *Plan Executors* are presented with respect to the construction of plans and their application to concrete reconfiguration scenarios. Finally, section 7.1.2.3 provides a short overview of the framework realization.

7.1.2.1. Steps

The types of tasks to perform during a reconfiguration are classified through a schema which specifies *Steps* with distinguishable objectives. In this context, a *Step* defines *What* has to be done while the corresponding executors provide concrete realizations. Consequently, the choice of a concrete *Step Executor* determines *How* the objective should be reached. Within his work *Vogel* organized the identified *Steps* into seven groups according to the major topic the constituent *Steps* address. The following list gives an overview of these groups while a detailed discussion of the contained *Steps* is left out here. Please refer to the work of *Vogel* for further details (cf. [156, 157]).

1. **Quiescence Management**: This group addresses the life cycle of quiescence regions. In this context, different *Steps* are defined to induce state transitions, as well as for waiting for a region to reach the `QUIESCENT` state.
2. **Module Life Cycle Management**: Analog to the management of quiescence regions, *Steps* for the management of module life cycles are subsumed within this group.
3. **Compositional Adaptation**: The tasks of *Steps* within this group relate to the execution of compositional adaptation. An example of a contained *Step* would be the rerouting of existing connections to SBs of a replacing module.
4. **Parameter Adaptation**: This group covers *Step*s for all aspects of parameter adaptation such as the setting of SEE values or the specification of security and transaction aspects.
5. **State Transfer of stateful SB Instances**: The transfer of conversational states of stateful SB instances is addressed by *Steps* of this group. This might cover, for instance, the extraction, transformation, and injection of state elements.
6. **Database Management**: *Steps* of this group consider the need for transfer or manipulation of underlying databases during reconfiguration.
7. **Supporting Steps**: Finally, this group contains all *Steps* which define supporting tasks for system reconfiguration. An example would be the integration of a delay which could be necessary between the activation of tracking for a quiescence region and the transfer of the region to the `BLOCKING` state.

The groups of *Steps* listed above address all aspects of system reconfiguration provided by *mKernel*. Nevertheless, the complexity of tasks assigned to *Steps* varies between groups as well as inside groups. While certain groups more or less directly reflect manipulation opportunities exposed by the *mKernel API*[38], others do cover more complex tasks[39]. The varying granularities are considered necessary to reach a high degree of flexibility regarding the construction of strategies. In this context, it would, for instance, be possible to interleave the execution of *Step Executors* for the management of a quiescence region with the execution of those for managing the life cycle of a new module to minimize system disruption. Furthermore, it would be possible to exclude certain *Steps* for special purpose scenarios such as situations where an already existing module should be used to replace another one. For such a situation it would not be necessary to apply executors for all *Steps* of module life cycle management. In this context, it is not required that all *Steps* are integrated into a *Strategy* through corresponding executors. Instead of that, *Steps* represent opportunities instead of requirements. Moreover, it is also possible to apply more than one executor associated with the same *Step*, for example, if more than one module should be integrated into a system.

As mentioned earlier, each *Step* might be realized through an arbitrary number of *Step Executors*. Depending on their concrete realization, *Step Executors* might require to be configured for their application within a concrete reconfiguration situation. An example of such a configuration demand would be the need to submit the unique identifier of an `EjbModule` which should be destroyed. Additionally, executors might provide outputs as results of execution such as the module identifier of an `EjbModule` which was newly created. Consequently, *Step Executors* are characterized

[38] Examples would be the *Steps* for state transfers of a quiescence region.

[39] Such a task would be the transformation of the conversational state of a stateful SB instance.

by their assignment to a certain *Step*, as well as by input and output *Parameters*.

7.1.2.2. Strategies

Strategies define templates for the execution of complete reconfiguration procedures. They cover a collection of *Step Executors* which are intended to fulfill partial tasks during reconfiguration. Additionally, each *Strategy* can define input parameters which can be used to configure the *Strategy* with respect to a concrete application context. A *Strategy* definition requires mappings for all input parameters of the constituent *Step Executors*. Such a mapping might be provided through a connection to an output parameter of another *Step Executor* or to one of the input parameters of the *Strategy*. Additionally, it is possible to define output parameters for returning the results of *Plan* execution. Each of these outputs must be connected to an output parameter of one of the *Step Executors*. Finally, each *Strategy* must refer to a *Plan Executor* which should be used for realizing the *Strategy* in a concrete application context.

Plan Executors are envisioned as generic elements which might be reused for different *Strategies*. In this context, the framework supports the integration and usage of different implementations for the execution of plans. This allows the application of *Plan Executors* which are capable to cope with specific needs of *Strategies* and *Step Executors*. In contrast, it is considered impossible to provide one single *Plan Executor* as inherent part of the framework which could be applied to all conceivable reconfiguration scenarios. An elementary *Step Executor* would, for instance, accept a *Plan* to execute and would address the constituent *Step Executors* according to the order they are contained within the *Plan*. In this context, the *Plan Executors* would receive and forward the different parameter values according to the parameter mappings defined within the corresponding *Strategy*. An alternative *Plan Executor* would, for example, be able to analyze a *Plan* against certain dependencies which might exist between the *Step Executor* according to the corresponding *Steps*. These do not necessarily need to be reflected by the parameter mappings. One example would be that a *Step Executor* which is responsible for the starting of a new module must be executed after an executor which is responsible for module deployment. In this case there might not exist a connection between the *Step Executors*, because both of them receive the needed module identifier from the output of the executor responsible for module creation. Moreover, there are *Plan Executors* conceivable which are optimized for a subset of reconfiguration scenarios or even for one single scenario.

7.1.2.3. Realization

The framework supports the implementation of *Step Executors* and *Plan Executors* and their integration into a managed system. Additionally, it allows the definition of *Strategies*, as well as the instantiation and execution of *Plans*. To provide an infrastructure for reconfiguration planning and execution, a specific EJB module must be integrated into the managed EJB container. Figure 7.2 depicts the core elements of the framework inside a managed system, as well as their relationships.

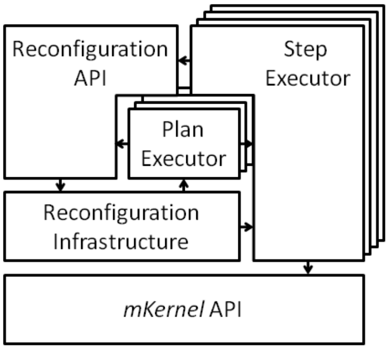


Figure 7.2.: Reconfiguration Framework Realization Overview

Step Executors and ***Plan Executors*** represent the core extension points of the developed framework, because they allow the integration of new functionalities. For the development of new executors the *Reconfiguration API* defines two special interfaces. Each of these is specific to one executor type and must be implemented by corresponding realizations. Additionally, certain annotations are defined as part of the API which can, for instance, be used to specify input and output parameters for *Step Executors*.

The ***Reconfiguration API*** provides the foundation of all aspects of the framework which are not directly concerned with the original reconfiguration. In particular it determines how executors must be realized through the definition of annotation types and interfaces. After finishing the development of executors, these must be packed into a so called *executor-jar file* which must be submitted to the API. The API creates a framework specific deployment descriptor from the information contained within the executor byte code and the included annotations. Furthermore, each executor is extended through byte code manipulation for being usable as session bean. It is not necessary that all *Step Executors* which are used during a reconfiguration do reside within the same archive as the *Plan Executor* which controls the reconfiguration. In fact, the framework does not state any requirements regarding the distribution of executors upon executor-jar files. *Strategies* can also be defined and manipulated through the API. Additionally, sensors and effectors for inspection and manipulation of the sets of available executors and *Strategies* are provided. Furthermore, executor modules can be integrated into and removed from a managed system through the API. For the planning of a reconfiguration the API enables the instantiation of a *Plan*, its configuration, and its submission to a *Plan Executor* instance. Finally, *Plan Executor* instances do control their associated *Step Executor* instances through corresponding API elements. To allow the application of the API from inside EJB containers, it is divided into two libraries. In this context, the processing of executor-jar files is sourced out into a separate archive, because it is used for byte code manipulation which is forbidden by the EJB standard. Nevertheless, all other aspects of the API can be used from inside a managed container.

Implementations of *Step Executors* might make use of the ***mKernel API*** for realizing the required functionality of the corresponding *Step*. In this context, the API provides the foundation for all activities directly related to the inspection and manipulation of a managed system. Nevertheless, *Step Executors* do not necessarily need to make use of *mKernel*. An example would be an executor which realizes a delay between the transfer of a quiescence region from the `TRACKING` to the `BLOCKING` state.

To establish the basis for framework application within a managed system, the ***Reconfiguration Infrastructure*** must be deployed into the target container. It is able to receive executor-jar files through the *Reconfiguration API*, extract the included information and create a corresponding representation. Based on these archives, the infrastructure allows managing entities to inspect the set of available executors, deploy corresponding modules, and manage their life cycles. In this context, the infrastructure also generates the necessary EJB DD during deployment. Finally, these modules can be destroyed and ejb-jar files can be removed from the system when they are not needed anymore. Regarding *Strategies*, the infrastructure provides sensors and effectors for their inspection and manipulation.

Strategies are not depicted within figure 7.2 on page 277. They are managed inside the *Reconfiguration Infrastructure* and are exposed through corresponding API elements. In order to provide a proof-of-concept, the three reconfiguration strategies identified by *Rosa et al.* are taken as foundation and are adjusted for the underlying application area, namely EJB. These strategies are in particular (cf. [129, 156]):

- **Flash**: This strategy does not consider the existence of established connections on *Instance Level*. In this context, the replaced module is stopped before the establishment of connections to the replacing one. Existing connections on *Deployment Level* are rerouted lazily. A temporal system disruption would be the result of strategy application, and existing *Instance Level* references would get lost. This strategy might, for example, be applied if the system is only used during business hours, and it can be ensured that no interactions are active during reconfiguration.

- **Interrupt**: For the application of this *Strategy* the affected modules are used as foundation for a quiescence region. After reaching quiescence, conversational states of stateful SB instances are extracted from the replaced module and injected – after a potential transformation – into corresponding instances of the replacing module. Afterwards, the replaced module is stopped and connections on *Deployment Level* are rerouted. Finally, the quiescence region is released. This strategy requires that a state transfer on *Instance Level* is possible, that is, a corresponding transformation is feasible. The strategy is intended to be used if the underlying data sources are not usable concurrently by the replaced an the replacing module.
- **Non-Interrupt**: On application of this *Strategy* connections on *Instance Level* are not rerouted while connections on *Deployment Level* are rerouted lazily. Consequently, existing references to the replaced module are used as long as they are needed while newly established ones refer to the replacing module. For the application of this *Strategy* it is necessary that both affected modules can be used concurrently.

Furthermore, a fourth strategy has been identified which might be interpreted as a hybrid form, covering aspects of the *Non-Interrupt* and the *Interrupt* strategy.

- **Non-Interrupt/Interrupt**: This *Strategy* might be applied if both modules might be used concurrently, but existing connections on *Instance Level* should be rerouted also. In such a scenario connections on *Deployment Level* are rerouted *lazily* before a quiescence region is defined. Afterwards, the replaced module is transferred to a quiescent state, and existing references on *Instance Level* are rerouted analog to the *Interrupt Strategy*. Consequently, no delay would be noticed for newly established connections, because they would not have to wait for the transfer of conversational states[40]. This strategy is meaningful if applied to scenarios where connections to stateful SB instances are used over a long timespan, and the reconfiguration should be finished in a timely manner.

For each of these strategies corresponding representations were created through the *Reconfiguration API*. Furthermore, reference implementations for all needed *Step Executors*, as well as an implementation of a *Plan Executor* are provided. They were tested based on a sample application. In this context, most of the *Step Executors* could be reused for more than one *Strategy*[41]

7.2. Self-Protection and Self-Healing based on Contracts

In this section a project is described which aims to address the self-healing and self-protection objectives of autonomic entities through application of the *Design by Contract* (DbC) concept. The project was realized as a full-time university practical during the semester break in summer 2007. It represented the final part of the course specialization *Practical Computer Science* at the *Distributed and Mobile Systems Group of the University of Bamberg*. The group of participants consisted of seven students who were in their main study period. The results of the project were published in the *Communications of SIWN, volume 4, June 2008* [35] and presented in a talk at the *4th International Conference on Self-organization and Adaptation of Computing and Communications (SACC 2008)*.

The remainder of this section is structured as follows. Section 7.2.1 provides a motivation of the project and discusses the relation to AC. Af-

[40] Besides potential delays due to the performance overhead for reconfiguration execution.

[41] For further details please refer to [157].

terwards, section 7.2.2 presents an overview of the project realization.

7.2.1. Motivation

The concept of DbC was developed by *Meyer* [116] and addresses relationships between providers and clients of functionalities. These relationships can be specified through so called *Contracts* consisting of *Precondition*s, *Postcondition*s, and *Invariant*s. Preconditions define *Obligation*s for clients of a functionality, that is, before usage they are responsible to ensure that the corresponding preconditions hold. In return the provider is – if the client adheres to its obligations – responsible to ensure that the usage leads to the fulfillment of the postconditions. The other way round, preconditions are *Benefit*s for the providers which can rely on their fulfillment while clients can benefit from the postconditions if using the corresponding functionality with holding preconditions. Invariants specify conditions which must hold in any observable state of the functionality provider. Consequently, the provider is responsible to ensure their fulfillment after its creation. Additionally, it has to ensure that each interaction with holding preconditions leads to a subsequent state guaranteeing invariants fulfillment. If it does so, its implementation can be assumed to be *correct*. If one of the conditions is violated at runtime, this is interpreted as *Fault* inside the system. A fault indicates that at least one part of the system contains a *Defect* inside its implementation leading to a deviation from the intended behavior. The identification of a defect depends on the type of the violated condition. If a precondition is violated, the client of the functionality can be assumed of containing the defect, because it does not follow its obligations. A violation of a postcondition or an invariant – if no precondition violation happened before – might indicate a defect inside the implementation of the provider. Nevertheless, it might also be possible that the provider itself made use of other functionality providers. If at least one of those did not perform its tasks correctly, this might also have led to an identified violation.

The *Self-Protection* and *Self-Healing* objectives of autonomic entities were already discussed in section 2.1.1.1. A self-protecting entity should be able to detect, identify, and defend against internal and external threats which might arise, amongst others, from malicious or erroneous interactions. In this context, an entity should react appropriately, for instance, through preventing these interactions from reaching their target or through confining the effects of harmful interactions. Self-healing entities should diagnose and recover from inconsistencies, malfunctions, or failures detected in a managed system to preserve or reestablish integrity and availability. As foundation for both objectives autonomic entities must be able to identify undesired behavior of system elements or external clients.

The approach presented in this section relies on the idea to apply the concept of DbC for the identification of undesired behavior. In this context, contracts are assumed of providing a reasonable foundation for defining correct behavior. At runtime the observation of contract adherence allows the identification of deviations from this behavior. Moreover, if such a deviation is identified, it would be possible to blame the responsible interaction partner based on the type of violated condition. Therefore, the goal of the project was to realize a foundation for the specification of contracts as part of interfaces and to enable the supervision of contract adherence. Additionally, immediate reactions to contract violations should also be supported.

7.2.2. Overview

The approach is based on the operationalization of contracts for supervising the runtime behavior of bean instances and their clients. In this context, the project concentrates on provided functionalities of SBs, because these are specified through Java interfaces which define syntactic aspects of interactions. These aspects might be used inside contract specifications, for example, to define the range of valid parameter values within preconditions. For the approach it was decided not to assign contracts to SB classes but to interfaces, because these are independent from concrete implementations and are intended to provide the foundation for interactions between arbitrary providers and clients of the corresponding functionalities. Furthermore, it was considered meaningful to directly integrate contracts into affected interfaces instead of provider classes. Within the Java programming language there do not exist any language constructs which directly support the concept of DbC for interfaces. Therefore, it was necessary to develop a contract language which supports the definition of contracts and provides a foundation for the integration of contracts into interfaces.

In order to make use of the proposed approach, autonomic entities must be provided with sensors to inspect existing contracts and to track contract violations. Additionally, they must be equipped with effectors to configure the system behavior, for example, regarding violation tracking or countermeasures to perform in response to violation situations. Analog to *mKernel*, the required sensors and effectors are exposed through a corresponding API.

The validation of contract adherence at runtime requires the interception of method invocations upon affected SB instances. In case of a contract violation information is stored for later analyses. Furthermore, direct countermeasures are supported which allow autonomic entities to immediately react to undesired behavior. All of these facilities are realized by a specific infrastructure within the managed container which internally makes use of the functionalities provided by *mKernel*.

The remainder of this section discusses the three core elements of the approach. Section 7.2.2.1 starts with a presentation of the contract language which was developed as part of the project. In this context, the operationalization of contracts is also addressed, including deployment preparations. Afterwards, section 7.2.2.2 presents the API which might be used by autonomic entities for inspection and manipulation purposes. Finally, section 7.2.2.3 provides an overview of the infrastructure inside a managed container which is needed for applying the approach.

7.2.2.1. Development and Deployment Preparation

Before the original presentation of the contract language it must be highlighted that the language itself was not a major goal of the project. Therefore, it contains only basic features for the inspection of parameter and return values, as well as opportunities to interact with the invocation target, namely an SB instance. Nevertheless, the language is designed extensible and might consequently also support more advanced features.

The contract language can be used to specify pre- and postconditions on method level, as well as invariants on interface level. Each condition is represented as a boolean expression. If evaluated to `true`, the condition is fulfilled while an evaluation to `false` indicates a contract violation. The specification of a condition can consist of boolean operations[42] and comparisons[43]. Additionally, a test for *null*-references is also supported. Internally, pre- and postconditions may refer to method parameters. The return value of a method call can also be referred to in postconditions. An `OLD` operator is included which allows to make use of the value of a certain expression evaluated before method execution in postconditions. Boolean, number, and string constants are also supported. A `COMBINE` operation is integrated to iterate over collections and arrays, and to derive a single value[44] as result which is usable within other expressions. The usage of the basic constructs guarantees termination of evaluation and side effect freeness. Method invocations are also supported as parts of expres-

42 `and`, `or`, `xor`,`not`.

43 $<, <=, ==, != , >=, >$

44 boolean, number, or string

sions, for example, to observe the state of a session bean instance as part of an invariant. Through this opportunity extensibility of the contract language is given. In contrast to the basic elements, for the use of method calls care must be taken to keep evaluation termination and side effect freeness.

Contracts can be declared for interfaces and the included methods through metadata annotations and thus become an integral part of them. Therefore, the three annotation types `Preconditions`, `Postconditions`, and `Invariants` are provided. Each of them holds a set of the corresponding conditions which contain contract language declarations. Furthermore, human readable documentations can be integrated. Although it would have been possible to integrate all conditions of a certain type for a particular target into one single condition, a differentiation allows more specific analyses regarding violations and reaction control at runtime.

Listing 7.1 on page 287 contains a Java interface which is extended with a contract specification. The interface `BankAccount` was taken from the example discussed in [35]. It represents an access point to a bank account. For such an account the covered balance must always be greater or equal to zero. This is specified through the invariant `BalanceGreaterEqual-Zero`. Note that an invocation of `getBalance` will be performed on the implementation of the interface during contract evaluation. The method `withdraw` can be used to withdraw the submitted amount from the account. To use the method correctly, a user must fulfill two preconditions:

- `BalanceSufficient`: The balance of the account must be high enough to process the withdrawal, that is, the operation must not lead to a negative balance.
- `ValidAmount`: Only positive amounts are valid for withdrawal.

Inside pre- and postconditions terms starting with '`$`' followed by a non-negative integer refer to a method parameter. The integer represents the position inside the parameter list, starting with zero.

```
@Invariants({
 @Invariant(
  name = "BalanceGreaterEqualZero",
  description =
    "Balance must be greater or equal '0'.",
  condition = "getBalance() >= 0;"
 )
})
public interface BankAccount{

 @Preconditions(conditions={
  @Precondition(
   name="BalanceSufficient",
   description=
     "Amount to withdraw must not exceed balance.",
   condition="$0 <= getBalance();"
  ),
  @Precondition(
   name="ValidAmount",
   description=
     "Amount to withdraw must be greater '0'.",
   condition="$0 > 0;"
  )
 })
 @Postconditions({
  @Postcondition(
   name="CorrectProcessing",
   description=
     "New balance must be correctly calculated.",
   condition=
     "OLD(getBalance()) - $0 == getBalance();"
  )
 })
 public void withdraw(long amount);

 public long getBalance();

}
```

Listing 7.1: Contract Example (cf. [35].)

After processing, a provider of the interface must ensure adherence to the postcondition `CorrectWithdrawalProcessing`, that is, it must guarantee that the balance is reduced by the submitted amount. Here, the `OLD` construct is used to refer to the balance before method execution. For a detailed discussion about implications regarding concurrency, as well as violation analyses and countermeasures, please refer to the corresponding article (cf. [35]).

As preparation for deployment, a module must be preprocessed by the *Contract Processor*. This command line tool identifies all provided remote business interfaces within a submitted ejb-jar file and generates an individual contract interceptor for each session bean included, if contracts are identified. This interceptor is capable to evaluate all relevant conditions on method invocation. On violation occurrence it also performs the corresponding reaction and publishes tracking information, if desired. The preprocessor was developed with the aid of the *Java Compiler Compiler* (JavaCC) [2] which was used to generate a Java-based parser for the contract language.

7.2.2.2. Contract API

The *Contract API* provides sensors and effectors for autonomic entities which are specific for the presented approach. In this context, the API can be seen as extension of the *mKernel API* discussed in chapter 5. Figure 7.3 on page 289 depicts the central concepts of the *Contract API* and their relationships to elements of the *mKernel API*.

The figure is divided into four areas which are delimited from each other through dashed lines. The horizontal line separates the *Type Level* in the upper part of the figure from the *Instance Level* in the lower part. The *Contract API* on the left hand side of the figure is separated from the *mKernel API* on the right hand side of the figure through the vertical line.

`Conditions` are used to represent the different elements of contracts, that is, preconditions, postconditions, and invariants. Depending on the particular type, conditions are either associated with a `JavaInterfaceType`

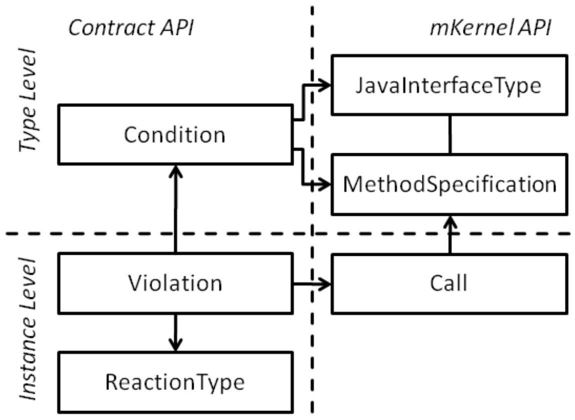


Figure 7.3.: Central Concepts of the *Contract API*

or a `MethodSpecification`, both belonging to the *mKernel API*[45]. On *Instance Level* an observed contract violation is represented through an instance of `Violation`. Each violation relates to exactly one corresponding `Condition` which has been violated. Furthermore, each violation occurred in the context of a concrete invocation which is represented as `Call` by the *mKernel API*. In this context, pre- and postcondition violations are only associated with `BusinessCalls` while invariant violations might also occur in the context of `LifecycleCalls`. Moreover, it is possible that there do exist more than one `Violation` for a single `Call`. Finally, each `Violation` is associated with a `ReactionType`[46] which represents the performed countermeasure. There are three basic reaction types supported by the approach, namely:

[45] Invariants are always associated with `JavaInterfaceTypes` while pre- and postconditions are associated with `MethodSpecifications`. The specialization was left out in the figure for reasons of simplification.

[46] `ReactionType` is realized as enum, this is indicated through the suffix `Type`. Nevertheless, `ReactionType` belongs to the *Instance Level* and not to the *Type Level*

1. `EXCEPTION`: If this `ReactionType` was performed, an exception is thrown in response to the `Violation`. This also implies the initiation of a transaction rollback, if possible.
2. `ROLLBACK`: This type indicates that a transaction rollback was initiated for confining the `Violation` effects. The concrete effects of applying this `ReactionType` vary according to the transaction demarcation type specified for the corresponding method.
3. `PROCEED`: For a contract violation identified before method execution this `ReactionType` indicates that the invocation was forwarded to the original target. If applied after execution, the corresponding method returned without any intervention.

In addition to the three types, a fourth type `FORWARD_TO_HANDLER` is also supported by the approach. If applied to a `Violation`, it indicates that a so-called *Violation Handler* has been contacted in response to a `Violation`. The handler concept represents an extension point to allow autonomic entities to intercept the call flow in case a `Violation` is identified. *Violation Handlers* are discussed in the following section.

Analog to the `ContainerFactory` class and the `Container` interface of the *mKernel API* there do exist a class `ContractSupervisorFactory` and an interface `ContractSupervisor` in the *Contract API*. These represent entry points to the approach. They are not depicted in figure 7.3 on page 289. A reference to a `ContractSupervisor` can be used to request `Violations` on different levels, that is, for a certain invocation, an SB instance, an SB, or a module as a whole. Furthermore, it is also possible to obtain all `Violations` which relate to a submitted `Condition`. Navigation is provided in both directions, from elements of the *Contract API* to elements of the *mKernel API* and vice versa. Consequently, the *Contract API* can be seen as an extension of the *mKernel API*.

Regarding analyses of contract violations, all opportunities provided by the *Instance Level* of the *mKernel* API might be used, for instance, for the inspection of call chains and call histories. The proceeding for the identification of relevant incidents might be the same as the one described in section 5.4.5. In this context, the concept of DbC provides additional, valuable information which allows, for instance, to identify the interaction partner responsible for a contract violation. Moreover, the concretely violated condition might give further information about the underlying defect. If a managed bean instance is responsible for a violation, it is possible to determine the `EnterpriseBean` on *Deployment Level* which contains the corresponding defect. With this bean as origin, other system elements can be identified through navigation along connections which might also be affected by the defect directly or indirectly. On *Type Level* the `EnterpriseBeanType` containing the causal error can be determined. This might also help to find other `EnterpriseBeanTypes` which rely on the same implementation. Based on this information, a navigation back to the *Deployment Level* would possibly reveal other `EnterpriseBeans` which contain the same defect, but did not yet cause a corresponding violation. In combination, this information might provide a helpful foundation for further analyses, as well as for planning a system reconfiguration.

The *Contract API* allows autonomic entities to activate information tracking regarding contract violations on different levels, similar to the *mKernel API*, as discussed in section 5.4.3. Additionally, it is possible to activate logging for single `Conditions`. The API internally ensures that *mKernel*-based logging is also activated for affected parts of a system. Furthermore, countermeasures for contract violations can be defined on *Deployment Level*, for example, for a certain SB or a module as a whole. In this context, reaction definitions can be based on `Conditions`, as well as on combinations of `Conditions` and *Deployment Level* elements of the *mKernel API*. The `ReactionTypes` discussed above, are used as foundation for the definition of reactions. Finally, the API allows the selective removal of violation information.

7.2.2.3. Runtime Infrastructure

After the extension of an ejb-jar file the archive must be preprocessed for its application with *mKernel* and finally be integrated into a managed system. Figure 7.4 depicts those elements of a managed system which are directly concerned with the application of the approach and their relationships. The figure does not cover all aspects of *mKernel* in detail, because these were already discussed in chapter 6. The gray shaded elements

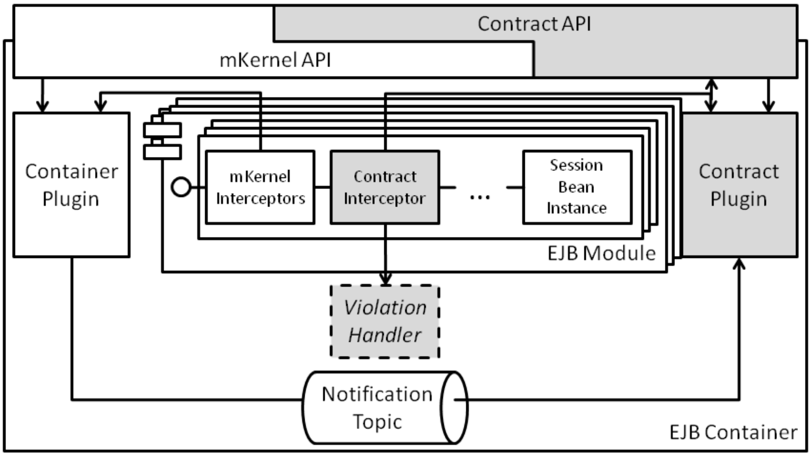


Figure 7.4.: Reconfiguration Framework Concepts

within the figure are specific to the presented approach. The *Contract Plugin* is used as infrastructure within a managed container. Sensors and effectors are provided through a corresponding *Contract API*, as discussed in the previous section. The *Contract Interceptor* was already introduced shortly in section 7.2.2.1. It is generated from identified interfaces and is afterwards integrated into the ejb-jar file during preprocessing. Finally, *Violation Handlers* represent extension points of the approach with respect to immediate reactions to contract violations.

In order to apply the approach to a managed system, the ***Contract Plugin*** must be deployed into the target container. This is supported by a command line script, similar to the proceeding described in section 6.1.1 for the *Container Plugin* of *mKernel*. After script execution all necessary resources do exist, and the plugin is ready for usage. For the integration of new ejb-jar files which are affected by the approach it is necessary that the corresponding contract information is submitted to the plugin. This can be performed through a tool which is provided as part of the project. It internally makes use of the *mKernel API* for archive integration and afterwards submits the approach specific information to the *Contract Plugin*. The plugin provides functionalities for the *Contract API* and for *Contract Interceptors*. Regarding the API, all inspection and manipulation opportunities are originally realized by the plugin. It internally stores necessary information and exposes it through API elements, similar to the proceeding performed for the *mKernel API*, as discussed in section 6.1.3. The *Contract Plugin* also accepts configuration instructions regarding information tracking and reaction execution. This information is provided to *Contract Interceptors* in a pull-oriented approach, that is, interceptor instances must request information, if needed. Although each violation occurrence requires an interaction with the plugin, the resulting performance overhead was considered acceptable. Violations basically reflect incorrect behavior and thus indicate a lack of reliability if a bean instance is responsible for their occurrence (see section 1.1). Otherwise, the incorrect behavior results from clients which would also indicate faults or even attacks. Summarizing, contract violations should only occur in rare situations, especially for enterprise systems. Anyway, the underlying problems are estimated of being much more serious than the performance overhead for requesting information of how to proceed. Inconsistencies between the plugin information base and the information stored by *mKernel* are avoided. Therefore, the *Container Plugin* is registered as listener at the *Notification Topic* through which *mKernel* publishes notifications. In this context, the plugin only receives *Type Level* events regarding the removal of ejb-jar files. For this case all tracked violations regarding the affected bean types are removed from the plugin information base[47].

Violation Handlers represent extension points of the approach providing the opportunity to directly engage with the call flow of an invocation during which a contract violation occurred. *Violation Handler*s must be implemented as session beans exposing a specific interface. For being usable by the approach they must be deployed into the target container. On registration of the corresponding reaction type through the API the mapped name of the handler to apply must be submitted. Handler instances are provided with context information and are enabled to initiate the execution of the other reaction types. Furthermore, they can perform direct countermeasures before an invocation is forwarded to its original target or before an invocation result is returned to a client.

In section 7.2.2.1 ***Contract Interceptors*** were introduced as interceptors which are generated individually for each SB, based on its provided interfaces. If a contract violation is identified during the interception of a method invocation, an interceptor instance behaves in accordance with the configuration requested from the *Container Plugin*. This might cover the submission of violation information to the plugin and the execution of the intended reaction such as throwing an exception or contacting a *Violation Handler*. Internally, interceptor instances make use of the context information provided by the *mKernel API* (see section 5.4.4).

[47] In this context, the plugin also stores the identifiers of `EnterpriseBeans` and `EnterpriseBeanTypes` as part of each violation data set. This was performed, because the information that a certain bean or bean type is related to a violation was considered important even if the corresponding *Instance Level* or *Deployment Level* information is not available anymore. For such a case the *Contract API* would provide access to the violation information, but would not be able to establish associations to the removed counterparts of the *mKernel API* anymore.

7.3. Evaluation

The project presented in section 7.1 addressed the self-configuration objective of autonomic entities. In this context, the planning and execution of seamless reconfigurations are supported. As conceptual foundation a stepwise approach for constructing complex reconfiguration procedures out of basic building blocks is realized, allowing the reuse of core functionalities in different contexts. Additionally, the provided parameterization support for complex reconfiguration strategies further promotes reusability. The resulting framework is designed extensible regarding the integration of custom executors for different tasks. Through applying the approach to *Strategies* identified in literature, it could be shown that the framework provides a reasonable and useful level of support. This covers the general capability of the framework to reflect relevant reconfiguration procedures, as well as the desired reusability of building blocks and *Strategies*.

The second project, which was discussed in section 7.2, focused on the self-healing and self-protection objectives of autonomic entities through applying the concept of DbC to AC. In this context, the provided approach supports the specification and integration of contracts during component development through a contract language and annotations, respectively. Regarding system management, the approach allows model based administration through an extension on top of the *mKernel API*. This extension provides effectors for configuring managed modules with respect to the tracking of violation information and the definition of countermeasures to violation situations. For inspection purposes sensors do support the inspection of *Type Level* aspects regarding contract definitions. Moreover, *Instance Level* inspection is also supported, for example, to analyze concrete violation situations. In this context, associations to elements of the *mKernel API* enable the utilization of *Instance Level* information provided by *mKernel*. Finally, the approach is extensible with respect to the integration of custom violation handlers. These allow managing entities to engage with a violation situation and to immediately perform custom countermeasures.

The two projects and their results make use of different sensors and effectors provided by the *mKernel API*. To give a general overview of the applied features, table 7.1 contains a compilation of relationships between the major topics of the API and their usage in the context of the particular project. Within the columns for the projects there do exist three possible characteristics ('-', 'x' and '+'). A '-' indicates that the particular project does not make use of the corresponding feature. Through an 'x', it is depicted that the corresponding feature is used in the context of the project, but the full potential is not exploited. Finally, a '+' is inserted if the full potential of the feature aspects are applied for the project.

Kernel API	Self-Configuration Project (Section 7.1)	DbC Project (Section 7.2)
Type Level	+	x
Deployment Level	+	x
Instance Level	-	+
Notification Facility	-	+
Seamless Reconfiguration	+	-

Table 7.1.: Relationships between Projects and *mKernel*

The table shows that each feature was used extensively in the context of at least one of the projects. Regarding the *Type Level* and the *Deployment Level* of the API, the *Self-Configuration Project* made use of nearly each of the provided inspection features for reconfiguration preparation. Moreover, the *Deployment Level* features for adaptation support were also used extensively. *Seamless Reconfiguration* was one major foundation of the project, because it provided the basis for dynamic adaptation. In contrast, the *Instance Level* and the *Notification Facility* were not relevant for the project, because it concentrated on adaptation aspects. The *DbC Project* mainly focused on the *Instance Level*. In this context, nearly all aspects of the corresponding part of the API were exploited also covering access to context information during invocation execution. Furthermore, the *Notification Facility* was used for keeping consistency between the information stored for the approach and the underlying database of *mKernel*. For the identification of defects and errors *Deployment Level* and *Type Level* aspect were also used. Nevertheless, they were only considered partially. For example on both levels aspects of parameter adaptation were not relevant. Finally, the project did not aim to deliver any support for system reconfiguration. Therefore, the corresponding effectors on *Deployment Level* and the *Seamless Reconfiguration* facility were not needed.

Both projects presented in this chapter were successfully finished. Their relevance in the context of AC could be shown through the publication of a corresponding paper and an article. For the two projects *mKernel* provided an auxiliary and profound foundation which enabled their realization. In this context, all of the features provided by *mKernel* were evaluated regarding their practical usability and their adequacy regarding the support for different application areas of AC.

8. Related Work

After the discussion of the AC-infrastructure and its application within the previous chapters, this chapter provides an overview of related work. The chapter is structured as follows: Section 8.1 presents approaches which mainly focus on architectural aspects of software and systems and highlights their relation to this thesis. In section 8.2, approaches for infrastructure management support are discussed. Afterwards, within section 8.3 approaches are discussed which explicitly address the EJB standard. Finally, section 8.4 discusses related work of the two projects presented in chapter 7.

8.1. Architecture-centric Approaches

The meta model of *mKernel* and the corresponding API can be used by autonomic entities, amongst others, for obtaining an insight into the architecture of a managed system and for manipulation purposes. On *Deployment Level* the basic building blocks of a system are represented through deployed components (EJB modules), their access points (enterprise beans) and connections among them.

The design of the *Type Level* and the *Deployment Level* regarding the top level element is – besides the specifications of the EJB standard – inspired by so called *Architecture Description Languages* (ADL). These languages are mainly used for the development of software to describe a software architecture through coarse-grained elements abstracting from the source code level and fine-grained constructs like classes for object oriented approaches (cf. [113]). Furthermore, they might be used for the description of system architectures. The majority of approaches focuses on special aspects of architectures and is not intended to provide a comprehensive foundation for architectural considerations. These approaches were of minor interest, because they demand for their integration into software development which was explicitly excluded from this thesis. Nevertheless, the general approach of ADLs inspired this thesis through the provision of ideas for the layout of the meta model. In this context, technological aspects are relevant while recent considerations regarding the inclusion of business and domain related aspects are negligible (cf. [112]). The remainder of this section discusses one exemplary ADL approach which is considered relevant for this thesis, namely *xADL 2.0*. For further considerations about ADLs, as well as their impact on AC, please refer to the corresponding literature (cf. , e.g., [85], [112] and [113]). Additionally, a model-based approach for system management, called *Fractal*, is sketched.

xADL 2.0 *xADL 2.0* is an XML-based ADL which can be used for describing software and system architectures (cf. [54,55]). It was realized with the main intention to provide an extensible foundation for supporting the development of different other ADLs. Therefore, *xADL 2.0* defines certain XML schemata which are associated with each other. An extension of the basic language elements can be realized through the definition of a new schema which imports existing ones.

xADL 2.0 supports, amongst others, the concepts of *Components, Connector,* and *Configurations*. In this context, the term *Component* denotes coarse-grained building blocks of a software or system in general. Connectors are used for representing communication channels, and configurations define constraints regarding the arrangement of components and connectors. A hierarchical structuring of architectures, as well as the aggregation of elements into groups are also possible. Furthermore, concepts for the declaration of optional or alternative elements, as well as versions of elements are also supported. *xADL 2.0* distinguishes between software and system architectures. Finally, an extensible development tool for *xADL 2.0* is provided [53].

Dashofy et al. [56] discuss the application of *xADL 2.0* in the context of self-healing. For this purpose they propose that the architecture of a running system is represented through an *xADL 2.0* model which is analyzed, for instance, regarding broken or missing connections. As results of planning, a new architecture representation would be provided which solves the identified shortcomings. Afterwards, the differences between the current and the desired architecture would be calculated, consisting of additions and removals of elements. Finally, the results of planning would be used as foundation to perform reconfiguration actions.

Fractal With *Fractal* a generic, extensible, and reflective meta model for component oriented systems is provided (cf. [36])[48]. The basic goal of this project is to provide a comprehensive foundation for the development of software, as well as the deployment and reconfiguration of systems. Therefore, the meta model considers a managed system as a collection of interconnected and potentially nested deployed components. Each component is, amongst others, characterized by a set of interfaces which might be used for inspection and manipulation purposes. In general, there do exist four types of interfaces which can be used for parameter adaptation, compositional adaptation, manipulations of component hierarchies, and for controlling the life cycle of deployed components. The generic nature of *Fractal* demands that the interfaces are kept rather gen-

[48] Actually, the authors of *Fractal* speak about a component model, but this terminology would be misleading in the context of this thesis. The same holds for the term Component which is – according to the authors – a runtime entity. In the following, the terminology of this thesis is used.

eral[49]. Therefore, they are envisioned to be extended for special application areas.

The meta model is intended to be realized for different platforms. In this context, a Java-based reference implementation, called *Julia*, is provided (cf. [37, 38]) which implements the meta model through a corresponding API. In order to make a component usable with *Fractal*, developers must implement the different interfaces required for components. In this context, the concept of separation of concerns regarding the implementation of the core application logic and management aspects is supported. Nevertheless, the need to consider management aspects during development or maintenance leads to increasing complexity during the software life cycle. Finally, an XML-based, extensible ADL for *Fractal* models and an extensible tool are also provided for *Fractal* (cf. [101]). For further details about implementations of *Fractal* and extensions, please refer to the project website [39].

The two representatives of architecture-centric approaches for software construction and system management discussed above, focus on the establishment of generic foundations for their particular application area. In contrast, *mKernel* represents a standard-specific management infrastructure. *mKernel* is intended for but not limited to its usage by autonomic entities. Furthermore, it could complement existing model-based approaches as the ones discussed above through its application as system infrastructure. This would also promote the integration of *mKernel* into a broader management context. The major conceptual task for the integration of *mKernel* with a model-based approach would be the need to provide a mapping for the *mKernel API* to fulfill the requirements of the particular approach, for example, through adapters.

[49] Regarding the life cycle of a deployed component, only the states *STOPPED* and *STARTED* are considered.

8.2. Infrastructure-centric Approaches

The AC-infrastructure establishes the foundation for the autonomic management of the business-tier of enterprise systems. In contrast, the management of the underlying infrastructure such as the internals of a Java EE server or an EJB container, are explicitly excluded from *mKernel*.

There do exist different approaches for the management of system infrastructures in literature. This section addresses two major areas of infrastructure management which are contiguous to the work on *mKernel*. First of all, the research area of *Reflective Middleware* is sketched shortly. Afterwards, Java EE-based approaches for infrastructure management are presented. Finally, a summarizing discussion regarding the relations of the presented approaches to *mKernel* is provided.

Reflective Middleware The research area of *Reflective Middleware* addresses the management of the underlying middleware infrastructures of systems. In this context, various approaches are proposed which address different aspects of middleware such as supported interaction paradigms, the target middleware platform ,or the application domain. The basic idea of reflective middleware platforms is to enable the configuration of the underlying infrastructure according to the particular needs of the concrete application context. This aspect of reflective middleware directly relates to compositional and parameter adaptation, as discussed in section 1.2.2. In this context, the underlying infrastructure instead of the application system is the target of adaptation. For further discussions regarding *Reflective Middleware*, provided platforms, and alternatives for their classification, please refer to the corresponding literature (cf., e.g., [65] and [132]). In the following, two reflective middleware platforms are considered as exemplary representatives, namely *dynamicTAO* and *OpenORB v 2*. They were chosen, because they follow a similar approach to *mKernel* concerning the addressing of an existing standard as foundation.

dynamicTAO [96] represents an extension of *The ACE ORB* (TAO) [136] which itself is an implementation of an *Object Request Broker*(ORB) for the *Common Request Broker Architecture,* revision 2.2 (CORBA) [120]. *TAO* was designed to provide a configurable *ORB* for real-time systems which allows its adjustment to concrete application scenarios through configuration files. Such a configuration can contain settings for various aspects of an *ORB* such as the applied thread scheduling strategy or security policies. The configuration is read during startup of an *ORB* and is used to instantiate *Strategies* for the different configuration aspects. In relation to the discussion in section 1.2.2, *TAO* supports load-time adaptation, because applied strategies cannot be changed after startup. *dynamicTAO* extends *TAO* through the concept of so-called *Configurators* which are used to intercept and manipulate bindings between system elements and strategies, as well as among strategies at runtime. In this context, they allow interaction blocking and connection rerouting. Furthermore, strategies can also be reloaded at runtime. In order to allow their dynamic exchange, strategies must exhibit different functionalities, for example, for state extraction and injection. For further details regarding *dynamicTAO,* please refer to the corresponding literature (cf., e.g., [96] and [98]).

OpenORB v2 represents an alternative, reflective realization of CORBA [23]. *OpenORB* relies on a combination of the three concepts component orientation, computational reflection, and component frameworks for providing opportunities to manage an ORB at runtime. It is established out of components which can be managed through a meta model consisting of a set of complementary views focusing on different aspects. In this context, two views are envisioned for addressing structural aspects. The first one considers interface-related aspects, that is, required and provided interfaces of components. The second view addresses architectural aspects, that is, connections between required and provided interfaces. Two additional views are provided covering behavioral aspects. These views allow the management of interceptors and resources. Components are organized into frameworks which address different aspects of an ORB such as the management of protocol stacks or scheduling strategies. These allow the addressing of specific features of the particular aspect of a framework. The set of component frameworks itself is configurable which enables a high degree of adaptation freedom. For a detailed discussion about the realization of *OpenORB v2*, please refer to the corresponding literature (cf. ,e.g., [23] or [24]).

Summarizing, it can be stated that *OpenORB* provides a higher degree of flexibility compared to *dynamicTAO* regarding the opportunities for infrastructure adaptation, because its support for reconfiguration is not limited to a predetermined set of aspects. *Sadjadi and McKinley* classify *dynamicTAO* as tunable, because it supports adaptation regarding aspects anticipated during development. In this context, *OpenORB v2* is characterized as mutable, because it also allows adjustments of unanticipated aspects. This terminology should not be confused with the one applied in section 1.2.2 for characterizing sub-types of compositional adaptation. *Kon et al.* [97] provide a summarizing comparison of *dynamicTAO* and *OpenORB*, specially highlighting design aspects. This article was jointly written by authors belonging to both projects.

Java EE-based Approaches The management of Java EE servers is addressed by different approaches. In the following, four approaches are discussed which address different aspects of Java EE servers.

The *PeKing University Application Server* (PKUAS) [83, 114] is an implementation of a Java EE server which supports version 1.3 of the Java EE standard [138]. It provides a reflective meta model which allows inspection and manipulation of application server internals such as the exchange of communication protocols, or to inject or remove interceptors[50].

[50] These interceptors are specific to *PKUAS* and should not be confused with the intercep-

PKUAS supports model-based management of deployed components, focusing on EJB modules, in accordance with version 2.0 of the EJB standard [61]. Most of the aspects of enterprise beans declared in a DD might be inspected and manipulated at runtime (cf. [83, 114]). *Wang et al.* [158] discuss the replacement of enterprise beans, also covering the proceeding for instances and references. In this context, instances can be replaced only if their internal state representation did not change between the replaced and the replacing version regarding the constituent fields. Otherwise, instances of the original beans are kept until their clients do not need them anymore. At the same time instance of the replacing bean are already accessible for newly established references. Alternatively, it is possible to delete original instances which might lead to connection losses for clients. The inspection and manipulation facilities of *PKUAS* are exposed through an API which is an extension of the JSR 88 discussed in section 3.4.1 (cf. [84]). Furthermore, certain performance statistics are provided. Consequently, *PKUAS* can be seen as application server which provides a high degree of freedom regarding its support for dynamic compositional adaptation and parameter adaptation at runtime, compared to other server implementations.

PKUAS also addresses the relation between software development and a software system. In this context, it follows an approach for graphically representing a managed system. For the deployment phase of the system life cycle corresponding tools allow the creation of necessary information, for example, for the DD. In order to represent a managed system, two approaches are considered depending on the availability of architecture descriptions from the development phase. If development information is available, it is used as foundation for the generation of a representation of a managed system. Otherwise, a model of the system is derived from the system itself, not covering development specific information (cf. [82]). In

tors considered in the EJB standard, version 3.0.

order to manipulate a managed system through its visual representation, the corresponding tool makes use of the JSR 88-compliant API (cf. [83]).

The support for parameter adaptation at runtime as provided by *PKUAS* is more flexible than the one provided by *mKernel*. *mKernel* does, for instance, not allow the manipulation of transaction or security settings during the whole life cycle of an EJB module. In contrast, *PKUAS* is able to support these adaptations, because it directly controls the implementations of the corresponding facilities. Comparable opportunities are not supported by the EJB standard which builds the foundation for *mKernel*. This does *not* mean that *mKernel* does not support parameter adaptation for deployed components regarding security or transaction settings. Nevertheless, its execution would require additional efforts for EJB modules in state `DISTRIBUTED` or `STARTED`. For both cases the module would have to be transferred to the `EXISTS` state before adaptation execution. Afterwards, it could be brought back to its original state. For a module in the `STARTED` state it would be additionally necessary to declare a quiescence region solely consisting of the affected module. This allows the extraction of conversational states of stateful SB instances. After adaption execution new instances of the affected beans must be created, and the extracted states must be injected into them. Finally, existing references must be rerouted. This proceeding avoids any system disruptions and is generic for all conceivable adaptation scenarios. It could be implemented once as extension on top of *mKernel*. Nevertheless, it would require certain state transitions and conversational state transfers. Therefore, it induces a certain performance overhead compared to *PKUAS*. Furthermore, the limitations regarding quiescence regions do also hold for this scenario (cf. section 5.6.4).

The provided level of granularity and the corresponding freedom for adaptation alternatives is much higher for the *mKernel* infrastructure[51].

[51] Except the case discussed above.

PKUAS does, for instance, not expose any concept comparable to quiescence support of *mKernel* which can be used for reconfiguration orchestration. Facilities for controlling the semantics of rerouting are also not provided. Furthermore, the fine-grained opportunities for the inspection and manipulation of conversational states of EJB instances as provided by *mKernel* have no corresponding counterpart inside *PKUAS*. Consequently, *mKernel* is much more flexible, because it also provides facilities for a state transfer between stateful SB instances with different underlying conversational state representations.

The meta model of *mKernel* is much more detailed than the one provided by *PKUAS* regarding enterprise bean related aspects. In this context, the *PKUAS* meta model does not support *Instance Level* aspects at all. Therefore, the corresponding opportunities discussed in section 5.4 are not available. Furthermore, *Type Level* aspects are not considered as fine-grained as in *mKernel*, for example, regarding the identification of the underlying implementation of an enterprise bean. In contrast, the meta model of *PKUAS* is mainly limited to the *Deployment Level* of a managed system. The design of a software architecture is also considered by *PKUAS*, but not covered directly. In contrast, it is mapped to an external design which stands in a one-to-one relation to the system architecture. In this context, it is intended to cover additional information not included in the system architecture itself. The information might, for instance, be specific to the development and maintenance phases of the software life cycle. Comparable information might also be integrated into the *Type Level* of the *mKernel* meta model and requested from it through custom artifacts of ejb-jar files. This would lead to a higher level of independence from external information sources and to self-contained archives. The inspection opportunities provided by *mKernel* on *Type Level* such as the identification of identical implementations of enterprise bean types, have no corresponding counterparts in the *PKUAS* meta model. In contrast, the design representation as considered by *PKUAS* can be seen as extended

Deployment Level.

Summarizing, it can be stated that the *PKUAS* project provides a container implementation which delivers a very high level of adaptation opportunities compared to other application servers. Its reconfiguration facilities are exposed through container-specific extensions of the JSR 88 standard. *PKUAS* was rejected as foundation for *mKernel* for three major reasons. First of all, *PKUAS* does not supported the target component standard EJB 3.0. In contrast, version 2.0 of the EJB standard – which is supported by *PKUAS* – lies two revisions behind the current version of EJB. Secondly, it would have been necessary to rely on container-specific facilities for the realization of *mKernel*. As stated in section 1.3 in the context of requirement *COR-SC*, the usage of container specific extensions should be minimized as far as possible. Finally, the provided facilities do not support all aspects required by *mKernel* as stated above such as quiescence support or conversational state transfers. Therefore, these facilities would have to be extended or could not be used at all.

Beyond a comprehensive support for Java EE through the provision of a specific application server, there do exist approaches which address the management of Java EE-based systems on top of application servers. Three of these approaches, addressing the recovery from failures, are discussed in the following.

The *JBoss with Application-Generic Recovery* (JAGR) approach proposed by *Candea et al.* addresses failure recovery in a single Java EE server environment (cf. [42]). It is realized on top of the *JBoss Application Server* [6]. The general idea is to provide an infrastructure for the identification of failures and the execution of autonomic recovery actions in response. The identification of failures is based on a monitoring facility which applies different monitors, for instance, to identify (system) exceptions[52]. Fur-

[52] In this context, system exceptions indicating null-pointer-references or out-of-memory

thermore, other monitors might be configured according to the specifics of a managed system, for example, to identify failure notifications returned to clients through web pages. Monitoring information is sent to so called *Recovery Managers* which analyze the information and initiate countermeasures. For the proposed approach these consist of the undeployment and subsequent deployment of affected enterprise beans. In this context, client invocations are blocked during redeployment, but no conversational states are transferred for stateful SB instances. Consequently, client connections might be broken. *Candea et al.* argue that this is negligible for the given scenario. Certain concepts of *JAGR* which address the business-tier of a managed system might also be realized based on *mKernel*. An example would be the identification of system exceptions and the execution of countermeasures. In order to identify relevant exceptions, a proceeding similar to the one presented in section 5.4.5 in the context of incident tracking could be applied. Reactions might be performed based on *Deployment Level* facilities of the *mKernel API*.

Abdellatif and Danes [12] address management automation based on the *Java Open Application Server* (JOnAS) [7] which is a Java EE server, compliant to version 1.4 of the Java EE standard [139]. The approach is realized on top of JSR 77 and JSR 88 (see section 3.4). Its main focus lies on the autonomic recovery from software or hardware failures in a cluster environment. The approach is based on a distributed system consisting of multiple application servers which are grouped into so called *Domains*. Newly started servers are identified and attached to the particular groups through a discovery service which is specific to *JOnAS*. Software or hardware failures are recognized through polling the availability of management endpoints in regular intervals. If a certain endpoint does not react for a certain timespan, recovery actions are initiated. These might either consist of the reboot of the affected software (system) or of the startup

exceptions are explicitly mentioned by *Candea et al.* (cf. [42]).

of a new *JOnAS* instance on a different server in case of a hardware failure. The approach of *Abdellatif and Danes* is specific to *JOnAS* in that it makes use of the corresponding discovery service and requires certain adjustments of the *JOnAS* implementation.

With the *Jade* system, *Bouchenak et al.* [26] propose an infrastructure for coarse-grained self-healing of distributed, multi-layered server infrastructures. In [26], *Bouchenak et al.* propose the application of *Jade* to a Java EE environment. The system is based on *Fractal* for representing the architecture of a managed environment. As top level elements of a server infrastructure, server nodes are considered. These can be used to deploy and configure so-called *Components* which represent system elements such as a web server, an application server, or a database management system. Components might be further configured, for instance, regarding the deployment of sub-components like web modules inside a web sever. The underlying, *Fractal*-based meta model is not discussed in detail. Especially sub-components are only mentioned casually. The major focus of *Jade* lies on the discussion of the extensible architecture of the self-healing system itself. In this context, the approach addresses failures on the level of node availability and component failures. For both cases it proposes rudimental recovery policies. Examples of these are the redeployment of a failed component on the same node or the deployment of affected components on a different node in case of a node failure, both including the rebinding of connections. In this context, *Bouchenak et al.* mention that the states of components are explicitly excluded from their considerations. Furthermore, the examples presented in [26] only consider failures within the web-tier. One specially highlighted aspect of Jade is its ability to perform self-healing for its own system architecture through the deployment of system elements into nodes.

The infrastructure-centric approaches discussed in this section are related to *mKernel* insofar as they address the underlying platform of a managed system. They could possibly be used in combination with *mKernel* to support separation of concerns regarding the management of a system on different layers, that is, on business-tier, the corresponding Java EE server, the underlying middleware platform, and the hardware environment. For this purpose it would at least be necessary to establish a relation between the different representations of a managed system covered by the particular management layer. This is already supported by *mKernel* for the relationship between its meta model and the view on a managed container exposed by the *GlassFish Application Server* [11] through JSR 77 and JSR 88.

In combination, the architecture-centric approaches discussed in the previous section and the infrastructure-centric approaches discussed in this section enframe the AC-infrastructure proposed in this thesis. Their underlying concepts could contribute to the development of comprehensive and holistic approaches for the management of layered system architectures in heterogeneous environments. In this context, architecture-centric approaches could be applied for addressing top-level aspects of the overall system architecture. Furthermore, they might provide integrated access points for the specific management of system parts on different layers. These might be realized by infrastructures like *mKernel* on the business-tier of systems and by approaches like the ones discussed in this section for the management of underlying containers, middleware, or hardware environments. The integration of additional layers such as operating systems or network infrastructures, might be meaningful extensions. Through such an approach, separation of concerns would be supported on the one hand, while the tracing of relationships and interdependencies among elements of different layers would provide helpful navigation opportunities for in-depth analyses.

8.3. EJB-specific Approaches

Within this section, related work is discussed which directly addresses the EJB standard. In this context, three approaches are presented, proposing facilities and concepts which might be considered as future work for their realization on top of *mKernel*.

Rutherford et al. [131] propose the *Bean Automatic Reconfiguration frameworK* (BARK) for supporting life cycle management of deployed EJB components through the automation of activities. *BARK* is designed as management tool which can be used by human administrators. It is realized on top of the *JBoss Application Server* [6]. *BARK* consists of a management frontend and a server plugin which is realized in a Java EE compliant way, but makes use of *JBoss*-specific facilities. *BARK* supports different activities for life cycle management such as the upload or removal of archives, the deployment and undeployment of modules, and the management of connections between enterprise beans. *BARK* is designed for a multi-server environment, that is, it explicitly considers multiple instances of the *JBoss Application Server*. In this context, it supports the orchestration of dependent reconfigurations performed on different servers. Furthermore, the tool allows the execution of reconfiguration scripts which contain a sequence of multiple reconfiguration actions. In order to perform adaptations, the framework manipulates the naming facility for supporting name indirection. Internally, *BARK* makes use of framework specific mapped names. This was done to minimize system disruption during reconfiguration, because the indirection might be changed during module replacement after the new module is ready for servicing client requests. This approach is similar to the proceeding applied for *mKernel* regarding the distinction between *Access Layer* and *Managed Layer* (see section 5.1). *Rutherford et al.* do not consider bean instances and do consequently not support the transfer of conversational states.

In relation to *mKernel*, *BARK* concentrates on the *Deployment Level*. Although *mKernel* provides far more comprehensive facilities for adaptation, the support for orchestrated reconfiguration in multi-server environments might be worth further considerations in case *mKernel* would be extended regarding this aspect. The design and realization of *mKernel* was developed with the intention to support autonomic entities with a rich set of sensors and effectors for fulfilling their objectives. Nevertheless, it would also be possible to develop tools which expose the opportunities provided by *mKernel* to human administrators. In this context, it would be possible to realize different wizards, for example, to generate configuration proposals according to the *Type Level* and *Deployment Level* plans presented in the sections 5.2.5 and 5.3.5.

Jarir et al. propose an approach for the dynamic injection and removal of so-called *Middleware Services* (cf. [90]). Their approach is based on version 2.0 of the EJB standard [61] and is applied to the *JOnAS* application server [7]. The concept is realized through an extension of the *JOnAS* implementation to integrate an interception facility. The facility redirects method interactions arriving at bean instances to a so-called *Adaptation Engine*. This engine forwards the call flow to *Services* in accordance with internally stored policies. These policies might, for instance, lead to a forwarding of invocations in case of performance degradation. *Jarir et al.* mention a logging facility as an example for a service. The approach of *Jarir et al.* does not address compositional adaptation regarding the manipulation of enterprise beans and connections among them. In contrast, it allows the dynamic integration and removal of interceptors. Furthermore, the conditional forwarding of interactions further promotes flexibility of their approach.

The infrastructure for interaction interception had to be integrated, because version 2.0 of the EJB standard did not address interceptors as in version 3.0 of the standard (see section 3.1.4). The realization of this approach would be facilitated for EJB 3.0, because the rerouting of interactions would be possible solely based on the interceptor facility. If the adaptation engine would also be realized on top of the EJB standard, the approach would not be limited to *JOnAS*.

In relation to *mKernel* a comparable approach could be integrated without great efforts. It could be realized through the integration of an additional interceptor which conditionally forwards interactions to services. The integration of the interceptor itself could be performed based on a rather straightforward extension of the preprocessing tool (see section 6.2). For performance reasons it would be meaningful to assign the task of policy evaluation to the interceptor, too. In this context, the distribution of context information could be performed similar to the proceeding discussed for the *Management Context* in section 6.1.2.1. Finally, the support for call context information as presented in section 5.4.2 would allow the definition of policies also covering aspects exposed through the *mKernel* meta model.

White et al. propose the *J3Process* as foundation for the development of self-managing software based on version 1.1 of the EJB standard [109] (cf. [162, 163]). For this purpose a set of tools is provided which supports developers to fulfill their tasks. In particular, a modeling tool, an artifact generation tool, and a framework providing basic facilities are contained as part of *J3Process*. Based on these tools, the approach proposes a stepwise proceeding for the development of software with autonomic capabilities based on QoS requirements. During design, functional aspects are modeled based on enterprise beans. Additionally, QoS goals and goal hierarchies can be defined and attached to enterprise beans. In the end of design, different artifacts can be generated through execution of the corresponding tool. In this context, the later content of an ejb-jar file is extended with so-called guardians which supervise interactions at runtime, for example, to identify exceptions or to collect performance measures. Furthermore, stubs for analysis classes are constructed which should later on evaluate goal fulfillment. At runtime guardians are intended to forward collected information to analysis instances which report goal violations back to the corresponding guardian and upward along goal hierarchies. Guardians might be configured with strategies to trigger countermeasures for goal violations at runtime. Furthermore, certain simulation facilities are also provided. These are not relevant in the context of this thesis, because they only affect the development phase of the software life cycle.

With *J3Process* an approach is provided which mainly addresses the development of EJB components into which facilities for the identification and reaction to QoS goal violations are integrated. Through this proceeding a certain architecture for autonomic management is determined. *mKernel* would be able to support similar approaches which are based on EJB 3.0. In fact, it could facilitate their application through the provision of interaction context information and the opportunities provided by the API in general. In relation to the project discussed in section 7.2, contract adherence could be interpreted as QoS goal. Moreover, the immediate reactions to contract violations could be seen as a kind of strategies.

8.4. Related Work of mKernel Applications

This section discusses related work of the two projects presented in chapter 7.

Support for Self-Configuration As already highlighted in section 7.1.1, the inspiration of the project for supporting self-configuration of EJB-based system was provided by *Rosa et al.* [129]. In this context, the proposed reconfiguration strategies were taken as foundation for the evaluation of the project regarding the appropriateness of the underlying concepts for supporting different reconfiguration scenarios.

Additionally, the project is also related to the concept of quiescence management and the corresponding related work, as discussed in section 5.6.1, because it heavily relies on the support of this concept for complex reconfiguration scenarios.

Finally, the project is related to the *PeKing University Application Server* [83, 114] discussed in section 8.2 in that this server supports the replacement of EJB modules according to one single strategy which is coarsely comparable to the non-interrupt strategy presented in section 7.1.2.2. In this context, the results of the project provide a very much higher level of flexibility for controlling the concrete execution of adaptation compared to *PKUAS*.

Self-Healing and Self-Protection based on Contracts There do exist different solutions for making the concept of DbC applicable to the Java Programming language, like *Jass* [19] or the *Java Modeling Language* (JML) [41]. These approaches mainly address the enforcement of contract adherence at runtime. In response to identified contract violations, exceptions are thrown. The *Assertion Facility* [107], being integrated into the Java programming language since version 1.4, goes into the same direction, but only allows the integration of contracts into classes. Therefore, it is not directly applicable to CO where contracts are considered with respect to interfaces. In contrast to these approaches, the project results provide opportunities for in-depth analyses of violation situations. Moreover, different alternatives regarding reactions to violations are integrated. Therefore, the approach does not focus on strong contract enforcement, but mainly addresses the analyzing and handling of violation situations in a platform specific context.

On top of *JML* there do exist approaches for supporting unit testing based on contract specifications. While these approaches concentrate on the extraction and generation of test oracles (cf. [45]) and test data (cf. [46]), the presented project focuses on the provision of context information for automated analyses of contract violations. Moreover, it provides facilities to directly react to violation situations. Nevertheless, the presented approach could also be used during software development and maintenance for testing purposes. In this context, the provided information about contract violations could provide a helpful foundation for the discovery of errors.

Finally, contracts as considered by *Meyer* [116] do not need to be solely used for the evaluation of violations regarding parameter values and return values. Instead of that, they might also be applied to services with underlying state machines. A corresponding framework was realized in the context of an earlier project by *Bruhn et al.* [31]. This project aimed to provide an infrastructure for scheduling client requests to instances of equivalent services. For this purpose clients do request service references through the provision of required states. These are matched with state information exposed by services. If at least one matching service is found which is free for servicing a new interaction, a corresponding reference is returned to a requesting client. Otherwise, requests are queued until a matching service is available. The underlying idea to reflect state machines through contracts could also be realized based on the project presented in section 7.2. In this context, correct states could be represented through invariants, and valid state transitions could be reflected through postconditions.

9. Conclusion

In the beginning of this thesis, complexity of software and systems was identified as one major challenge information technology has to address to enable its future development. All the more, this can be stated for enterprise software and systems because of the diversity of business areas to support and the corresponding relationships between software and system elements. Moreover, dynamic environments, changing requirements, and potential relationships with external clients further complicate the management of enterprise systems, as well as the maintenance of the corresponding software. Finally, high demands on trustworthiness have to be taken into account also. Consequently, enterprise software and systems are highly affected by complexity and therefore require its addressing badly.

Software Engineering was considered as discipline which mainly addresses the complexity of software during development and maintenance. In this context, the concept of *Component Orientation* was developed to support the construction of software based on clearly distinguishable building blocks called *Components*. These are ideally characterized by a comprehensive specification which facilitates their integration into complex software and systems. Furthermore, components are usually developed against a certain *Component Standard* which provides the general frame for their construction and execution.

Through the vision of *Autonomic Computing*, the complexity of systems and the corresponding tasks during their management are addressed. AC is based on the idea to assign administrative tasks to a managed system itself to disburden human administrators. Consequently, administrators can concentrate on the specification of high-level goals while leaving their realization open to autonomic entities. To enable autonomic management, autonomic entities must be equipped with *Sensors* and *Effectors* which build the foundation for inspection and manipulation of a managed system.

CO and AC both address the complexity problem, but focus on different aspects. While CO concentrates on the software life cycle and the deployment of systems, AC mainly focuses on system management. In combination, CO and AC cover all aspects of software and system life cycles regarding complexity treatment.

The underlying idea of this thesis is to bring together CO and AC through the realization of an infrastructure for supporting the autonomic management of component oriented enterprise systems. Such an infrastructure is assumed to provide a common ground which enables the realization of concepts and approaches for different application areas of AC. To reach this goal the infrastructure must provide a comprehensive set of sensors and effectors and must not be limited to certain AC aspects. To ensure that the provided infrastructure is of practical relevance, a broadly accepted and widely used component standard was chosen as technological foundation for this thesis. In this context, the *Enterprise JavaBeans, Version 3.0* standard was selected, because it is the de-facto standard for the development of the business-tier of Java-based enterprise software and the operation of corresponding systems. Furthermore, it provides an appropriate component model and has reached a sophisticated maturity level.

The remainder of this conclusion is organized as follows: Section 9.1 provides an evaluation of the provided infrastructure. Within section 9.2, an outlook is provided which consists of conceivable extensions of *mKernel* and considerations on the upcoming *Enterprise JavaBeans, version 3.1* standard [133].

9.1. Evaluation

Within section 1.3 of this thesis a set of requirements for a generic AC-infrastructure was established. These requirements were organized into four groups, namely *Component Orientation Requirements, Software Requirements, Manageability Requirements,* and *System Requirements.* In order to evaluate *mKernel* against these requirements, five characteristics were applied. These are shown in table 9.1. For each of the characteristics (*Char.*), a short description is also covered within the table.

Char.	**Description**
--	Indicates a violation of a requirement which makes the practical application of *mKernel* impossible.
-	The requirement is violated, causing serious impacts on the practical application of the infrastructure.
0	The corresponding requirement is not fulfilled, but the violation is not considered serious.
+	The requirement is fulfilled to a very high degree, only minor violations are present.
++	Indicates the complete fulfillment of the requirement.

Table 9.1.: Evaluation Characteristics

Table 9.2 on page 322 contains an evaluation of *mKernel* against these requirements. The first column covers the group specific abbreviation of the particular requirement. Within the second column, the full requirement name is given. Afterwards, the third column shows the evaluation results against the characteristics depicted in table 9.1. Finally, the last column contains a short remark regarding the fulfillment of the particular requirement.

In the following the four groups are shortly discussed with respect to the fulfillment of the constituent requirements.

Component Orientation Requirements (COR-)			
Abbr.	**Name**	**Eval.**	**Remark**
RAS	Realistic Application Scenario	++	Usage of EJB as technological foundation
SC	Standard Compliance	+	Violations: static fields, object substitution, mapped names
UCI	Unchanged Container Implementation	++	*GlassFish Application Server*, many releases applied over time

Software Requirements (SoftR-)			
Abbr.	**Name**	**Eval.**	**Remark**
FSS	Full Standard Support	+	Interceptors not completely supported
MT	Management Transparency	++	No impacts on component development
SCS	Self-managed Component Support	++	Context information provided at runtime (See section 5.4.4)

Manageability Establishment Requirements (MER-)			
Abbr.	**Name**	**Eval.**	**Remark**
MA	Manageability Automation	++	Automated through preprocessing tool
LCI	Life Cycle Independence	++	Tool can be executed after development or before integration into *mKernel*

System Requirements (SysR-)			
Abbr.	**Name**	**Eval.**	**Remark**
CMS	Centralized Management Support	++	API can be used outside container
RMM	Reflective Meta Model	++	Provided through API
LCC	Life Cycle Coverage	++	EJB module life cycle fully supported
SR	Software Relation	++	Provided through associations between *Type Level* and *Deployment Level*
G	Genericity	++	Generic Meta Model, no limitations known
E	Extensibility	++	No limitations known
CT	Client Transparency	++	Interaction through original interfaces and messages, no extensions necessary

Table 9.2.: Evaluation of *mKernel* against Requirements

Component Orientation Requirements These requirements address the demands on the infrastructure which are directly related to the concept of component orientation, a corresponding standard, and its application by the infrastructure.

Through the selection of EJB as technological foundation for *mKernel*, the requirement *COR-RAS* can be assumed of being completely fulfilled, because EJB represents an accepted standard which is used in practice.

Regarding standard compliance, it was not possible to adhere to all demands of the EJB standard. Nevertheless, it can be stated that for the vast majority of all parts of *mKernel COR-SC* is fulfilled. Therefore, it is rated with a '+'. In particular, *write access to static field* and *object substitution during serialization* are two applied features which directly violate the EJB standard. For both of them alternatives were presented in chapter 6 and the rejection of these alternatives was justified. Additionally, *mKernel* internally makes use of mapped names as considered within the corresponding API of EJB and the XML schema for DDs. This feature is classified as product-specific, not required, and not portable. Consequently, it was not possible to realize a platform independent solution. Therefore, the implementation of *mKernel* is specific to the *GlassFish Application Server* [11] regarding naming. Regarding static fields and naming, please refer to section 9.2.2, because the upcoming version 3.1 of EJB contains new features and considerations which might make the violation of both requirements obsolete.

For the application of *mKernel* no changes of the *GlassFish Application Server* implementation are necessary. Additionally, most of the application server builds which were released in the meantime, were used as foundation for the development of *mKernel*. Therefore, the independence of *mKernel* from one single release or a subset of releases could be shown. The deployment of the container plugin is the only prerequisite for the application of *mKernel* as foundation for system management. Consequently, *COR-UCI* is fulfilled completely ('++').

Software Requirements These requirements directly relate to enterprise software of which corresponding systems should be managed with the help of the infrastructure. They mainly address aspects of the development and maintenance phases of the software life cycle.

The later application of *mKernel* for system management does not restrict or hinder the development of enterprise software. Nevertheless, it was decided not to consider interceptors within the meta model and the corresponding API for two reasons. First of all, interceptors were considered as integral part of an enterprise bean regarding the external view on a component. Secondly, through the exclusion of interceptors it is possible to bypass *mKernel* for interceptors which should perform management tasks like described in section 7.2 in the context of the second project.

Component developers are still allowed to make use of the interceptor facility provided by the EJB standard. Furthermore, they might perform any kind of environmental interactions inside the source code of interceptors. It is only necessary to pass on the application of dependency injection for interceptors and to integrate entries for the local namespace of interceptors into the configuration of the corresponding beans. The former aspect is necessary to avoid the establishment of undesired connections. The later one allows the configuration of interceptors through the corresponding bean configuration, because beans and the attached interceptors share their local namespace.

The integration of interceptors into the meta model would be unproblematic. It would require the extension of the API with additional elements for interceptor representation. The corresponding information could be extracted during preprocessing in the same way as performed for enterprise beans. Moreover, interceptor classes would have to be manipulated for DI simulation which would have to be integrated into the interceptors themselves. DI could be initiated analog to the proceeding for enterprise bean instances. Summarizing, the requirement *SoftR-FSS* is not fulfilled completely, but for the vast majority of the EJB standard which leads to a rating of '+'.

During the development of EJB-based software, developers do not have to consider the later management of corresponding systems at all, that is, they do not need to make use of any *mKernel*-specific APIs. This leads to a rating of '++' regarding the fulfillment of *SoftR-MT*.

For the fulfillment of *SoftR-SCS*, the API provides the opportunity to obtain *mKernel*-specific context information from inside the source code of beans and interceptors. This information can be used as access point for the identification of relevant elements. In this context, all opportunities provided by the API can be used by self-managing entities. Therefore, the fulfillment of *SoftR-SCS* was rated with a '++'.

Manageability Establishment Requirements In order to make components manageable by *mKernel*, adjustments and extensions are necessary. Restrictions regarding the proceeding for manageability establishment are defined by these requirements.

The establishment of manageability for EJB-based components is automated through the application of the preprocessing tool. *MER-MA* was rated '++', because the tool solely requires the archive to process and references to additional archives for its successful execution.

MER-LCI is fulfilled completely ('++'), because the execution of the preprocessing tool is not limited to the software or system life cycle. Furthermore, preprocessed ejb-jar files can be successfully integrated into arbitrary managed systems[53].

System Requirements During the system life cycle, managing entities should be provided with appropriate facilities for supporting system management. In this context, system requirements define the necessary aspects to guarantee a comprehensive support for the vision of autonomic

[53] Assuming the necessary infrastructure is given.

computing.

A centralized management approach, as required by *SysR-CMS*, is supported by the API, because it might be used inside or outside a managed container, or in combination. Consequently, the *SysR-CMS* is rated '++'.

SysR-RMM is fulfilled completely through the *mKernel* meta model and the corresponding API ('++').

Requirement *SysR-LCC* is completely fulfilled ('++') through the *Deployment Level* of the API in combination with the opportunities provided for seamless reconfiguration. Additionally, the original deployment of components is supported based on ejb-jar files which are provided on *Type Level*.

The relationships between the different API levels support *SysR-SR*, because they enable the navigation between software and system aspects from various starting points. This requirement is consequently fulfilled completely ('++').

The design of *mKernel* is kept generic and was established to provide a comprehensive foundation for managing EJB-based systems. Furthermore, during the supervision of the two projects presented in chapter 7 no limitations of *mKernel* regarding its application to different areas of AC could be identified. Therefore, *SysR-G* was rated with a '++'.

Extensibility of *mKernel* is provided through the modular design of the preprocessing tool which allows the integration of additional aspects for special application areas. Furthermore, the support for the integration of artifacts into archives during deployment enables the injection of custom elements for special application areas. Finally, through the notification facility, information consistency between *mKernel* and external data sources can be reached. Summarizing, this led to a rating of '++' for *SysR-E*.

The source code of clients does not need to contain any aspects specific to *mKernel*. Furthermore, clients are enabled to interact with a managed system as if no management is performed. In combination, client transparency is given, and *SysR-CT* could be rated with '++'.

The AC-infrastructure presented in this thesis completely fulfills most of the requirements. Only two requirement (*COR-SC* and *SoftR-FSS*) were not met to the full extent, but could both be rated with a '+'. Summarizing, *mKernel* can be successfully evaluated against the requirements stated in section 1.3.

Beyond the requirements, *mKernel* was evaluated in the context of two projects. Both projects were undertaken under the supervision of the author of this thesis and were finished successfully. The projects addressed distinct application areas of AC, namely self-configuration, as well as self-protection and self-healing. For each of the two projects a corresponding publication points out the relevance of the addressed topics. In combination, both projects made extensive use of all of the features provided by the API. Consequently, the practical usability of the proposed AC-infrastructure can be concluded.

mKernel provides a comprehensive, standard-specific, and realistic foundation for supporting the vision of autonomic computing on top of the EJB standard. The aim to provide a generic foundation for AC exposed various aspects which might help to develop AC-infrastructures for other standards or platforms as well. An example of such an aspect would be the need to establish associations between software and systems to enable complex analyses. Furthermore, the proposed design and realization might contribute to a discussion regarding the requirements for establishing a high-level basis for AC to support the integrated management of heterogeneous environments.

By addressing software and systems on different levels and through the provision of enhanced opportunities for system inspection and manipulation *mKernel* revealed the advantages of such a comprehensive approach. Nevertheless, a comparable solution is – to the best of the author's knowledge – not yet provided in practice. In contrast, the standards for the management of Java EE-based systems (JSR 77 and JSR 88) provide only very rudimental foundations which are left open for vendor specific approaches. In this context, *mKernel* provides a very much more integrated and concrete basis. Therefore, it might contribute to a discussion about the demands for supporting the management phase of the system life cycle in a comprehensive and integrated way. Furthermore, the API shows the advantages of supporting reflection on component level. Therefore, the presented approach might also contribute to considerations about the integration of comparable facilities into component standards.

9.2. Outlook

This section provides an outlook on aspects which were not considered as part of this thesis. In this context, section 9.2.1 addresses possible extensions of *mKernel*. Afterwards, section 9.2.2 shortly discusses the upcoming EJB standard, version 3.1, with respect to relevant changes.

9.2.1. Extension Opportunities

Although *mKernel* provides a comprehensive foundation for the autonomic management of EJB-based systems, there are three areas of extensions conceivable for the future development of *mKernel*. These areas do not directly relate to the provided facilities, but could broaden the range of conceivable application scenarios.

Support for Multi-Container-Environments The application of *mKernel* is limited to one single container which should be sufficient for a broad range of application scenarios. Nevertheless, there are also environments conceivable within which more than one container is used. Each of these containers might be dedicated to a certain field of responsibility. With the current state of *mKernel*, each of these containers would have to be managed in isolation based on independently deployed container plugins. In this case, relationships between elements belonging to different containers could neither be established nor reflected through the API. Nevertheless, such relationships could be of special interest for various reasons, for instance, to allow orchestrated reconfigurations or to analyze interactions across container boundaries.

In order to extend *mKernel* to support multi-container-environments, first of all the meta model would have to be extended with at least one element for representing containers. This would also imply the need to adjust the internal representation of the container plugin, as well as the representation inside managed modules. Secondly, the container plugin would require to be adjusted and extended to support interactions with remote containers and the included plugins, respectively. In order to distribute responsibilities upon the different plugins, there would exist a broad range of alternatives. These might reach from a configuration where each plugin is responsible for its corresponding container up to a centralized solution where a kind of ”*master-plugin*” stores all relevant information and acts as endpoint for the API. For this case all other containers would only contain a kind of ”*executor-plugin*” which accepts and realizes management instructions.

The corresponding conceptual and technological foundations were already considered within the context of a separate project for enhanced naming, called *dyName* (cf. [33]). It has not been discussed in the context of this thesis, because it does not directly contribute to the understanding of the thesis subject.

Integration with related Standards and Approaches For the application of *mKernel* in a heterogeneous environment it would be meaningful to integrate the infrastructure with related management standards and approaches. This might promote the development of holistic approaches for a comprehensive management of systems and infrastructures.

As a first step, the integration of *mKernel* with the two standards discussed in section 3.4 (JSR 77 and 88) would be meaningful. Regarding JSR 77, it would, for example, be possible to extend information provided by the *mKernel API* with additional information exposed through the standard such as statistical data. Furthermore, JSR 88 might be used to manage the underlying container. An establishment of relationships between elements of the *mKernel API* and information exposed by the JSRs would be possible without great efforts. In fact, *mKernel* already exposes the module identifiers assigned by JSR 88 during deployment. These identifiers are also used by JSR 77.

Additionally, the integration of *mKernel* with architecture-centric and infrastructure-centric approaches could be meaningful as discussed in the end of section 8.1 in the context of related work. This might result in a comprehensive approach for the management of complex and heterogeneous IT-infrastructures. Moreover, the disclosure of relationships among system elements and infrastructure layers could enable the development of more sophisticated forms of autonomic management.

Extension of Application Area This thesis concentrated on the EJB standard. Nevertheless, it might be meaningful to extend the underlying concepts to other standards in the context of Java EE or even to client software and systems. This would support the autonomic management of complete systems and would allow more comprehensive analyses such as the identification of defects and errors. Moreover, a broader management scope would enable the orchestration of more complex reconfiguration scenarios. In this context, the restrictions regarding quiescence management discussed in section 5.6.4 could be eliminated.

This extension would demand for a partial redesign of the meta model, at least on *Instance Level*, because relationships with external interaction partners would have to be reflected. Furthermore, quiescence management would have to be revised to allow the establishment of quiescence regions across container boundaries, also enabling the integration of elements not relying on the EJB standard. Finally, a foundation like *mKernel* would have to be established for the affected standards and platforms. This might be a project covering a complexity comparable to the design and realization of *mKernel*.

9.2.2. Enterprise JavaBeans, Version 3.1

At the time of writing, version 3.1 of the EJB standard is in the process of specification. Currently[54], it reached the status of *Public Review*. For version 3.0 of the standard it took approximately nine month from this status up to the final release. Based on the public review of the standard [133], this section discusses the major adjustments and extensions which were considered of having major impacts on a potential further development of *mKernel*.

Singleton Session Beans Within version 3.1 of the standard, the new concept *Singleton Session Beans* is planned to be introduced (cf. [133], p. 99 - 111). Singleton beans are intended to be instantiated once per application and exist for the entire life time of the corresponding application[55]. If an application is distributed across multiple JVMs, one instance should be created within each JVM. The life cycle of a singleton SB is nearly the same as that of stateless SBs. For managing concurrent access to a singleton SB two types of concurrency management are considered, namely *Container Managed Concurrency* and *Bean Managed Concurrency*.

In order to support singleton SBs, the *mKernel* meta model would have to be adjusted at least to allow the identification of singleton SBs. Additionally, the concurrency settings would have to be reflected to allow their management. The preprocessing tool would have to be adjusted to extract the relevant information and to create a corresponding representation for

[54] November, 3rd 2008

[55] An instance is not required to survive container crashes.

the container plugin. Finally, the container plugin would require certain adjustments regarding information management, configuration options, and deployment preparation.

Internally, *mKernel* could make use of singleton SBs to avoid the violation of requirement *COR-SC* regarding the use of static fields. In this context, the information stored in static fields could be relocated into the state of an *mKernel*-specific singleton SB which could replace the *Management Context*.

Asynchronous Methods The execution of asynchronous calls is planned to be supported upon instances of SBs (cf. [133], p. 78 - 81). As underlying concept for realizing asynchronous invocations, the interface `java.-util.concurrent.Future` is used which also allows the cancellation of invocations.

The integration of this concept would not demand for major adjustments of *mKernel*. In this context, opportunity to cancel invocations would have to be reflected through the meta model, for example, through a corresponding property of a call representation.

In combination with singleton SBs, asynchronous methods could be used to perform quiescence management without the need for blocking interactions inside modules. In this context, the container plugin could initiate an asynchronous, blocking invocation upon a specific singleton SB inside each module which internally would wait for the receipt of a message from a specific JMS-topic. Each interaction which should be prevented from entering a quiescence region, would have to perform an invocation upon the singleton SB which would get blocked until the message is received and the blocking call returned. During release or destruction of a blocking or quiescent region, the expected message would be sent by the container plugin.

Global JNDI Access The EJB standard, version 3.1, defines a required name schema for the integration of SBs into the global namespace of a container (cf. [133], p. 76 - 78). This name schema might be used by *mKernel* to overcome the violation of *COR-SC* regarding mapped names.

No-interface View Local interactions between SB instances and their clients are planned to be supported without the need for an interface (cf. [133], p. 45, 46 and 117). In this context, a client might directly make use of the class of the target session bean instance as type for a reference. Clients are still required to use naming or DI for obtaining references. In contrast, they must not directly invoke the constructor of the target bean.

To integrate this facility into *mKernel*, the meta model would have to be adjusted to reflect method invocations not performed through an interface. Furthermore, SB proxies would have to be generated for no-interface views. This might be realized through sub-classing where proxies extend the corresponding session bean and overwrite all affected methods for invocation delegation purposes.

The goal of this thesis to design and realize a generic infrastructure for autonomic management of EJB-based enterprise systems can be considered as being met. Through this thesis and the corresponding projects it could be shown that the concept of *Component Orientation* and the vision of *Autonomic Computing* can provide a meaningful and promising foundation for addressing the complexity challenge of information technology in a holistic way at least for the considered application area.

A. Type Level Planning

```
import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import mKernel.ejb.EjbInterfaceType;
import mKernel.ejb.EjbModuleType;
import mKernel.ejb.EjbReferenceType;
import mKernel.ejb.JavaInterfaceType;
import mKernel.ejb.SessionBeanType;

/** Instances of this class can be used for
 * deployment planning based on JavaInterfaceTypes.
 * The provided methods of TypeLevelPlan can be
 * divided into four groups:
 *
 * Construction: An instance of TypeLevelPlan can be
 * created using the only provided constructor. This
 * constructor expects a boolean value indicating
 * whether planning should be performed for the
 * Managed Layer (true) or the Access Layer (false)
 * of a system.
 *
 * Requirements Definition: As foundation for
 * planning Java interface types, which are required
 * on the target layer, must be submitted to the plan
 * through invocations of the method
 * addJavaInterfaceType.
 *
 * Inspection: The methods belonging to this group
 * allow the inspection of a plan regarding required
 * Java interface types (getProvidedJavaInterfaces)
 * and their potential providers
 * (getJavaInterfaceTypeProviders). Furthermore,
 * required interfaces for the realization of the
 * plan (getRequiredReferences), as well as
 * alternatives for their connection can be inspected
 * (getConnectionAlternatives). Finally, the required
 * module types to deploy (getModuleTypesToDeploy)
 * and bean types can be requested
 * (getSessionBeanTypes).
 *
 * Unambiguity Establishment: A plan is only
 * unambiguous if there do not exist alternatives for
 * desired Java interface types or required interface
 * types anymore. Unambiguity can be reached through
 * the removal of alternative providers of desired
 * Java interface types
 * (removeJavaInterfaceTypeProvider) and providers
 * for required interface types
 * (removeReferenceTypeProvider). An invocation of
 * the method isUnambiguous returns a boolean value
 * indicating whether unambiguity is reached (true)
 * or not (false). */
public class TypeLevelPlan {

 /** Map for representing alternative providers
  * (value) for a desired Java interface type
  * (key). */
 private Map<JavaInterfaceType,
   Set<EjbInterfaceType>> p =
   new HashMap<JavaInterfaceType,
   Set<EjbInterfaceType>>();

 /** Map for representing alternative providers
  * (value) for required interface types (key). */
 private Map<EjbReferenceType,
   Set<EjbInterfaceType>> a = new
   HashMap<EjbReferenceType,
   Set<EjbInterfaceType>>();

 /** Field indicating if the plan is constructed for
  * the Managed Layer (true) or the Access Layer
  * (false). */
 private boolean m = false;

 /** Constructor for the creation of a plan.
  *
  * @param iniManagedLayer Parameter for the
  * initialization of the field m. */
 public TypeLevelPlan(
   boolean iniManagedLayer){
  this.m = iniManagedLayer;
}

/** Method can be used to submit a desired Java
 * interface type to the plan.
 *
 * @param j Desired Java interface type.
 * @return true if Java interface type could be
 * provided, false otherwise. */
public boolean addJavaInterfaceType(
  JavaInterfaceType j){
 if(this.p.containsKey(j)) return true;
 Set<EjbInterfaceType> r =
   new HashSet<EjbInterfaceType>();
 for(EjbInterfaceType ei:j.getEjbInterfaceTypes()){
  SessionBeanType s = ei.getSessionBeanType();
  if(s.getEjbModuleType().isManagedLayer() ==
    this.m){
   Map<EjbReferenceType, Set<EjbInterfaceType>>
     tmpA = new HashMap<EjbReferenceType,
     Set<EjbInterfaceType>>();
   tmpA.putAll(this.a);
   for(EjbReferenceType er :
     s.getEjbReferenceTypes()){
    tmpA = this.provideEjbReferenceType(er, tmpA);
    if(tmpA == null) break;
   }
   if(tmpA != null){
    this.a = tmpA;
    r.add(ei);
   }
  }
 }
 if(r.size() > 0){
  this.p.put(j, r);
 }
 return r.size() > 0;
}

/** Returns the set of all Java interface types
 * which are planned to be provided as targets of
 * the plan.
 *
 * @return Set of target JavaInterfaceTypes. */
public Set<JavaInterfaceType>
  getProvidedJavaInterfaces(){
 return this.p.keySet();
}
```

```

/** Delivers alternative provided interface types
 * for a submitted Java interface type.
 *
 * @param javaInterfaceType Relevant
 *    JavaInterfaceType.
 * @return EjbInterfaceTypes which might be used to
 * provide the JavaInterfaceType. */
public Set<EjbInterfaceType>
  getJavaInterfaceTypeProviders(
  JavaInterfaceType javaInterfaceType) {
 return this.p.get(javaInterfaceType);
}

/** Returns the set of all required reference types
 * for the current state of the plan.
 *
 * @return Set of target required EjbReferenceTypes.
 */
public Set<EjbReferenceType>
  getRequiredReferences(){
 return this.a.keySet();
}

/** Delivers the set of alternative providers for a
 * certain EjbReferenceType as reflected by the
 * current state of a plan.
 *
 * @param ref Target reference type.
 * @return Set of alternative providers for the
 *    required interface type. */
public Set<EjbInterfaceType>
  getConnectionAlternatives(EjbReferenceType ref){
 return this.a.get(ref);
}

/** Delivers all module types which would have to be
 * deployed based on the current state of the plan.
 *
 * @return Set of all module types to deploy. */
 public Set<EjbModuleType> getModuleTypesToDeploy(){
  Set<EjbModuleType> result =
    new HashSet<EjbModuleType>();
  for(SessionBeanType sbt:
    this.getSessionBeanTypes()){
   result.add(sbt.getEjbModuleType());
  }
```

```
    return result;
  }

 /** Delivers all session bean type which are planned
  * to provide at least one EjbInterfaceType as
  * part of the plan.
  *
  * @return Session bean types which are considered
  *   by the plan. */
 public Set<SessionBeanType> getSessionBeanTypes() {
  Set<SessionBeanType> r =
    new HashSet<SessionBeanType>();
  r.addAll(this.getSBTypesFrom(this.p.values()));
  r.addAll(this.getSBTypesFrom(this.a.values()));
  return r;
 }

 /** This method can be used to reach unambiguity of
  * a plan through removing an alternative provider
  * of a target Java interface type. The method does
  * not remove the provider if it is the last one
  * within the set of alternative providers.
  *
  * @param provider Interface type provider which
  *   should be removed.
  * @return true if provider could be removed, false
  *   if the invocation represented an attempt to
  *   remove the last provider from the set of
  *   potential providers. */
 public boolean removeJavaInterfaceTypeProvider(
   EjbInterfaceType provider){
  JavaInterfaceType jit =
    provider.getJavaInterfaceType();
  Collection<EjbInterfaceType> providerCollection =
    this.p.get(jit);
  if(providerCollection.size() > 1){
   providerCollection.remove(provider);
   this.recalculateReferences(
     provider.getSessionBeanType());
   return true;
  }else{
   return false;
  }
 }

 /** This method can be used to reach unambiguity of
  * a plan through removing an alternative provider
 * from the set of potential providers of an
 * EjbReferenceType required on Managed Layer. The
 * method does not remove the provider if it is the
 * last one within the set of alternative providers.
 *
 * @param requestor Reference type from which the
 *   provider should be removed.
 * @param provider Provider which should be removed.
 * @return true if provider could be removed, false
 *   if the invocation represented an attempt to
 *   remove the last provider from the set of
 *   potential providers. */
 public boolean removeReferenceTypeProvider(
   EjbReferenceType requestor,
   EjbInterfaceType provider){
  Collection<EjbInterfaceType> providerCollection =
    this.a.get(requestor);
  if(providerCollection.size() > 1){
   providerCollection.remove(provider);
   this.recalculateReferences(
     provider.getSessionBeanType());
   return true;
  }else{
   return false;
  }
}

/** Can be used to identify if unambiguity is
 * reached for the plan.
 *
 * @return true if unambiguity is reached, false
 * otherwise. */
public boolean isUnambiguous(){
 for(Collection s:this.p.values()){
  if(s.size() > 1){
   return false;
  }
 }
 for(Collection t:this.a.values()){
  if(t.size() > 1){
   return false;
  }
 }
 return true;
}

// Internally used methods
```

```

/** This method is used internally to fulfill the
 * interface demands of a session bean type which is
 * intended to be integrated into a plan.
 *
 * @param r Required interface type of the session
 *   bean type.
 * @param tmpA Temporary mapping from required
 *   interface types to potential providers.
 * @return New mapping from required interface types
 *   to potential providers. The return value will
 *   be null if no provider for r could be found
 *   recursively. */
private Map<EjbReferenceType, Set<EjbInterfaceType>>
  provideEjbReferenceType(EjbReferenceType r,
  Map<EjbReferenceType, Set<EjbInterfaceType>> tmpA){
 JavaInterfaceType j = r.getJavaInterfaceType();
 boolean success = false;
 for(EjbInterfaceType i:j.getEjbInterfaceTypes()){
  SessionBeanType s = i.getSessionBeanType();
  EjbModuleType m = s.getEjbModuleType();
  if(m.isManagedLayer() && ((i.isLocal() &&
    r.getEjbType().getEjbModuleType().equals(m)) ||
    !i.isLocal())){
   if(this.getSBTypesFrom(
     tmpA.values()).contains(s)){
    this.addEjbInterfaceProvider(r,i,tmpA);
    success = true;
   }else{
    Map<EjbReferenceType, Set<EjbInterfaceType>>
      subTempA = new HashMap<EjbReferenceType,
      Set<EjbInterfaceType>>();
    subTempA.putAll(tmpA);
    this.addEjbInterfaceProvider(r,i,subTempA);
    for(EjbReferenceType sr:
      s.getEjbReferenceTypes()){
     subTempA =
       this.provideEjbReferenceType(sr,subTempA);
     if(subTempA == null) break;
    }
    if(subTempA != null){
     tmpA = subTempA;
     success = true;
    }
   }
  }
 }
```

```
  if(success) return tmpA;
   return null;
 }

 /** Internally used method to integrate a potential
  * provider type for a required interface type into
  * a mapping data structure.
  *
  * @param requestor Required interface type.
  * @param provider New, alternative provider type.
  * @param tempAlternatives Existing mappings
  *   between required and provided interface types.
  */
 private void addEjbInterfaceProvider(
   EjbReferenceType requestor,
   EjbInterfaceType provider, Map<EjbReferenceType,
   Set<EjbInterfaceType>> tempAlternatives){
  Set<EjbInterfaceType> knownProviders =
    tempAlternatives.get(requestor);
  if(knownProviders == null){
   knownProviders = new HashSet<EjbInterfaceType>();
   tempAlternatives.put(requestor,knownProviders);
  }
  knownProviders.add(provider);
 }

 /** This methods extracts the set of SB types from a
  * collection of sets of EjbInterfaceTypes.
  *
  * @param ejbIfs Provided interface types from which
  *   the set of corresponding SB types should be
  *   extracted.
  * @return Set of extracted SB types.
  */
  private Set<SessionBeanType> getSBTypesFrom(
   Collection<Set<EjbInterfaceType>> ejbIfs){
  Set<SessionBeanType> sbts =
    new HashSet<SessionBeanType>();
  for(Set<EjbInterfaceType> c:ejbIfs){
   for (EjbInterfaceType e:c){
    sbts.add(e.getSessionBeanType());
   }
  }
  return sbts;
 }

 /** This method is used to revise the mapping of
```

```
   * required interface types to alternative providers
   * (this.a) recursively based on a SB type to remove
   * from the plan.
   *
   * @param s SessionBeanType which should be removed
   * from the plan. */
  private void recalculateReferences(
    SessionBeanType s){
   if(this.getSessionBeanTypes().contains(s)) return;
   Set<EjbInterfaceType> is =
     new HashSet<EjbInterfaceType>();
   for (EjbReferenceType r:s.getEjbReferenceTypes()){
    Set<EjbInterfaceType> rs = this.a.remove(r);
    if(rs != null){
     is.addAll(rs);
    }
   }
   for(EjbInterfaceType i:is){
    this.recalculateReferences(
      i.getSessionBeanType());
   }
  }

}
```

Listing A.1: Implementation of *Type Level* Plan

B. Deployment Level Planning

```
package mKernel.casestudy;

import java.util.HashSet;
import java.util.Set;

import mKernel.ejb.Container;
import mKernel.ejb.ContainerFactory;
import mKernel.ejb.EjbInterface;
import mKernel.ejb.EjbInterfaceType;
import mKernel.ejb.EjbModule;
import mKernel.ejb.EjbModuleType;
import mKernel.ejb.EjbReference;
import mKernel.ejb.EnterpriseBean;
import mKernel.ejb.JavaInterfaceType;
import mKernel.ejb.SessionBean;
import mKernel.ejb.SessionBeanType;

/** A deployment level plan can be used to realize
 * a type level plan. Therefore, an unambiguous type
 * level plan must
 * be submitted to the deployment level plan during
 * construction. An instance of DeploymentLevelPlan
 * is able to create, deploy, and start new modules
 * based on a Type Level plan. During deployment, it
 * performs compositional adaptation in accordance
 * with the TypeLevelPlan. A Deployment Level plan
 * does not take the current architecture of a
 * system into account. The realization of a plan
 * can be controlled through the three method create,
 * deploy, and start. These methods must be executed
 * consecutively.
 *
 * Additional management actions such as parameter
 * adaptation can be performed between the execution
 * of the three methods. For inspecting the elements
 * of a plan the remaining methods of the class can
 * be used. */
```

```
public class DeploymentLevelPlan{

 /** Type Level plan which builds the foundation
  * for an instance of DeploymentLevelPlan. */
 private TypeLevelPlan tp;

 /** EJB modules which are the targets of the plan.
  */
 private Set<EjbModule> ms =
   new HashSet<EjbModule>();

 /** Reference to the mKernel system. */
 private Container c =
   ContainerFactory.getNewContainer();

 /** Constructor for the creation of a plan.
  *
  * @param iniTp Parameter for the initialization of
  *    the field tp. */
 public DeploymentLevelPlan(TypeLevelPlan iniTp){
  if(!iniTp.isUnambiguous()){
   throw new IllegalArgumentException(
     "Only unambiguous plans can be realized.");
  }
  this.tp = iniTp;
 }

 /** The execution of this method leads to the
  * creation of EJB modules for all module types
  * being part of tp. Additionally, compositional
  * adaptation is performed. */
 public void create(){
  for(EjbModuleType m:
     this.tp.getModuleTypesToDeploy()){
   this.ms.add(this.c.createEjbModule(m));
  }
  for(SessionBean s:this.getSessionBeans()){
   for(EjbReference r:s.getEjbReferences()){
    EjbInterfaceType i =
      this.tp.getConnectionAlternatives(
    r.getEjbReferenceType()).iterator().next();
    boolean success = false;
    for(SessionBean sp:this.getSessionBeans()){
     for(EjbInterface ip:sp.getEjbInterfaces()){
      if(ip.getEjbInterfaceType().equals(i)){
       r.connectTo(ip);
       success = true;
```

```
        break;
       }
      }
      if (success) break;
     }
    }
   }
  }

  /** All modules affected by the plan (this.ms) are
   * deployed through an invocation of this method. */
  public void deploy(){
   for(EjbModule m: this.ms){
    m.deploy();
   }
  }

  /** All modules affected by the plan (this.ms) are
   * started through an invocation of this method. */
  public void start(){
   for(EjbModule em: this.ms){
    em.start();
   }
  }

  /** Grants access to the modules affected by the
   * plan. Might only be invoked meaningfully if at
   * least create() was invoked before.
   *
   * @return Set of affected modules. */
  public Set<EjbModule> getEjbModules(){
   return this.ms;
  }

  /** Grants access to the session beans affected by
   * the plan. The method might only be invoked
   * meaningfully if at least create() was invoked
   * before. The set needs not necessarily contain all
   * session beans which are part of a module returned
   * by getEjbModules(). In contrast, only session
   * beans are returned of which the corresponding
   * type is affected by the underlying type level
   * plan.
   *
   * @return Set of affected SBs. */
  public Set<SessionBean> getSessionBeans(){
   Set<SessionBeanType> sbTypes =
   this.tp.getSessionBeanTypes();
 Set<SessionBean> sbs = new HashSet<SessionBean>();
 for(EjbModule em: this.ms){
  for(EnterpriseBean eb:em.getEnterpriseBeans()){
   if(eb instanceof SessionBean){
    SessionBean sb = (SessionBean) eb;
    if(sbTypes.contains(sb.getSessionBeanType())){
     sbs.add(sb);
    }
   }
  }
 }
 return sbs;
}

/** Delivers the intended provider of a
 * JavaInterfaceType as required by the underlying
 * type level plan. The method might only be invoked
 * meaningfully if at least create() was invoked
 * before.
 *
 * @param jit Required Java interface type.
 * @return Intended provider within the managed
 *   system. */
public EjbInterface
  getEjbInterfaceForJavaInterfaceType(
  JavaInterfaceType jit){
 EjbInterfaceType eit = this.tp.
   getJavaInterfaceTypeProviders(jit).iterator().
   next();
 for(SessionBean sb:this.getSessionBeans()){
  for(EjbInterface ei:sb.getEjbInterfaces()){
   if(ei.getEjbInterfaceType().equals(eit)){
    return ei;
   }
  }
 }
 return null;
}

/** Delivers the Deployment Level counterpart of a
 * SB type affected by the underlying Type Level
 * Plan. The method might only be invoked
 * meaningfully if at least create() was invoked
 * before.
 *
 * @param type SB type for which the Deployment
```

```
   *    Level counterpart is required.
   * @return Deployment Level counterpart of type. */
  public SessionBean getSessionBeanForSessionBeanType(
    SessionBeanType type){
   for(SessionBean sb:this.getSessionBeans()){
    if(sb.getSessionBeanType().equals(type)){
     return sb;
    }
   }
   return null;
  }

  /** Delivers the Deployment Level counterpart of a
   * module type affected by the underlying type level
   * plan. The method might only be invoked
   * meaningfully if at least create() was invoked
   * before.
   *
   * @param type Module type for which the Deployment
   *    Level counterpart is required.
   * @return Deployment Level counterpart of type. */
  public EjbModule getEjbModuleForEjbModuleType(
    EjbModuleType type){
   for(EjbModule em:this.ms){
    if(em.getEjbModuleType().equals(type)) return em;
   }
   return null;
  }

}
```

Listing B.1: Implementation of *Deployment Level* Plan

C. Seamless Reconfiguration

```
import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import mKernel.ejb.Container;
import mKernel.ejb.ContainerFactory;
import mKernel.ejb.EjbInterface;
import mKernel.ejb.EjbModule;
import mKernel.ejb.EjbModuleType;
import mKernel.ejb.EjbReference;
import mKernel.ejb.EnterpriseBean;
import mKernel.ejb.JavaInterfaceType;
import mKernel.ejb.SessionBean;
import mKernel.ejb.SimpleEnvironmentEntry;
import mKernel.ejb.quiescence.HoldingReference;
import mKernel.ejb.quiescence.QuiescenceRegion;
import mKernel.ejb.quiescence.StateElement;

/** Instances of this class can be used to perform
 * the replacement a set of EJB modules seamlessly
 * regarding compositional adaptation. In this
 * context, only session beans of the replaced and
 * the replacing modules are considered. For
 * seamless reconfiguration it is required that no
 * quiescence region does exist within the managed
 * system. As preparation for execution, all
 * required references of the session beans which
 * are intended to replace an existing one must be
 * connected to provided interfaces of session
 * beans which do not belong to the modules to
 * replace. Furthermore, the replacing modules must
 * have been deployed and started. Finally, session
 * beans to replace and the replacing counterparts
 * must have the same EJB name. All configuration
 * demands besides compositional adaptation and
 * basic state transfers must be performed outside
 * the implementation.
 *
 * As foundation for reconfiguration it is assumed
 * that each affected Java interface type does only
 * exist once inside the replacing modules. For the
 * realization of a quiescent reconfiguration the
 * methods provided by this class should be
 * executed in the following order:
 *    1. defineRegion
 *    2. reachQuiescence
 *    3. transferTimers
 *    4. transferState
 *    5. replaceConnections
 *    6. finish
 *    7. destroyRegion
 * Between the execution of two subsequent methods
 * custom reconfiguration actions might be performed.
 */
public class SeamlessReconfigurator{

 /** Set of orignal modules which should be replaced.
  */
 private Set<EjbModule> om =
   new HashSet<EjbModule>();

 /** Set of orignal SBs which should be replaced. */
 private Set<SessionBean> os =
   new HashSet<SessionBean>();

 /** Set of replacing modules. */
 private Set<EjbModule> rm =
   new HashSet<EjbModule>();

 /** Set of session beans belonging to the replacing
  * modules (rm). */
 private Map<String,SessionBean> rs =
   new HashMap<String,SessionBean>();

 /** Map of Java interface types to replacing
  * provided interfaces. */
 private Map<JavaInterfaceType,EjbInterface> ri =
   new HashMap<JavaInterfaceType, EjbInterface>();

 /** Reference to the mKernel system. */
 private Container c =
   ContainerFactory.getNewContainer();
```

```

/** Quiescence region which builds the foundation
 * of seamless reconfiguration. */
private QuiescenceRegion q = null;

/** Constructor for the creation of a
 * reconfigurator.
 *
 * @param originalModules Parameter for the
 *   initialization of om.
 * @param replacingModules Parameter for the
 *   initialization of rm. */
public SeamlessReconfigurator(
  Set<EjbModule> originalModules,
  Set<EjbModule> replacingModules){
 this.om = originalModules;
 for(EjbModule em:this.om){
  for(EnterpriseBean ejb:em.getEnterpriseBeans()){
   SessionBean sb = (SessionBean)ejb;
   this.os.add(sb);
  }
 }
 this.rm = replacingModules;
 for(EjbModule em:this.rm){
  for(EnterpriseBean ejb:em.getEnterpriseBeans()){
   if(ejb instanceof SessionBean){
    SessionBean sb = (SessionBean)ejb;
    this.rs.put(sb.getEnterpriseBeanType().
      getEjbName(),sb);
    for(EjbInterface ei:sb.getEjbInterfaces()){
     this.ri.put(ei.getEjbInterfaceType().
       getJavaInterfaceType(),ei);
    }
   }
  }
 }
}

/** This method declares a quiescence region
 * covering the modules to replace and the replacing
 * ones. Furthermore, tracking is activaed for the
 * region. */
public void defineRegion(){
 Set<EjbModule> qm = new HashSet<EjbModule>();
 qm.addAll(this.om);
 qm.addAll(this.rm);
 this.q = c.declareQuiescenceRegion(qm, null, null);
```

```java
 this.q.track();
}

/** This method transfers the quiescence region to
 * the BLOCKING state and subsequently wait for the
 * transition to the QUIESCENT state. The return
 * value indicates if quiescence could successfully
 * be reached.
 *
 * @return true if quiescence has been reached
 *   successfully, false otherwise. */
public boolean reachQuiescence(){
 this.q.block();
 return this.q.waitForQuiescence();
}

/** Through an invocation of this method timers of
 * all stateless SBs to replace are transferred to
 * their replacing counterparts. */
 public void transferTimers(){
  for(SessionBean o:this.os){
   if(!o.getSessionBeanType().isStateful()){
    SessionBean r = this.rs.get(
      o.getSessionBeanType().getEjbName());
    r.setTimers(o.getTimers());
   }
  }
}

/** This method transfers all instances of stateful
 * SB to replace to instances of their replacing
 * counterparts. */
public void transferState(){
 for(HoldingReference ob:this.q.getReferences()){
  SessionBean s = rs.get(ob.getSessionBean().
    getType().getEjbName());
  HoldingReference rb = c.createReferenceTo(s);
  Map<String,Collection<StateElement>> st =
    ob.getState();
  Collection<StateElement> elems = st.get(
    ob.getSessionBean().getSessionBeanType().
    getFullyQualifiedEnterpriseBeanClassName());
  Map<String, Collection<StateElement>> newM =
    new HashMap<String,Collection<StateElement>>();
  newM.put(rb.getSessionBean().getSessionBeanType().
    getFullyQualifiedEnterpriseBeanClassName(),
    elems);
    rb.setState(newM);
    // The following three source code lines are
    // commented out, because they are specific to the
    // case study. Nevertheless, they are left in here
    // for the sake of completeness. The same result
    // could be reached outside this class. For this
    // purpose holding references would have to be
    // expose, for example, through a corresponding
    // method.
    // if(s.getSessionBeanType().getEjbName().
    //    equals("TxControllerBean")){
    //  rb.setFieldValue("fee", new Long(10));
    // }
    ob.replaceWith(rb);
  }
 }

 /** Incoming connections of SB to replace are
  * reconnected to the corresponding, replacing SBs
  * by this method. */
 public void replaceConnections(){
  for(SessionBean s:this.os){
   for(EjbInterface i:s.getEjbInterfaces()){
    for(EjbReference r:
      i.getConnectedEjbReferences()){
     if(!this.os.contains(r.getEnterpriseBean())){
      r.connectTo(this.ri.get(
        i.getEjbInterfaceType().
        getJavaInterfaceType()));
     }
    }
   }
  }
 }

 /** This method stops, undeploys, and destroys the
  * modules to replace. Furthermore, the quiescence
  * region is released. */
 public void finish(){
  for (EjbModule m:this.om){
   m.stop();
  }
  this.q.release();
  for (EjbModule m:this.om){
   m.undeploy();
   m.destroy();
  }
  }

  /** This method finally destroys the quiescence
   * region. */
  public void destroyRegion(){
   q.destroy();
  }

}
```

Listing C.1: Implementation of Seamless Reconfiguration

Bibliography

[1] "An architectural blueprint for autonomic computing. (Third Edition)", http://www-03.ibm.com/autonomic/pdfs/AC Blueprint White Paper V7.pdf, June 2005.

[2] Java Compiler Compiler (JavaCC) – The Java Parser Generator, https://javacc.dev.java.net/, July 2007.

[3] "Sun Java System Application Server 9.1 Reference Manual", Sun Microsystems, October 2007, http://docs.sun.com/app/docs/doc/819-3675.

[4] IBM WebSphere Application Server, http://www-306.ibm.com/software/webservers/appserv/was/, August 2008.

[5] Java Programming Assistant (Javassist), http://www.csg.is.titech.ac.jp/ chiba/javassist/, November 2008.

[6] JBoss Application Server, http://www.jboss.org/jbossas, August 2008.

[7] JOnAS – Java Open Application Server, http://jonas.objectweb.org/, December 2008.

[8] Oracle WebLogic Application Server, http://www.oracle.com/appserver, August 2008.

[9] Sun GlassFish Enterprise Server, http://www.sun.com/software/products/appsrvr, August 2008.

[10] The Apache Velocity Project,
http://velocity.apache.org, September 2008.

[11] The GlassFish Application Server,
https://glassfish.dev.java.net, November 2008.

[12] T. Abdellatif and A. Danes, "JMX-Based Autonomic Management of J2EE Servers", *International Transactions on Systems Science and Applications (ITSSA)*, vol. 2, no. 3, pp. 289–295, September 2006.

[13] J. P. A. Almeida, M. Van Sinderen, and L. Nieuwenhuis, "Transparent Dynamic Reconfiguration for CORBA", in *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA '01)*. IEEE Computer Society, 2001, pp. 197–207.

[14] M. AlSharif, W. P. Bond, and T. Al-Otaiby, "Assessing the Complexity of Software Architecture", in *Proceedings of the 42nd Annual Southeast Regional Conference (ACM-SE 42)*. ACM Press, 2004, pp. 98–103.

[15] H. Balzert, "Lehrbuch der Software-Technik – Software-Entwicklung", ser. Lehrbücher der Informatik. Spektrum Akademischer Verlag, December 2000, vol. 1.

[16] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, "Software Complexity and Maintenance Costs", *Communications of the ACM*, vol. 36, no. 11, pp. 81–94, November 1993.

[17] R. Barret, P. P. Maglio, E. Kandogan, and J. Bailey, "Usable Autonomic Computing Systems: The Administrator's Perspective", in *Proceedings of the First International Conference on Autonomic Computing (ICAC 2004)*. IEEE Computer Society, 2004, pp. 18–26.

[18] R. Barrett, E. Kandogan, P. P. Maglio, E. M. Haber, L. A. Takayama, and M. Prabaker, "Field Studies of Computer System Administra-

tors: Analysis of System Management Tools and Practices", in *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (CSCW '04).* ACM Press, November 2004, pp. 388–395.

[19] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim, "Jass - Java with Assertions", *Electronic Notes in Theoretical Computer Science,* vol. 55, no. 2, pp. 103–117, October 2001.

[20] A. F. Benner, P. K. Pepeljugoski, and R. J. Recio, "A Roadmap to 100G Ethernet at the enterprise data center," *IEEE Communications Magazine,* vol. 45, no. 11, pp. 10–17, November 2007.

[21] K. H. Bennett and V. T. Rajlich, "Software Maintenance and Evolution: a Roadmap", in *Proceedings of the Conference on The Future of Software Engineering (ICSE '00).* ACM Press, 2000, pp. 73–87.

[22] P. V. Biron and A. Malhotra, *XML Schema Part 2: Datatypes Second Edition,* http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/, World Wide Web Consortium (W3C) Std., October 2004.

[23] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. M. Costa, H. A. Duran-Limon, T. Fitzpatrick, L. Johnston, R. S. Moreira, N. Parlavantzas, and K. B. Saikoski, "The Design and Implementation of Open ORB 2", *IEEE Distributed Systems Online,* vol. 2, no. 6, 2001.

[24] G. S. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas, "Reflection, Self-Awareness and Self-Healing in OpenORB", in *Proceedings of the First Workshop on Self-Healing Systems (WOSS '02).* ACM Press, November 2002, pp. 9–14.

[25] B. W. Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Computer,* vol. 21, no. 5, pp. 61–72, May 1988.

[26] S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, A. Mos, N. De Palma, V. Quéma, and J.-B. Stefani, "Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters", in *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005).* IEEE Computer Society, October 2005, pp. 13–24.

[27] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, *Extensible Markup Language (XML) 1.0 (Fifth Edition),* http://www.w3.org/TR/xml/, World Wide Web Consortium (W3C) Std., November 2008.

[28] A. W. Brown and K. Short, "On Components and Objects: The Foundations of Component-Based Development", in *Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST '97).* IEEE Computer Society, June 1997, pp. 112–124.

[29] A. B. Brown, A. Keller, and J. L. Hellerstein, "A Model of Configuration Complexity and its Application to a Change Management System", in *Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management 2005 (IM 2005).* IEEE Computer Society, May 2005, pp. 631–644.

[30] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stal, and C. A. Szyperski, "What characterizes a (software) component?" *Software - Concepts and Tools,* vol. 19, no. 1, pp. 49–56, June 1998.

[31] J. Bruhn, S. Kaffille, and G. Wirtz, "Hierarchical Scheduling for State-Based Services", in *Proceedings of the 2004 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2004),* vol. 1. CSREA Press, June 2004, pp. 179–185.

[32] J. Bruhn, C. Niklaus, T. Vogel, and G. Wirtz, “Comprehensive Support for Management of Enterprise Applications”, in *Proceedings of the 6th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2008).* IEEE Computer Society, March 2008, pp. 755 – 762.

[33] J. Bruhn and G. Wirtz, “DyName: Enhanced Naming for EJB”, in *Proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2007)*, vol. 1. CSREA Press, June 2007, pp. 17–23.

[34] J. Bruhn and G. Wirtz, “mKernel: A manageable Kernel for EJB-based Systems”, in *Proceedings of the First International Conference on Autonomic Computing and Communication Systems (Autonomics 2007).* ACM Press, October 2007, pp. 1–10.

[35] J. Bruhn and G. Wirtz, “Using Contracts for Self-Management”, *Communications of SIWN*, vol. 4, pp. 110–115, June 2008.

[36] E. Bruneton, T. Coupaye, and J.-B. Stefani, “Recursive and Dynamic Software Composition with Sharing”, in *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP’02)*, June 2002.

[37] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, “An Open Component Model and its Support in Java”, *Lecture Notes in Computer Science*, vol. 3054/2004, pp. 7–22, May 2004.

[38] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, “The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems”, *Software – Practice and Experience (SP&E)*, vol. 36, no. 11-12, pp. 1257–1284, September 2006.

[39] E. Bruneton, T. Coupaye, and J.-B. Stefani, “The Fractal Project – Website”, http://fractal.objectweb.org/, December 2008.

[40] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, “Towards a taxonomy of software change: Research Articles”, *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 5, pp. 309–332, September 2005.

[41] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, “An overview of JML tools and applications”, *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7, no. 3, pp. 212–232, June 2005.

[42] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox, “JAGR: an autonomous self-recovering application server”, in *Proceedings of the Autonomic Computing Workshop 2003.* IEEE Computer Society, 2003, pp. 168 – 177.

[43] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan, “Types of software evolution and software maintenance”, *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, February 2001.

[44] X. Chen, “Extending RMI to Support Dynamic Reconfiguration of Distributed Systems”, in *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS’02).* IEEE Computer Society, November 2002, pp. 401–408.

[45] Y. Cheon and G. T. Leavens, “A Simple and Practical Approach to Unit Testing: The JML and JUnit Way”, in *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP ’02).* Springer-Verlag, 2002, pp. 231–255.

[46] Y. Cheon and C. E. Rubio-Medrano, “Random Test Data Generation for Java Classes Annotated with JML Specifications”, in *Proceedings*

of the 2007 International Conference on Software Engineering Research and Practice (SERP'07). CSREA Press, 2007, pp. 385–391.

[47] S. Cheung and V. Matena, *Java Transaction API (JTA)*, http://java.sun.com/javaee/technologies/jta/, Sun Microsystems Inc. Std., Rev. 1.1, November 2002.

[48] S. Chiba, "Load-Time Structural Reflection in Java", in *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP '00)*. Springer-Verlag, June 2000, pp. 313–336.

[49] S. Chiba and M. Nishizawa, "An easy-to-use toolkit for efficient Java bytecode translators", in *Proceedings of the 2nd international conference on Generative programming and component engineering (GPCE '03)*. Springer-Verlag, September 2003, pp. 364–376.

[50] T. Coupaye and J. Estublier, "Foundations of Enterprise Software Deployment", in *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR 2000)*. IEEE Computer Society, February 2000, pp. 65–73.

[51] I. Crnkovic, "Component-based Software Engineering – New Challenges in Software Development", in *Proceedings of the 25th International Conference on Information Technology Interfaces (ITI 2003)*. IEEE Computer Society, June 2003, pp. 9–18.

[52] I. Crnkovic, M. Chaudron, and S. Larsson, "Component-Based Development Process and Component Lifecycle", in *Proceedings of the International Conference on Software Engineering Advances (ICSEA '06)*. IEEE Computer Society, October 2006, pp. 44–53.

[53] E. Dashofy, H. Asuncion, S. Hendrickson, G. Suryanarayana, J. Georgas, and R. Taylor, "ArchStudio 4: An Architecture-Based

Meta-Modeling Environment", in *Companion of the 29th International Conference on Software Engineering (ICSE 2007)*. IEEE Computer Society, May 2007, pp. 67–68.

[54] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "A Highly-Extensible, XML-Based Architecture Description Language", in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*. IEEE Computer Society, August 2001, pp. 103–112.

[55] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "An Infrastructure for the Rapid Development of XML-based Architecture Description Languages", in *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM Press, May 2002, pp. 266–276.

[56] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "Towards Architecture-based Self-Healing Systems", in *Proceedings of the first Workshop on Self-healing Systems (WOSS '02)*. ACM Press, November 2002, pp. 21–26.

[57] A. Dearle, "Software Deployment, Past, Present and Future", in *Proceedings of the International Conference on Software Engineering*. IEEE Computer Society, May 2007, pp. 269–284.

[58] L. DeMichiel and M. Keith, *JSR 220: Enterprise JavaBeans, Version 3.0 – EJB Core Contracts and Requirements,* http://jcp.org/aboutJava/communityprocess/final/jsr220, Sun Microsystems Std., May 2006.

[59] L. DeMichiel and M. Keith, *JSR 220: Enterprise JavaBeans, Version 3.0 – Java Persistence API,* http://jcp.org/aboutJava/communityprocess/final/jsr220, Sun Microsystems Std., May 2006.

[60] L. DeMichiel and M. Keith, *JSR 220: Enterprise JavaBeansTM,Version 3.0 – EJB 3.0 Simplified API,* http://jcp.org/aboutJava/communityprocess/final/jsr220, Sun Microsystems Std., May 2006.

[61] L. G. DeMichiel, L. Ümit Yalcinalp, and S. Krishnan, *JSR 19: Enterprise JavaBeans Specification, Version 2.0,* http://jcp.org/aboutJava/communityprocess/final/jsr019, Sun Microsystems Std., August 2001.

[62] Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung, "Self-Managing Systems: A Control Theory Foundation", in *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05).* IEEE Computer Society, April 2005, pp. 441–448.

[63] E. W. Dijkstra, "The humble Programmer", *Communications of the ACM,* vol. 15, no. 10, pp. 859–866, October 1972.

[64] J. Dochez, *JSR-88: Java Enterprise Edition 5 Deployment API Specification, Version 1.2,* http://jcp.org/aboutJava/communityprocess/mrel/jsr088, Sun Microsystems Std., May 2006.

[65] H. A. Duran-Limon, G. S. Blair, and G. Coulson, "Adaptive Resource Management in Middleware: A Survey", *IEEE Distributed Systems Online 1541-4922,* vol. 5, no. 7, July 2004.

[66] D. Eastlake and P. Jones, *RFC 3174: US Secure Hash Algorithm 1 (SHA1),* http://www.ietf.org/rfc/rfc3174.txt, Internet Engineering Task Force (IETF) Std., September 2001.

[67] M. Fowler, *Patterns of Enterprise Application Architecture.* Addison Wesley, November 2002.

[68] A. G. Ganek and T. A. Corbi, "The dawning of the autonomic computing era", *IBM Systems Journal*, vol. 42, no. 1, pp. 5–18, 2003.

[69] A. G. Ganek, C. P. Hilkner, J. W. Sweitzer, B. Miller, and J. L. Hellerstein, "The Response to IT Complexity: Autonomic Computing", in *Proceedings of the Third IEEE International Symposium on Network Computing and Applications (NCA'04)*. IEEE Computer Society, 2004, pp. 151–157.

[70] D. Garlan and D. E. Perry, "Introduction to the Special Issue on Software Architecture", *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 269–274, April 1995.

[71] D. Gupta, P. Jalote, and G. Barua, "A Formal Framework for On-line Software Version Change", *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 120–131, February 1996.

[72] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks", in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14)*. ACM Press, November 2006, pp. 175–185.

[73] G. Halprin, "The Work Flow of System Administration", in *Proceedings of the 6th Annual Conference of the System Administrators Guild of Australia (SAGE-AU '98)*, July 1998.

[74] G. Hamilton, *JavaBeans Specification 1.01*, http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html, Sun Microsystems Std., August 1998.

[75] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, *Java Message Service, Version 1.1*, http://java.sun.com/products/jms/, Sun Microsystems Std., April 2002.

[76] S. Hariri, B. Khargharia, H. Chen, J. Yang, Y. Zhang, M. Parashar, and H. Liu, "The Autonomic Computing Paradigm", *Cluster Computing,* vol. 9, no. 1, pp. 5–17, 2006.

[77] W. Hasselbring and R. Reussner, "Toward Trustworthy Software Systems", *Computer,* vol. 39, no. 4, pp. 91–92, April 2006.

[78] J. He, X. Li, and Z. Liu, "Component-Based Software Engineering The Need to Link Methods and Their Theory", in *Proceedings of the Second International Colloquium on Theoretical Aspects of Computing (ICTAC 2005).* Springer-Verlag, October 2005, pp. 70–95.

[79] J. Hillman and I. Warren, "Quantitative Analysis of Dynamic Reconfiguration Algorithms", in *Proceedings of the International Conference on Design, Analysis and Simulation of Distributed (DASD 2004),* April 2004.

[80] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology",
http://www.research.ibm.com/autonomic/manifesto/
autonomic_computing.pdf, October 2001, IBM Corporation.

[81] H. Hrasna, *JSR-77: Java 2 Platform, Enterprise Edition Management Specification,*
http://jcp.org/aboutJava/communityprocess/mrel/jsr077, Sun Microsystems Std., Rev. 1.1, May 2006.

[82] G. Huang, "Post-Development Software Architecture", *ACM SIGSOFT Software Engineering Notes,* vol. 32, no. 5, pp. 1–9, September 2007.

[83] G. Huang, H. Mei, and Q. xiang Wang, "Towards Software Architecture at Runtime", *ACM SIGSOFT Software Engineering Notes,* vol. 28, no. 2, p. 8, March 2003.

[84] G. Huang, H. Mei, and F.-Q. Yang, “Runtime recovery and manipulation of software architecture of component-based systems”, *Automated Software Engineering*, vol. 13, no. 2, pp. 257–281, April 2006.

[85] M. C. Huebscher and J. A. McCann, “A survey of Autonomic Computing—Degrees, Models, and Applications”, *ACM Computing Surveys*, vol. 40, no. 3, pp. 1–28, August 2008.

[86] M. T. Ibrahim, R. Telford, P. Dini, P. Lorenz, N. Vidovic, and R. Anthony, “Self-Adaptability and Man-in-the-Loop: A Dilemma in Autonomic Computing Systems”, in *Proceedings of the 15th International Workshop on Database and Expert Systems Applications (DEXA’04)*. IEEE Computer Society, August 2004, pp. 722–729.

[87] *IEEE Standard Glossary of Software Engineering Terminology (610.12-1990)*, Institute of Electrical and Electronics Engineers Inc. (IEEE) Std., Rev. 2002, January 2003.

[88] I. Jacobson, G. Booch, and J. Rumbaugh, “The Unified Process”, *IEEE Software*, vol. 16, no. 3, pp. 96–102, May/June 1999.

[89] S. Jansen and S. Brinkkemper, “Definition and Validation of the Key process of Release, Delivery and Deployment for Product Software Vendors: turning the ugly duckling into a swan”, in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM ’06)*. IEEE Computer Society, September 2006, pp. 166–175.

[90] Z. Jarir, P.-C. David, and T. Ledoux, “Dynamic Adaptability of Services in Enterprise JavaBeans Architecture”, in *Proceedings of the 7th International Workshop on Component-Oriented Programming (ECOOP-WCOP’2002)*, June 2002.

[91] E. Jendrock, J. Ball, D. Carson, I. Evans, S. Fordin, and K. Haase, *The Java EE 5 Tutorial*, 3rd ed. Addison Wesley, February 2007.

[92] A. Keller, J. L. Hellerstein, K.-L. Wu, and V. Krishnan, "The CHAMPS System: Change Management with Planning and Scheduling", in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS 2004)*. IEEE Computer Society, April 2004, pp. 395–408.

[93] J. O. Kephart, "Research Challenges of Autonomic Computing", in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM Press, May 2005, pp. 15–22.

[94] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing", *Computer Magazine*, vol. 36, no. 1, pp. 41–50, January 2003.

[95] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming", in *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*. Springer-Verlag, June 1997, pp. 220–242.

[96] F. Kon and R. H. Campbell, "Dependence Management in Component-Based Distributed Systems", *IEEE Concurrency*, vol. 8, no. 1, pp. 1–11, January-March 2000.

[97] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware", *Communications of ACM*, vol. 45, no. 6, pp. 33–38, June 2002.

[98] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB", in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*. Springer-Verlag, April 2000, pp. 121–143.

[99] J. Kramer and J. Magee, “The Evolving Philosophers Problem: Dynamic Change Management”, *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, November 1990.

[100] K.-K. Lau and Z. Wang, “Software Component Models”, *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 709–724, October 2007.

[101] M. Leclercq, A. E. Ozcan, V. Quema, and J.-B. Stefani, “Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset”, in *Proceedings of the 29th International Conference on Software Engineering (ICSE ’07)*. IEEE Computer Society, May 2007, pp. 209–219.

[102] J. Lee, K. Siau, and S. Hong, “Enterprise integration with ERP and EAI”, *Communications of ACM*, vol. 46, no. 2, pp. 54–60, February 2003.

[103] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, “Metrics and Laws of Software Evolution - The Nineties View”, in *Proceedings of the 4th International Symposium on Software Metrics (METRICS ’97)*. IEEE Computer Society, November 1997, p. 20.

[104] M. Lehman, “Programs, life cycles, and laws of software evolution”, *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, September 1980.

[105] P. Lin, A. MacArthur, and J. Leaney, “Defining Autonomic Computing: A Software Engineering Perspective”, in *Proceedings of the 2005 Australian Software Engineering Conference (ASWEC’05)*. IEEE Computer Society, March 2005, pp. 88–97.

[106] P. Maes, “Concepts and experiments in computational reflection”, in *Proceedings of the ACM SIGPLAN International Conference on*

Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '87). ACM Press, October 1987, pp. 147–155.

[107] Q. H. Mahmoud, "Using Assertions in Java Technology", http://java.sun.com/developer/technicalArticles/JavaLP/assertions/, June 2005.

[108] S. Mamone, "The IEEE standard for software maintenance", *SIGSOFT Software Engineering Notes*, vol. 19, no. 1, pp. 75–76, January 1994.

[109] V. Matena and M. Hapner, *Enterprise JavaBeans Specification, v1.1,* http://java.sun.com/products/ejb/docs.html, Sun Microsystems Std., December 1999.

[110] J. A. McCann and M. C. Huebscher, "Evaluation Issues in Autonomic Computing", in *Proceedings of the International Workshop on Agents and Autonomic Computing and Grid Enabled Virtual Organizations (AAC-GEVO'2004).* Springer-Verlag, September 2004, pp. 597–608.

[111] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng, "Composing Adaptive Software", *Computer*, vol. 37, no. 7, pp. 56–64, July 2004.

[112] N. Medvidovic, E. M. Dashofy, and R. N. Taylor, "Moving Architectural Description from Under the Technology Lamppost", *Information and Software Technology*, vol. 49, no. 1, pp. 12–31, January 2007.

[113] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, January 2000.

[114] H. Mei and G. Huang, "PKUAS: An Architecture-based Reflective Component Operating Platform", in *Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2004)*. IEEE Computer Society, May 2004, pp. 163–169.

[115] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in Software Evolution", in *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE '05)*. IEEE Computer Society, September 2005, pp. 13–22.

[116] B. Meyer, *Object-oriented Software Construction (2nd ed.)*. Prentice-Hall, Inc., 1997, ch. Design by Contract: Building reliable Software, pp. 331–410.

[117] K. Moazami-Goudarzi and J. Kramer, "Maintaining Node Consistency in the Face of Dynamic Change", in *Proceedings of the 3rd International Conference on Configurable Distributed Systems (ICCDS '96)*. IEEE Computer Society, May 1996, pp. 62–69.

[118] G. E. Moore, "Cramming more components onto integrated circuits", *Electronics Magazine*, vol. 38, no. 8, pp. 114–117, April 1965.

[119] F. Niessink and H. van Vliet, "Software Maintenance from a Service Perspective", *Journal of Software Maintenance: Research and Practice*, vol. 12, no. 2, pp. 103–120, March/April 2000.

[120] *The Common Object Request Broker: Architecture and Specification*, http://www.omg.org/docs/formal/98-07-01.pdf, Object Management Group (OMG) Std., Rev. 2.2, February 1998.

[121] *The Common Object Request Broker: Architecture and Specification*, http://www.omg.org/docs/formal/99-10-07.pdf, Object Management Group (OMG) Std., Rev. 2.3.1, October 1999.

[122] *CORBA CosNaming Service Specification,* http://www.omg.org/docs/formal/00-06-19.pdf, Object Management Group (OMG) Std., Rev. 1.0, April 2000.

[123] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based Runtime Software Evolution", in *Proceedings of the 20th international Conference on Software Engineering (ICSE '98).* IEEE Computer Society, April 1998, pp. 177–186.

[124] M. Parashar and S. Hariri, "Autonomic Computing: An Overview", *Lecture Notes in Computer Science (LNCS): Unconventional Programming Paradigms,* vol. 3566/2005, pp. 257–269, 2005.

[125] D. L. Parnas, "Software Aging", in *Proceedings of the 16th International Conference on Software Engineering (ICSE '94).* IEEE Computer Society, May 1994, pp. 279–287.

[126] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture", *SIGSOFT Software Engineering Notes,* vol. 17, no. 4, pp. 40–52, October 1992.

[127] V. T. Rajlich and K. H. Bennett, "A Staged Model for the Software Life Cycle", *Computer,* vol. 33, no. 7, pp. 66–71, July 2000.

[128] D. J. Reifer, V. R. Basili, B. W. Boehm, and B. Clark, "COTS-Based Systems - Twelve Lessons Learned about Maintenance", in *Proceedings of the 3rd International Conference on COTS-Based Software Systems (ICCBSS 2004).* Springer-Verlag, February 2004, pp. 137–145.

[129] L. Rosa, L. Rodrigues, and A. Lopes, "A Framework to Support Multiple Reconfiguration Strategies", in *Proceedings of the First International Conference on Autonomic Computing and Communication Systems (Autonomics 2007).* ACM Press, October 2007.

[130] W. W. Royce, “Managing the Development of Large Software Systems: Concepts and Techniques”, in *Proceedings of the 9th international conference on Software Engineering (ICSE ’87)*. IEEE Computer Society, 1987, pp. 328–338.

[131] M. J. Rutherford, K. M. Anderson, A. Carzaniga, D. Heimbigner, and A. L. Wolf, “Reconfiguration in the Enterprise JavaBean Component Model”, in *Proceedings of the IFIP/ACM Working Conference on Component Deployment (CD ’02)*. Springer-Verlag, June 2002, pp. 67–81.

[132] S. M. Sadjadi and P. K. McKinley, “A Survey of Adaptive Middleware”, Computer Science and Engineering, Michigan State University, Tech. Rep. MSU-CSE-03-35, December 2003.

[133] K. Saks, *JSR 318: Enterprise JavaBeans, Version 3.1*, http://jcp.org/aboutJava/communityprocess/pr/jsr318, Sun Microsystems Std., Rev. Public Review, November 2008.

[134] M. Salehie and L. Tahvildari, “Autonomic Computing: Emerging Trends and Open Problems”, in *Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software (DEAS ’05)*. ACM Press, May 2005, pp. 1–7.

[135] SAP AG, “SAP Upgrade Factory Ihr schneller, einfacher Umstieg auf SAP ERP 6.0 zum Festpreis”, http://www.sap.com/germany/media/mc_402/50082803.pdf, April 2008.

[136] D. C. Schmidt and C. Cleel, “Applying Patterns to Develop Extensible ORB Middleware”, *IEEE Communications Magazine*, vol. 37, no. 4, pp. 54–63, April 1999.

[137] J.-G. Schneider and O. Nierstrasz, *Software Architectures: Advances and Applications*. Springer-Verlag, November 1999, ch. Components, Scripts and Glue, pp. 13–25.

and Applications. Springer-Verlag, November 1999, ch. Components, Scripts and Glue, pp. 13–25.

[138] B. Shannon, *Java 2 Platform Enterprise Edition Specification, v 1.3,* http://java.sun.com/j2ee/1.3/download.html, Sun Microsystems Std., July 2001.

[139] B. Shannon, *Java 2 Platform Enterprise Edition Specification, v1.4,* http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf, Sun Microsystems Std., November 2003.

[140] B. Shannon, *JSR 244: Java Platform, Enterprise Edition (Java EE) Specification, v5,* http://jcp.org/aboutJava/communityprocess/final/jsr244, Sun Microsystems Std., April 2006.

[141] Software Engineering Institute, Carnegie Mellon University, "How Do You Define Software Architecture?", http://www.sei.cmu.edu/architecture/definitions.html, June 2008.

[142] I. Sommerville, *Software Engineering*, 6th ed. Addison Wesley, 2001.

[143] R. Sterritt, "Towards Autonomic Computing: Effective Event Management", in *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW '02).* IEEE Computer Society, December 2002, pp. 40–47.

[144] R. Sterritt and D. Bustard, "Autonomic Computing – a Means of Achieving Dependability?" in *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS 2003).* IEEE Computer Society, April 2003, pp. 40–47.

[145] R. Sterritt and D. Bustard, “Towards an Autonomic Computing Environment”, in *Proceedings of the 14th International Workshop on Database and Expert Systems Applications (DEXA ’03)*. IEEE Computer Society, September 2003, pp. 694–698.

[146] R. Sterritt and M. Hinchey, “Autonomic Computing – Panacea or Poppycock?” in *Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS’05)*. IEEE Computer Society, April 2005, pp. 535–539.

[147] *JAR File Specification*, http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html, Sun Microsystems Std., 2003.

[148] *Java Naming and Directory Interface (JNDI)*, http://java.sun.com/products/jndi/, Sun Microsystems Inc. Std., July 1999.

[149] E. B. Swanson, “The dimensions of maintenance”, in *Proceedings of the 2nd International Conference on Software Engineering (ICSE ’76)*. ACM Press, October 1976, pp. 492–497.

[150] R. S. Swarz and J. K. DeRosa, “A Framework for Enterprise Systems Engineering Processes”, MITRE Corporation, Tech. Rep., November 2006, http://www.mitre.org/work/tech_papers/tech_papers_06/06_1163/.

[151] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley Professional, November 2002.

[152] C. Szyperski, “Component Technology: What, Where, and How?” in *Proceedings of the 25th International Conference on Software Engineering (ICSE ’03)*. IEEE Computer Society, May 2003, pp. 684–693.

[153] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn, *XML Schema Part 1: Structures Second Edition*, http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/, World Wide Web Consortium (W3C) Std., October 2004.

[154] S. Vajjhala and J. Fialli, *JSR 222: Java Architecture for XML Binding (JAXB) 2.0*, jcp.org/aboutJava/communityprocess/final/jsr222, Sun Microsystems Std., April 2006.

[155] Y. Vandewoude and Y. Berbers, "Component state mapping for runtime evolution", in *In Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. CSREA Press, June 2005, pp. 230–236.

[156] T. Vogel, "Design and Implementation of Autonomous Reconfiguration Procedures for EJB-based Enterprise Applications", Diploma thesis, Otto-Friedrich-University Bamberg, 2008.

[157] T. Vogel, J. Bruhn, and G. Wirtz, "Autonomous Reconfiguration Procedures for EJB-based Enterprise Applications", in *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering 2008 (SEKE'08)*. Knowlegde Systems Institute, July 2008, pp. 48–53.

[158] Q. Wang, F. Chen, H. Mei, and F. Yan, "Using Application Server To Support Online Evolution", in *Proceedings of International Conference on Software Maintenance (ICSM'02)*. IEEE Computer Society, October 2002, pp. 131–140.

[159] R. Want, T. Pering, and D. Tennenhouse, "Comparing autonomic and proactive computing", *IBM Systems Journal*, vol. 42, no. 1, pp. 129–135, January 2003.

[160] I. Warren and I. Sommerville, "A Model for Dynamic Configuration Which Preserves Application Integrity", in *Proceedings of the 3rd*

International Conference on Configurable Distributed Systems (ICCDS '96). IEEE Computer Society, May 1996, pp. 81–88.

[161] M. Wermelinger, "A Hierarchic Architecture Model for Dynamic Reconfiguration", in *Proceedings of the Second International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97).* IEEE Computer Society, May 1997, pp. 243–254.

[162] J. White, D. C. Schmidt, and A. Gokhale, "Simplifying Autonomic Enterprise Java Bean Applications Via Model-Driven Development: A Case Study", in *Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS / UML 2005).* Springer-Verlag, October 2005, pp. 601–615.

[163] J. White, D. C. Schmidt, and A. Gokhale, "Simplifying autonomic enterprise Java Bean applications via model-driven engineering and simulation", *Software and Systems Modeling*, vol. 7, no. 1, pp. 3–23, February 2008.

[164] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart, "An Architectural Approach to Autonomic Computing", in *Proceedings of the First International Conference on Autonomic Computing 2004 (ICAC 2004).* IEEE Computer Society, May 2004, pp. 2–9.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und ohne die Hilfe eines Promotionsberaters angefertigt habe. Dabei habe ich keine anderen Hilfsmittel als die im Literaturverzeichnis genannten benutzt. Alle aus der Literatur wörtlich oder sinngemäß entnommenen Stellen sind als solche kenntlich gemacht.

Weder diese Arbeit noch wesentliche Teile derselben wurden einer anderen Prüfungsbehörde zur Erlangung des Doktorgrades vorgelegt.

Die Arbeit wurde bisher noch nicht in ihrer Ganzheit publiziert. Alle bereits veröffentlichten Beiträge, auf denen diese Arbeit basiert, sind im Literaturverzeichnis unter [31], [32], [33], [34], [35] und [157] angegeben.

UNIVERSITY OF BAMBERG PRESS

Seit einigen Jahrzehnten ist eine fortschreitende Durchdringung immer weiterer Bereiche des menschlichen Lebens mit IT-Systemen festzustellen. Hiermit verbunden ist ein massives Ansteigen der inhärenten Komplexität dieser Systeme. Ein für die Zukunft zu erwartender weiterer Komplexitätsanstieg erfordert eine explizite Adressierung um die Weiterentwicklung der IT nicht zu behindern.

Das Konzept der Komponentenorientierung beinhaltet Ansätze zur Komplexitätsreduktion für die Entwicklung und Konfiguration von Software durch funktionale Dekomposition. Mit der Vision des Autonomic Computing existiert ein Ansatz zur Komplexitätsbewältigung für Betrieb und Wartung von Softwaresystemen durch die Übertragung von Aufgaben zur Feinsteuerung eines Systems auf das verwaltete System selbst.

Diese Arbeit stellt eine realistische Infrastruktur für die autonome Verwaltung von Geschäftsanwendungen vor. Basierend auf einem etablierten Komponentenstandard wird eine Plattform vorgestellt die autonomen Verwaltungseinheiten eine ganzheitliche und modellbasierte Verwaltungsschnittstelle zur Informationsversorgung und zur Systemanpassung bietet. Die vorgestellte Plattform unterstützt den eingesetzten Komponentenstandard vollständig. Gleichzeitig werden keine Zusatzanforderungen an die Entwicklung von verwalteten Komponenten gestellt. Somit ist die Herstellung der Verwaltbarkeit von Softwarekomponenten nicht mit einem Komplexitätsanstieg verbunden.

ISBN 978-3-923507-49-8
ISSN 1867-7401
18,00 Euro