Design and Implementation of Autonomous Reconfiguration Procedures for EJB-based Enterprise Applications

Diplomarbeit

im Studiengang Wirtschaftsinformatik
der Fakultät Wirtschaftsinformatik
und Angewandte Informatik der
Otto-Friedrich-Universität Bamberg

Vorgelegt von:
Thomas Vogel

Referent:
Prof. Dr. Guido Wirtz
Datum:
07.07.2008
# Contents

1 Introduction 1  
1.1 Software Evolution and Maintenance 1  
1.2 Foundations of the Approach 3  
1.3 Structure of the Text 4  

2 System Reconfiguration 5  
2.1 Classification of Reconfiguration 5  
2.2 Types of Reconfiguration: Parameter and Compositional Adaptation 6  
2.3 Objectives of Reconfiguration: Minimizing System Disruption and Preserving Consistency 7  
2.4 Reconfiguration Strategies 11  

3 Autonomic Computing 13  
3.1 Autonomous Management: the Self-Properties 13  
3.2 AC System Architecture 14  

4 Component Orientation and Enterprise Java Beans 16  
4.1 The Concept of Component Orientation 16  
4.2 The Enterprise Java Beans Component Standard 18  
4.2.1 Enterprise Java Beans and EJB Modules 18  
4.2.2 EJB Module Development and Deployment Life Cycle and its Roles 20  
4.2.3 Container Services 22  

5 mKernel: A Manageable Kernel for EJB-based Systems 25  

6 A Model for Autonomous Reconfiguration of EJB-based Enterprise Applications 29  
6.1 Parameter and Compositional Adaptation of EJB Applications 31  
6.2 A Conceptual Overview of the Reconfiguration Model 32  
6.3 The Reconfiguration Model in Detail 33  
6.3.1 Specification of the Provided Set of Steps 34
List of Figures

1  Simple Staged Model for the Software Life Cycle ([RB00], p.67) ........ 2
2  AC system architecture [BNVW08] ........................................ 14
3  Architecture of a component-based system consisting of three components . 17
4  Conceptual view on an mKernel managed system from the administrator perspective ...................................................... 25
5  Reconfiguration Model ........................................................... 33
6  Ejb Module Life Cycle ............................................................ 34
7  Quiescence Region Life Cycle .................................................. 37
8  Schematic view on a step and on an associated step executor .............. 41
9  Schematic view on a strategy consisting of four steps ........................ 42
10 Simplified Strategy Creator API ............................................. 51
11 Simplified Reconfiguration Planner API .................................... 53
12 Simplified API Representation of an Executor Module and its executors 59
13 Simplified Life Cycle of a Stateful Session Bean Instance ................. 67
14 Enhanced Life Cycle of a Stateful Session Bean Instance ................ 69
15 Specification of the Flash Strategy ........................................... 94
## List of Tables

1. Classification for software composition using the time of composition or recomposition as a classification metric (cf. [MSKC04], p.61) ........................................... 6
2. Reconfiguration procedures and their steps ................................................. 81
## Listings

1. An Example for a Step Executor ........................................ 55
2. An Example for a Plan Executor ........................................ 57
3. IPlanStepExecutor interface .............................................. 58
4. XML-based Executor Descriptor for an Example Executor Module ........................................ 62
5. Adjusted Example Step Executor ........................................ 63
6. Example Executor Module Deployment Descriptor ........................................ 66
**List of Abbreviations**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Autonomic Computing</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CBSE</td>
<td>Component-based Software Engineering</td>
</tr>
<tr>
<td>CO</td>
<td>Component Orientation</td>
</tr>
<tr>
<td>EA</td>
<td>Enterprise Application</td>
</tr>
<tr>
<td>ED</td>
<td>Executor Descriptor</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise Java Beans</td>
</tr>
<tr>
<td>DD</td>
<td>Deployment Descriptor</td>
</tr>
<tr>
<td>DI</td>
<td>Dependency Injection</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IS</td>
<td>Information System</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>JAXB</td>
<td>Java Architecture for XML Binding</td>
</tr>
<tr>
<td>JAR</td>
<td>Java Archive</td>
</tr>
<tr>
<td>JEE</td>
<td>Java Enterprise Edition</td>
</tr>
<tr>
<td>JMS</td>
<td>Java Message Service</td>
</tr>
<tr>
<td>JNDI</td>
<td>Java Naming and Directory Interface</td>
</tr>
<tr>
<td>JSR</td>
<td>Java Specification Request</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definition</td>
</tr>
</tbody>
</table>
1 Introduction

Enterprise Applications (EA) are complex software systems for supporting the business of a company. According to Lehman’s laws [LRW+97] software implementing real world applications like EAs must continually evolve, else they become progressively less satisfactory and their use and value would decline. Therefore, huge amounts of the IS budgets of companies are spent on software maintenance and evolution [RBBC04, Rau00, Erl].

1.1 Software Evolution and Maintenance

The term software evolution lacks a standard definition and it is often used as a substitute for software maintenance [BR00]. The standard for software maintenance of the Institute of Electrical and Electronics Engineers (IEEE) defines software maintenance as (cf. [Mam94, BR00]):

*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.*

This definition states that the software has been developed and deployed and it is operational. For the purpose of this work, both terms, software evolution and software maintenance, are considered as synonyms. To identify the role of software maintenance in the software life cycle, the work of Rajlich and Bennett [RB00] is helpful. They propose a model for a complete software life cycle that consists of five coarse-grained and distinct stages (see figure 1 on page 2).

1. **Initial development:** This stage describes the development of the system such that a first functioning version is available. This stage is the subject of software engineering research that has proposed several well-known process models, like, e.g., the waterfall model [Roy87], the spiral model [Boe88] or the Rational Unified Process [JBR99], that might be used. Nevertheless, this stage is important for the maintenance of a developed system, because this stage defines the system architecture. The kind of an architecture and its properties might enable or restrict future changes to the system.

2. **Evolution:** For Rajlich and Bennett, software evolution and maintenance are not synonyms, but evolution is a stage that is part of software maintenance. This evolution stage comprises arbitrary changes, even major ones, to the system because of changing requirements, like, e.g., extending or modifying its functionality and capabilities.

3. **Servicing:** In contrast to evolution, in this stage only minor changes are performed, like, e.g., the integration of patches to fix bugs or minimal functional changes. Major changes are difficult and expensive with the continuing aging of a system because its architecture has been degraded with every change iteration. Thus, maintainability or evolvability have almost got lost.

4. **Phaseout:** During this stage, the company does not undertake any more servicing, i.e., no changes are done to the system. Nevertheless, the system stays operational as long as it fulfills sufficiently the requirements of its customers and the company could
generate revenues from it. Performing no changes, the system becomes increasingly outdated with each changing requirement, such that it will enter the next stage at some point in time.

5. **Closedown**: The system will be closed down and withdrawn from the market because it does not fulfill the business needs appropriately. Usually, a newly developed system that meets the current requirements replaces the outdated one.

![Simple Staged Model for the Software Life Cycle](image)

**Figure 1**: Simple Staged Model for the Software Life Cycle ([RB00], p.67)

Using the terminology of Rajlich and Bennett, software maintenance can be classified into the software life cycle though the traditional boundary between software development and software maintenance might be blurred. This blurring happens, e.g., when the release occurs in several steps including alpha and beta releases. Software maintenance covers the stages *Evolution* and *Servicing* during which iterative changes occur. Transitions from one stage to another especially depend on the revenues being gained from the system and on the amount the company is willing to invest in order to maintain the system. Furthermore, these transitions are influenced by the ability of the system architecture to facilitate changes. This ability decreases with each change done to the system such that the system and its architecture become hard to understand and therefore, hard to maintain [RB00].

The need for software evolution originates, e.g., from failures, inefficiencies or changes of the business or of the system environment that lead to new or changing requirements for EAs [Som04]. Thus, software evolution can be categorized as corrective (removing software faults), adaptive (adjusting the system to the changing environment), or perfective (enhancing or improving the functional and non-functional system characteristics) (cf. [Swa76, OMT98]). Though there are attempts to provide more fine-grained taxonomies, like, e.g. the classification of Chapin et. al [CHK+01], the categorization of corrective, adaptive and perfective changes is sufficient because it delivers an insight that several origins for evolution are conceivable. Furthermore, it has been adopted by the IEEE software maintenance standard [Mam94].
Due to the critical role of an EA within a company this inherent evolution should not affect the availability of an EA. A temporal shutdown may affect business operations. Thus, the company might miss business opportunities and lose reputation and trust. One approach to address this problem is the post-deployment runtime evolution, i.e., applying the relevant changes to the system while it is running [BMZ+05]. The reconfiguration should minimize the system disruption and it should be transparent to the clients of an EA, i.e., clients may notice delays in the response time of an EA, but no failures. Therefore, the reconfiguration should be carried out seamlessly and it should preserve application consistency. The approach of seamless reconfiguration can be seen as one critical challenge in software evolution [MWD+05]. This challenge is enhanced by the complexity of EAs that even increases with the evolution of EAs [LRW+97].

1.2 Foundations of the Approach

One possibility to meet the challenge of seamless runtime reconfiguration and to address the complexity of EAs is to combine the concept Component Orientation (CO) [Szy02] and the vision of Autonomic Computing (AC) [Hor01, GC03, KC03]. Component-based software engineering (CBSE) produces modular software systems. This modularity enables changes at the architectural level of component-based applications during the maintenance phase. The objective of AC is, among others, the automation of system maintenance tasks to disburden human administrators. Both approaches may help to facilitate runtime reconfiguration and they are integrated in the mKernel system. With mKernel [BW07, BNVW08] a generic AC infrastructure is available that enables the comprehensive management of component-oriented EAs that are realized with the Enterprise Java Beans (EJB) 3.0 technology [DK06a, DK06b].

The subject of this work is to reconfigure EJB-based enterprise applications seamlessly at runtime. Therefore, mKernel provides runtime support by offering several facilities that can be combined to enable manifold reconfigurations. Thus, based on mKernel, a comprehensive set of steps is provided. Each of them is customizable and specifies fine-grained reconfiguration tasks. These steps can be combined flexibly to generic and autonomous reconfiguration procedures for EJB-based EAs. Each of these procedures realizes a certain reconfiguration strategy, i.e., a certain way to perform a reconfiguration. Theses strategies are reusable and serve as templates for easing the planning and execution of a concrete reconfiguration. Instead of prescribing, how a reconfiguration should be applied, several strategies provided by this work can be used. Furthermore, it is possible to create and integrate new strategies, such that this approach is not limited to the provided ones. This flexibility allows to meet the demands of concrete changes that should be done to concrete applications. Finally, this work aims at simplifying the work of an administrator regarding a reconfiguration, e.g., through the reusability of strategies for several reconfiguration situations. The approach of this work has been implemented and the implementation assists an administrator in developing and applying a reconfiguration of EJB-based applications.

The basic concepts and ideas of the approach have been published at the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE 2008) at Redwood City, San Francisco Bay, USA, July 1 - July 3, 2008 [VBW08].
1.3 Structure of the Text

The text of this thesis is structured as follows:

Section 2 presents background information on System Reconfiguration. This comprehends a classification of system reconfiguration and a description of two general types of reconfiguration, namely parameter and compositional adaptation. For performing a reconfiguration, two objectives should be considered. First, disruption to the system should be minimized, and second, consistency of the system should be preserved throughout a reconfiguration. Finally, reconfiguration strategies, being obtained from literature, are discussed.

The vision of Autonomic Computing is described in section 3. The description includes the so called self-properties that characterizes autonomic systems. Additionally, the generic architecture of autonomic systems is presented that enables autonomic behavior. An important part of this architecture is the so called control loop.

Basic information about Component Orientation is provided by section 4. Especially, the modular architecture of component-based applications is emphasized. Afterwards, the concrete component standard Enterprise Java Beans and its most relevant aspects are presented. These aspects are, among others, the main building blocks of the EJB component model and the different roles that are involved in the development and deployment of EJB-based applications.

Section 5 gives an introduction to mKernel, which is a generic Autonomic Computing infrastructure for EJB-based systems. Since this approach makes primarily use of mKernel, only the Application Programming Interface (API) provided by mKernel is described and its internals are left here.

The Reconfiguration Model as the design of this approach is presented in section 6. After discussing parameter and compositional adaptation in the context of EJB, an overview on the reconfiguration model is given. Then, the building blocks of the model are described in detail, and finally, the roles that are involved in the development and application of a reconfiguration are defined.

Section 7 covers the Implementation of the Reconfiguration Model. The model is presented in the previous section. The API, being provided by the implementation, for each role in the development and application of a reconfiguration is presented. Afterwards, internals of the implementation that are hidden by the API are discussed. These internals addresses, among others, aspects that enable a state transfer.

Using the reconfiguration model and its implementation, which are both discussed in the previous sections, the Design and Realization of Provided Strategies are presented in section 8. Four reconfiguration strategies have been implemented and they are provided by this approach. They might be used to replace a running EJB module with an alternative implementation for this module. The implementation of these strategies is a proof of concept for the reconfiguration model and for the model implementation.

Section 9 discusses Related Work in the area of reconfiguration in the EJB domain. It compares selected approaches from literature to this work. Finally, section 10 gives a Conclusion of this approach and suggests ideas for Future Work.
2 System Reconfiguration

All changes done to a system in order to maintain it can be summarized under the term system reconfiguration. After classifying system reconfiguration, two general different types of reconfiguration, namely parameter and compositional adaptation are presented. Subsequently, desired objectives of a reconfiguration are described and finally the question of how to apply a reconfiguration is discussed.

2.1 Classification of Reconfiguration

Reconfiguration can be classified, among others, according to the time of change, anticipation, degree of automation, and activeness [BMZ+05].

- **Time of change** describes when the changes are integrated into the software system. As already described in section 1.1, the approach of seamless reconfiguration modifies the software during its execution. In contrast to this runtime evolution, changes can happen at compile time or load time. The first one modifies the source code of the system and recompiles it to make the changes available. For the latter one, changes occur while the software is loaded into the executable system. These three types of software changes do not exclude each other. E.g., compile-time evolution may enable runtime evolution if the modified and recompiled source code is integrated into an already running system. Runtime evolution imposes that the affected system, i.e., the system under reconfiguration, is available to clients during the change phase.

- **Anticipation** characterizes the time when software changes and their requirements are foreseen. Anticipated changes can be foreseen during the development of the affected system. Therefore, decisions regarding the system design and architecture can consider the future changes to enable or to facilitate them. In contrast, unanticipated changes cannot be foreseen during development. Therefore, it is difficult to prepare the system during its development for such changes that may happen in the future.

- The **degree of automation** distinguishes between automated, partially automated, and manual change support. This comprehends the planning, preparation, and the execution of a reconfiguration.

- The **activeness** property splits systems into two types: for a reactive systems changes are driven externally, e.g., by an administrator. In contrast, a proactive system autonomously drives changes to itself. Proactive systems can be characterized as self-adaptive or self-reconfiguring.

In practice, EAs are often shut down temporarily for their maintenance. This shutdown may have negative consequences, because business operations are affected. The company might miss business opportunities and lose reputation and trust. To avoid these negative outcomes and to meet the demand for EAs to be permanently available, reconfigurations should occur at runtime (see section 1.1). Thus, a seamless reconfiguration happens at runtime and does not disrupt significantly system operations. Since it is not possible to
predict all future change requests originating, e.g., from the evolving business or technological environment, unanticipated changes should be supported. Another objective is to automate the planning, preparation, and application of reconfiguration as much as possible because of the complexity of EAs. Thus, human administrators or reconfigurators could be disburdened. Regarding the property of activeness, a reliable reactive system is sufficient because later on, it can be extended to a proactive system. Thus, at first, the functionality to perform changes to a system, i.e., to execute a reconfiguration, is important. If this functionality is available and reliable, it could be attempted to move the task of detecting reconfiguration needs and the tasks of planning and preparing a concrete reconfiguration from a human administrator to the system itself.

### 2.2 Types of Reconfiguration: Parameter and Compositional Adaptation

The architecture of a software system is the high-level organization of its constituent computational elements and the interactions between those elements ([GP95], p.269). In this context, according to McKinley et. al [MSKC04], there are two general types of software reconfiguration: parameter adaptation and compositional adaptation. The first one modifies variables of one or more elements that determine behavior of the elements. The second one addresses structural reconfiguration at the architectural level through addition and removal of elements, including the manipulation of connections among them (cf. e.g. [KM90, OMT98, RAC+02]). Therefore, compositional adaptation changes the topology of the system. The weakness of parameter adaptation is that it allows only anticipated changes, because the elements have to provide the variables and react appropriately to their modifications. In contrast, compositional adaptation is intended for the dynamic and unanticipated reconfiguration of a system. E.g., a faulty element is updated with a correct one, or a new element with algorithms for addressing concerns that were unforeseen during development is integrated into the system. Consequently, compositional adaptation provides more possibilities than parameter adaptation.

<table>
<thead>
<tr>
<th>Application type</th>
<th>Mutable or Tunable</th>
<th>Configurable</th>
<th>Customizable</th>
<th>Hardwired</th>
<th>✓</th>
<th>dynamic comp.</th>
<th>✓</th>
<th>static comp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development time</td>
<td>Compile or link time</td>
<td>Load time</td>
<td>Runtime</td>
<td>Increasing dynamism</td>
<td>→</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Classification for software composition using the time of composition or recomposition as a classification metric (cf. [MSKC04], p.61)

Comparable to the classification of Buckley et. al [BMZ+05] presented in section 2.1, McKinley et al. [MSKC04] classify the time when composition occurs (see table 1). Static composition takes place at development, compile, or load time, and dynamic composition
at runtime. In this order, dynamism increases. If application composition occurs at development time, then any adaptive behavior is hardwired into the software. Changes of the behavior require its recoding. Customizable applications are composed at compile or link time. Therefore, adaptations require the recompilation or relinking of the software. Composition at load time may decide to choose one of several elements when the corresponding element is loaded for the first time or when the application is deployed. Such applications are configurable and require no recompilations to make the changes available. The most flexible approach is the dynamic composition that changes the structure of the application during execution without stopping and restarting the program. Such applications are mutable if the adaptations modify the business logic. The application is tunable if modifications of code for the business logic is not permitted and if only non-functional aspects are altered.

Using this classification, the approach of reconfiguring applications seamlessly can be associated with the dynamic composition. As the need for changes can be manifold, which is shown in section 1.1, alterations of functional and non-functional software characteristics should be supported. Thus, the software of the application that is subject to reconfiguration should be of type tunable and mutable.

2.3 Objectives of Reconfiguration: Minimizing System Disruption and Preserving Consistency

For carrying out a reconfiguration, especially two objectives are to be considered and desirable according to Kramer and Magee [KM90]. First, the reconfiguration should minimize the disruption to the system, i.e., the affected part of the system may notice delays but no failures, while the rest of the system should be able to continue its execution normally. Thus, reconfiguration should be carried out seamlessly. Second, a consistent state of the system must be preserved during and after reconfiguration. Application consistency is especially critical for structural changes of an application, like, e.g., an update or replacement of an element at runtime.

A formal specification of an element update at runtime is given by Gupta, Jalote and Barua [GJB96], who use the term on-line instead of runtime. They consider an element as a process $P$ that executes a program $\Pi$. Thus, a process consists of the program code being executed and its current state $s$. A process starts with an initial state and the execution is a sequence of state transitions. A state characterizes completely a point in the lifetime of a process, if the program that is executed by the process is given. Based on this terminology, Gupta, Jalote and Barua define an on-line update, i.e., a replacement of a program, as follows:

**Definition (On-line change [GJB96]):** An on-line change from program $\Pi$ to $\Pi'$ at time $t$ using the state mapping $S$, in a process $P$ (executing $\Pi$) is equivalent to the following sequence of steps:

1. $P$ is stopped at time $t$ in state $s$.
2. The code of $P$ (which, until now, was the program $\Pi$) is replaced by the program $\Pi'$, its state is mapped by $S$ and $P$ is then continued (from state $S(s)$ and with code of $\Pi'$).
Since any arbitrary change will not produce meaningful and acceptable results, Gupta, Jalote and Barua argue, that the best that can be expected is that after a transition period after the change, the process behaves like a process executing the replacing code [GJB96]. Therefore, the notion of a reachable state is important. A concrete state \( s \) is reachable within a program \( \Pi \) if and only if a process executing \( \Pi \) beginning in its initial state can reach \( s \) at some time for some inputs. A formal definition of the validity of an on-line change is as follows:

**Definition (Validity of an on-line change [GJB96]):** An on-line change in the process \( P \) from \( \Pi \) to \( \Pi' \) at time \( t \) (in state \( s \)) and using the state mapping \( S \) is valid if after the change, \( P \) is guaranteed to reach a reachable state of \( \Pi' \) in a finite amount of time.

Using this definition, given arbitrary programs \( \Pi \) and \( \Pi' \), a state mapping \( S \), and a reachable state \( s \) of \( \Pi \), Gupta, Jalote and Barua proofed that it is undecidable to determine whether an on-line change from \( \Pi \) to \( \Pi' \) using state mapping \( S \) in state \( s \) is valid or not. The state \( s \) characterizes the time of change \( t \) because \( s \) and \( t \) depend on each other. Undecidability of the problem has an important consequence: given \( \Pi, \Pi', S, \) and \( s \), there does not exist a general purpose algorithm that determines whether or not an on-line change is valid. Therefore, computable necessary and sufficient conditions for validity could not be obtained and validity checks may only rely on sufficient conditions. Though the proof is given for a simple and restricted program model (imperative language with typed variables as a sequence of statements without procedures and functions), this proof implies undecidability in more general program models such as for languages with procedures and functions, object-oriented models, and distributed program models [GJB96].

In general, the replaced code \( \Pi \), the replacing code \( \Pi' \), and the state mapping function \( S \) are given. Thus, sufficient conditions may only refer to the state \( s \) at time \( t \) to ensure validity [GJB96]. Therefore, an on-line change is performed when the application has reached or is transferred into a safe state during which consistency could be preserved, i.e., no inconsistencies would occur (cf. [AWPvS01, MGK96, Van07]). Consequently, the process of a runtime change can be divided into four parts: initialize the placement of the affected part of the system into a safe state, detecting that the safe state has been reached, performing the reconfiguration, and finally, releasing the affected part of the system from the safe state in order to resume operation.

A state is safe if the affected elements are quiescent [KM90], i.e., none of them is currently engaged in servicing a request and none of them will initiate a request. Furthermore, no requests initiated by non-affected elements are forwarded to affected ones. To reach a quiescent state, requests that are currently serviced must be finished. New requests must be blocked except those which are needed to finish servicing ones. Otherwise, some elements are not able to reach a quiescent state and they end up in a deadlock (cf. [KM90, Che02a]). Quiescence of the affected part of the system gives new elements the opportunity to be initialized in a state which is consistent with the rest of the system, and elements to be removed the opportunity to leave the system in a consistent state. The assumption is that a systems moves from one consistent state to the next. While requests are in progress, the states of elements might be inconsistent, but before or after servicing requests the states are consistent. Kramer and Magee have shown that quiescence is a sufficient criterion for ensuring consistency throughout a reconfiguration [KM90].
There exist different algorithms to place a system or parts of it into a quiescent state. Regarding the aspect for which elements communication should be limited, these algorithms can be divided into two categories that distinguish between static and dynamic algorithms [HW04a, HW04b]:

- **Static algorithms**, like, e.g., the one presented by Kramer and Magee [KM90], determine those elements whose communication should be limited based on their potential to interfere with affected elements, regardless of whether they actually attempt to communicate with them or not. An affected element is one that should be transferred into a quiescent state. Kramer and Magee [KM90] define, that an element in the *active* state can initiate, accept, and service requests. In contrast, an element in the *passive* state must continue to accept and service requests. It may initiate consequent requests that are required to finish the currently servicing requests, but it is not currently engaged in a non-consequent request that it initiated, and it will not initiate any new non-consequent requests. To place an element into a quiescent state, all elements being part of the so called *passive set* of this element must be directed into the passive state. The passive set of an element is defined as the affected element itself, all elements that can directly initiate requests to the affected element, and all elements that can initiate requests that result in consequent requests to the affected element. Therefore, several elements must be placed into the passive state, even if this might not be necessary, because requests may potentially involve the affected element but actually do not. Thus, significant system disruption might be the consequence.

- **Dynamic algorithms**, as, e.g., presented by Goudarzi and Kramer [MGK96], Chen and Simons [CS02, Che02b], or Ameida et al. [AWvSN01, AWPvS01], determine the set of elements whose communications should be restricted by means of the *actual* requests taking place in the system at the time a change is performed. E.g., a request to an affected element is blocked when the request is actually initiated, i.e., when an element tries to communicate with an affected one. Nevertheless, at runtime, it must be distinguished whether the request is a completely new one that must be blocked or a consequent one that must be serviced. This decision requires the analysis of the chain of requests [CS02]. If the chain of a request contains an element that should be quiescent, then this request must be served. E.g., Almeida et al. [AWPvS01] propagate such chain information along the chain through implicit parameters. In contrast, Kramer and Goudarzi [MGK96] define a so called *blocked set* that initially contains only the affected elements. Requests originating from members of this set must be serviced, all others must be blocked. To manage the problem of consequent requests initiated by an element, which received a request from a member of the set, but which is not itself a member, this element becomes a member of the set, too. Thus, consequent requests are serviced, because all elements in the request chain have become or have already been members of the set.

This distinction between static and dynamic algorithms is also done by Wermelinger [Wer97] who calls them the *passive* and the *blocking* approach, respectively. It might be suggested that the blocking approach causes less system disruption than the passive approach. Goudarzi and Kramer [MGK96] argue, that their blocking approach always performs at least as well as the passive method presented by Kramer and Magee [KM90] with respect to minimizing the system disruption. Nevertheless, Wermelinger found a scenario for which this does not hold.
Additionally to the distinction between consequent and non-consequent requests, it may be necessary to consider constrained requests [WS96]. Such a request is constrained in its execution by the state of the element that services the request. An example is a bounded buffer provided by an element. Requests to read and remove data from the buffer are pending if the buffer is empty. Likewise, requests to insert data into the buffer are pending if the buffer is full. Consider a scenario where the buffer is empty and one consumer request is pending. If the element providing the buffer should be placed into a quiescent state, then the already servicing requests, like the pending one, should finish and non-consequent requests are blocked. But, the pending consumer request cannot continue execution until a producer request is serviced. This may not happen since non-consequent requests are blocked and there may not exist any consequent producer request. Warren and Sommerville [WS96] propose that a configuration manager knows about all pending requests and may query the state of the buffer (full or empty). For the above example, the manager concludes that a producer request is necessary and therefore, it starts the blocking of non-consequent requests after enough producer requests have been accepted to free pending consumer requests. Other approaches do not mention such issues, but they avoid them. E.g., Goudarzi and Kramer [MGK96] require that requests may not interleave, i.e., an element is non-reentrant. Therefore, the scenario, that a request is pending in an element and another request is accepted by the same element, may not happen. Nevertheless, the functionality of the buffer element could be realized if two non-reentrant elements, one for producer and one for consumer requests, share one buffer, e.g., a database. If a consumer request in the consumer element is pending, then the other element may accept a producer request without the need for having more than one request being active in one element. But this contradicts another requirement of Goudarzi and Kramer that the state of an element is self-contained because both elements share the same buffer. Nevertheless, this remains an issue if the reconfiguration system does not impose any restrictions on the application to be reconfigured.

To ensure consistency in case of an element replacement, this may include the need for transferring the internal state of a replaced element to a replacing one [GJB96, OMT98, RRL07]. The advantage of requiring quiescence before any replacement happens is that no information about the execution must be transferred. Replacing an element while it services a request would require to transfer, e.g., the runtime stack or the exact location in the code when the replacement has happened [Hof93]. Therefore, the concept of quiescence limits the state, e.g., to global or instance variables in object oriented languages. Nevertheless, state transfer can be more complex than simply copying values of variables from one element to another. Replaced and replacing element may have states that differ in their syntax (e.g., different variable names or data structures) or semantics (e.g., two variables of the replaced element are mapped to one variable of the replacing element). Such changes in the state can be hardly managed automatically, and therefore, all approaches that consider a state transfer require from the element developer to provide functionality to convert the state of an element or even to get and set the state of an element (cf. e.g. [PMSD07, CHS01, Che02b, MH02, HG98]). Though there are approaches, like, e.g., the one of Vandewoude and Berbers [VB05a, VB05b], that try to partially automate the complete state transfer process, it is conceivable that this process cannot be fully automated and require assistance by the element developer or reconfigurator, i.e., the person who reconfigures the application.
2.4 Reconfiguration Strategies

How to apply changes are questions of reconfiguration strategies. One basic problem when replacing an element is what to do with existing element instances of the replaced element. At least three approaches are described by Hjalmtýsson and Gray [HG98] for C++ classes and their instances. These approaches differ in their handling of existing instances of a class that should be replaced and they could be applied to a general model of elements and element instances:

1. One approach is to raise a barrier that prevents the creation of new instances of the element that should be replaced. After all already existing instances have expired, the replaced element is no longer in use and it could be removed. Then the replacing element takes over, the barrier is removed, and new instances being created are of the type of the replacing element. According to Hjalmtýsson and Gray, this approach is conceptually equivalent to halting, modifying and restarting the system. If the expiration phase takes a long time, then system disruption is significant and hardly acceptable. Therefore, this approach is not adopted by the authors.

2. In contrast to letting all instances of the replaced element expire, one approach is to recreate all those instances using the replacing element, i.e., all those instances are converted to instances of the replacing element. This approach requires the transfer of the internal state of a replaced element instance to the corresponding replacing element instance. The conversion of instances happens globally in a change phase such that after the change phase when the operational execution resumes no instances of both the replaced and replacing element exist.

3. The last approach only makes sure, that, beginning with a certain point in time, instances are created only from the replacing element, i.e., no instances of the replaced element are created any more. Already existing instances of the replaced element continue their work until they expire. During this expiration phase, instances of both, replaced and replacing, elements are active, and after this phase only instances of the replacing element are in use. However, Hjalmtýsson and Gray consider an additional option that developers might use and that converts all or certain instances of the replaced element. The authors adopted this approach, but per default no instances are converted unless the developer uses the option and implements state-capture and restoration routines for both, replaced and replacing elements.

The idea of the second and third approach can also be found in two of the three reconfiguration strategies presented by Rosa, Rodrigues and Lopes [RRL07]. The three strategies are called Flash, Non-Interrupt, and Interrupt.

1. **Flash**: The Flash strategy reconfigures one element without concerning about other elements. Reconfiguration takes place immediately without handling existing interactions and the states of the affected elements. No state transfer is performed and existing connections to old elements are not updated. Therefore, these connections become invalid and are likely to cause errors. Finally, the system may become inconsistent. Consequently, Flash does not always perform a seamless and consistent
reconfiguration. Nevertheless, it can be used, among others, for parameter adaptation or for reconfiguring elements not being critical for the consistency of the application. In contrast, the other strategies preserve consistency of the system and perform a seamless reconfiguration.

2. **Non-Interrupt**: The *Non-Interrupt* strategy supports the exchange of elements without the need for quiescence, hence reducing system disruption significantly. Both elements, the old one that is going to be replaced and the replacing one, are active. An intercepting facility forwards requests of already existing sessions to the old one and requests of new sessions to the replacing one. After all sessions on the old element have finished, it can be removed and only the new element is used. This strategy does not require a state transfer, but it requires that the two elements can be used concurrently.

3. **Interrupt**: The *Interrupt* strategy transfers the affected part of the system into a quiescent state before reconfiguration takes place. The states of the affected elements and existing connections between elements are handled, such that, e.g., an element replacement can be performed without causing any failures. After reconfiguration, the affected part of the system is released at once from the quiescent state, such that it can be assured that all elements and connections are reconfigured appropriately, before resuming their execution. Comparably with the other strategies, an advantage of requiring quiescence is that, e.g., an underlying database is not used during quiescence, which enables its consistent modification or transfer.

As denoted above, regarding the handling of running elements, the strategies *Interrupt* and *Non-Interrupt* can be compared respectively to the second and third approach of Hjálmtýsson and Gray.

Consideration of several strategies is important to find the best way to reconfigure a system and to compromise between different reconfiguration operations and the costs involved. E.g. forcing quiescence may be expensive, but therefore it enables a database reconfiguration. Furthermore, different strategies can address different requirements of concrete applications that should be reconfigured.
3 Autonomic Computing

The vision of Autonomic Computing (AC) [Hor01, GC03, KC03] is one response to the increasing complexity of information technology (IT) systems. Its basic idea is to assign low level, administrative tasks to the managed system itself to disburden human administrators. The system manages itself according to the goals specified by the administrator. Thus, the IT staff can focus on strategic tasks, like, e.g., the further development of the systems. The objective is the autonomous management of systems at runtime while the system administrator is only concerned with high level functions, like, e.g., specifying the goals of the system.

3.1 Autonomous Management: the Self-Properties

This autonomous management covers the four aspects, known as the self-properties: self-healing, self-protection, self-optimization, and self-configuration. These four properties are summarized under the term self-management [LML05] and they are broadly adopted in literature (cf. [Hor01, GC03, KC03, GHS+04]).

- **Self-healing**: Self-healing systems detect and react to disruptions because of failures or extraordinary events. They must be able to recover from a malfunctioning component by detecting and isolating the affected component, and by integrating the fixed or a replacement component into the system. This replacement of a component constitutes a system reconfiguration that should not disrupt system operations because the objective of self-healing is to keep systems reliable and available. Additionally, the system must prevent the failure from having negative impacts on itself and on other systems [GC03].

- **Self-protection**: Self-protecting systems anticipate, detect, and protect themselves from malicious attacks [GC03, GHS+04]. The objective is to maintain overall system security and integrity [Hor01]. Attacks can be, e.g., viruses or intrusions by hackers that access resources without authorization. Additionally, backup and recovery capabilities must be provided to ensure system availability if the original system facility is affected by a successful attack [GC03].

- **Self-optimization**: Self-optimizing systems monitor and tune their resources automatically to improve their operation. Therefore, systems should efficiently maximize resource utilization to meet end-user or business needs without human intervention [GC03, GHS+04, KC03]. The objective is that system operations are always efficient and that the quality of service goals are achieved. This includes, e.g., adaptations to changing workloads.

- **Self-configuration**: System configuration must occur automatically, as must system reconfiguration, i.e., dynamic adaptations of the configuration under varying and unpredictable conditions [Hor01]. Such adaptations stand for the possibility that new features or software can be dynamically manipulated, added to, or removed from the system without disrupting system operations [GC03, GHS+04]. The self-reconfiguring facility of a system supports the other three self-properties by adjusting the system as required in a concrete situation.
All four properties work together to devote themselves to the intent of autonomic computing. System administrators should be freed from details of system operation and maintenance and users should be provided with services that run at peak performance 24/7 [KC03]. This permanent availability of systems implies that AC systems must support runtime software evolution that does not disrupt system operations substantially. The aspect of self-configuration addresses reconfiguration explicitly. Furthermore, other properties can be mapped to at least one of the different kinds of system evolution discussed in section 1.1, namely corrective, adaptive, and perfective. In the following, some examples are shown. Self-healing facilities perform corrective changes to remove failures. The malfunctioning part of the system can be replaced with an functioning one, or a patch will be applied. Self-protection covers actions to adapt the system to incidents within the environment. A system may decide to activate the CAPTCHA functionality [vABL04] if it assumes that a spam bot is active in its environment. Finally, perfective changes are addressed, among others, by self-optimization. To improve quality of service, like to reach or maintain a certain performance level, a system always tries to optimize its workings.

### 3.2 AC System Architecture

Generally, the architecture of an autonomous system consists of two layers that are depicted in figure 2. The bottom layer is called *Managed Layer*. This layer represents the managed system that consists of the *Application* providing the actual functionality to the user or to another system and *Sensors* and *Effectors*. Sensors and effectors are used for structural and behavioral inspection and manipulation of the application to be managed. The top layer, called *Management Layer*, is responsible for the administration of the bottom layer. Therefore, the top layer uses the sensor and effector interfaces provided by the bottom layer. This partition into two layers separates application logic from application management concerns (*Separation of Concerns*). Finally, a system executes within an *Environment* that influences the system, and the system influences its environment. Thus, the context surrounding the managed application has to be considered.

![Figure 2: AC system architecture [BNVW08]](image-url)
To perform the autonomous management, the *Management Layer* implements a *Control Loop* [DHP+05, KC03] consisting of four stages as shown in figure 2 on page 14. The first stage (*Monitoring*) inspects the structure and behavior of the managed application through sensors. Furthermore, the environment is monitored, too. To support all self-properties of autonomous management, different kinds of information are relevant and have to be obtained from the system and its environment. Therefore, the information that is gathered depends on the supported AC aspects. The architecture does not imply how information about the managed application is provided. There might be scenarios where a *push oriented* approach for information provision is appropriate, e.g., to inform the top layer about the occurrence of an event. On the other hand, the *Managed Layer* may require information from the managed application using a *pull oriented* approach, e.g., to inspect the structure of the application that is usually more stable. The information being collected by the *Monitoring* stage is the basis for the subsequent stages of the control loop. This information is processed by the second stage *Analysis*. Therefore, this stage analyzes the information in order to detect situations that require a reaction of the *Management Layer*. This detection might span from simple to complex analysis. A simple case is, e.g., the identification of a single exception event being directly provided by the *Monitoring* stage. On the other hand, analysis might require complex evaluation and aggregation tasks to correlate the information, e.g., to detect attack patterns. If the managed application needs to be reconfigured, the *Planning* stage is addressed. It generates a plan, how the application can be adjusted to meet the reconfiguration needs and to fulfill the overall goals of the system. The system adjustments might be simple, like, e.g., tuning configuration parameters, or more complex. E.g., comprehensive adjustments might change the architectural structure of the managed application. Finally, the last stage *Execution* is responsible for executing the plan that has been generated by the previous stage. Therefore, the effector interfaces are used to actually adjust the system as described by the plan. Each stage might use internal *Knowledge* during its execution. This knowledge base covers, e.g., information about symptoms that describe problematic situations. These symptoms can be used by the *Analysis* stage for comparison with the current state of the application to detect these problematic situations. Another example for the knowledge base are options for reconfiguration that can be used by the *Planning* stage to find those options which are suited best to meet the reconfiguration needs and system goals.

Though, in this description, the process of autonomous management passes each stage of the control loop in clockwise direction, this flow does not have to be unidirectional. E.g., in order to create an appropriate plan, it might be necessary that the *Planning* stage requests additional information from the *Monitoring* stage. Or, during plan execution, the current state of the application must be monitored and analyzed to proof whether or not the currently executed adjustments are successful. Moreover, it is conceivable that several interacting control loops are organized in a hierarchy [KC03] which leads to more complex interaction patterns.
4 Component Orientation and Enterprise Java Beans

In the late 1990s Component-Based Software Engineering (CBSE) emerged as an approach to software systems development that emphasizes the reusability of software entities. CBSE has become important because software systems are becoming larger and more complex. To cope with this complexity and to deliver better software more quickly, software components should be reused rather than re-implemented. Therefore, Sommerville argues that reuse-based software engineering is becoming the main development approach for business and commercial systems [Som04]. CBSE covers the development of components and the composition or integration of components into systems. After describing basic concepts of Component Orientation, one concrete component model, namely Enterprise Java Beans, is presented.

4.1 The Concept of Component Orientation

The concept of Component Orientation (CO) [Szy02, Szy03] is a paradigm for the development of software systems in a modular way through functional decomposition. Such systems are composed of modules, called Components. A component is defined as follows:

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 ([Szy02], p.41).

This definition states that a component encapsulates a certain functionality and provides it through Interfaces. A component may specify context dependencies it needs to function appropriately. Dependencies are, among others, especially required services from other components that can be used through their provided interfaces. An interface required by a component is called Receptacle. Consequently, the architecture of a component-based system can be seen as a collection of loosely-coupled modules which collaborate among each other through their interfaces. Figure 3 on page 17 shows an example for an architecture of a component-based application consisting of three components. The Component Order provides the interface OrderEntry and requires the interfaces Person and OrderableItem which are provided by the components Customer and Product, respectively. Furthermore, another component providing the Account interface would be necessary to fulfill the receptacle of the Customer component, such that the overall application can function appropriately. This modularity of component-oriented applications addresses the complexity during development and deployment of systems by modularity of requirements, architectures, designs, implementations and deployments. Instead of always having to address the whole application, it is possible to focus upon the relevant component. In contrast to the component definition of Szyperski, the following definition of Jed Harris considers the maintenance of components explicitly:

A component is a piece of software small enough to create and maintain, big enough to deploy and support, and with standard interfaces for interoperability (cf. [And03, Szy02]).

Therefore, the modularity can be extended to the maintenance phase by enabling the partial reconfiguration of component-oriented applications. Each component can be reconfigured individually without having to consider the whole application, i.e., all components.
Figure 3: Architecture of a component-based system consisting of three components

The fact that components can be composed by third parties emphasizes the reusability of components such that a component might be integrated into several systems. This reusability is enabled by the encapsulation of implementation of a component, constraining interaction through well-defined interfaces and a clear specification what a component requires and provides.

These well-defined interfaces are technically a set of named operations that can be invoked by client components. Moreover, an interface specifies the semantics of its operations and of its usage. This specification is a contract between the provider of the corresponding interface and the clients using this interface. A contract states what a provider must implement to meet the services promised by the interface, and what a client needs to do to use the interface. E.g., for each operation of an interface a contract can be specified by pre- and postconditions (c.f. [Szy02, Mey92]). Preconditions are requirements the client has to fulfill before calling the operation. Thus, the provider can rely on these preconditions whenever the operation is called. Postconditions express properties that must be ensured by the provider before the operation results are returned to the client. After a call returns, the invoking client can rely on these properties to be met. Nevertheless, using pre- and postconditions not all aspects of an interface can be specified (cf. [Szy02, Gen99]). Additionally to pre- and postconditions, invariants might be specified that constrain all operations of an interface. Each operation must guarantee that the invariants are satisfied when the invocation of this operation finishes if the invariants were satisfied just before the invocation has started. Given this short overview on pre- and postconditions and invariants, it should be clear that an interface is not merely a syntactical specification of operation signatures, but it includes aspects regarding the semantics of, among others, operations or parameters. Both aspects constitute the contract of an interface.

In a practical application, a component adheres to a particular component model that defines, among others, how a component is constructed, what type of dependencies it may have, how components can be composed, and how components can communicate and interact with each other. One representative of such a component model is the Enterprise Java Beans (EJB) standard.
4.2 The *Enterprise Java Beans* Component Standard

The component standard *Enterprise Java Beans* (EJB), version 3.0, [DK06a, DK06b] defines a sound component model. It is intended for the realization of enterprise applications on top of the Java programming language. In a distributed multitiered application, EJB components provide the server-side business logic [Sun07]. Therefore, the standard for the *Java Enterprise Edition* (JEE), version 5, [Sha], which specifies aspects relevant for enterprise applications based on Java technologies, requires the support of EJB. The EJB specification addresses the development and deployment of components and it defines requirements an *EJB container*, the runtime environment for components, has to fulfill. A container provides services, like, e.g., support for transactions, security, or persistence, which may be used by components. Imposing requirements on components and containers, EJB follows the Java philosophy *Write Once, Run Anywhere*, i.e., a component can be developed once, and then deployed in multiple containers without recompilation or source code modification of the component. Therefore, reusability is supported. Because of brevity, this section only gives an overview of the EJB standard and describes only the most relevant aspects.

4.2.1 *Enterprise Java Beans* and EJB Modules

The EJB component model is based on so called *Enterprise Beans*, or *Beans* for short. There are two types of beans considered in the standard, namely *Message-Driven Bean* and *Session Bean*.

A message-driven bean is accessed through asynchronous message passing. A client sends a message to the destination or endpoint for which a message-driven bean is the message listener. During deployment of a message-driven bean, the bean is associated with the destination or endpoint. As a result of the arrival of a message at the destination or endpoint, the container invokes a method on a corresponding message-driven bean instance in order to process the message. Since the client only interacts with the destination or endpoint but not directly with a message-driven bean instance, all instances are anonymous for the client, i.e., they have no client-visible identity.

Session beans provide interfaces, the so called *business interfaces*, to access its encapsulated functionality. An interface can be a *local* or *remote* one. Local clients, i.e., clients using a local interface of a session bean, must be colocated in the same *Java Virtual Machine* (JVM) with the session bean providing the local interface. The arguments and results of methods of local interfaces are passed by reference. In contrast, remote interfaces provide location transparency because remote clients and the session beans providing the remote interfaces need not to be part of the same JVM. The arguments and results of methods of remote interfaces are passed by value. Furthermore, session beans can be either *stateless* or *stateful*. An instance of a stateful session bean is exclusively used by a single client and retains its client-specific *Conversational State* across multiple invocations. Therefore, a client can rely on interacting with the same instance in case it uses the same reference for multiple invocations. The conversational state is defined as the field values of the session bean instance, its associated interceptors (Interceptors are described in section 4.2.3) and their instance field values, plus the transitive closure of the objects from all these fields reached by following object references. In contrast, an instance of a stateless session bean is not exclusively used by a client and therefore, has no conversational state. Moreover, each invocation from a client on the same reference may be executed
on different instances. Thus, the states of stateless session beans are client-independent and all instances of one stateless session bean are equivalent when they are not actually involved in servicing a method invoked by a client. Nevertheless, they may retain a state for performance benefits, e.g., to keep an open database connection for reuse. Regarding the conversational state, the specification of message-driven beans is equivalent to those of stateless session beans. Thus, message-driven beans are stateless, too. Beans are by definition non-reentrant, i.e., it is not possible that more than one method call is active on a bean instance at the same time. Furthermore, bean instances are not allowed to perform any kind of thread handling to avoid the need for synchronization and to simplify the programming model of EJB.

Beans can declare context dependencies and the EJB standard specifies how beans gain access to those dependencies. Dependencies can be, among others, receptacles or Entity Managers (see paragraph Persistence in the EJB Component Model in section 4.2.3). Receptacles are called EJB References and they can be connected to interfaces provided by session beans. Dependencies can be declared within the source code of a bean using metadata annotations or within the Deployment Descriptor (DD) (see below for a description of a DD). To gain access to the declared dependencies, two facilities are considered. The first one enables the lookup of a dependency by its name at runtime through the Java Naming and Directory Interface (JNDI) [jnd]. Each container must provide a JNDI implementation. The second one is called Dependency Injection (DI). As the name of this facility suggests, the dependency is injected into a field or setter method of the bean declaring the dependency. Instead of the bean instance having to look up the dependency, the container is responsible for providing the dependency to the bean instance at runtime. Therefore, the approach of DI follows the idea of Inversion of Control [Fow04].

To customize the business logic of an enterprise bean, a bean may provide Simple Environment Entries. These entries can be interpreted as a kind of configuration parameter or property for a bean. They are called simple, because the types of their values can only be basic data types. The values of entries are set within the DD, and at runtime, they are looked up by beans or injected into a bean like a dependency.

As unit of deployment the EJB standard specifies the EJB module, a Java Archive (JAR) file. A module must contain at least one enterprise bean and related user-defined classes, and it may contain an optional Deployment Descriptor (DD). In general, during the development of a bean, among others, the provided interfaces of the session bean, the dependencies of the bean, including its receptacles, and the simple environment entries of the bean are specified. This configuration can be changed until the beans are deployed. Specification and changes of the configuration can be done through metadata annotations within the source code of beans or through a DD. The options a DD offers cover all aspects of the metadata annotation and provides additional facilities, like, e.g., setting the values of simple environment entries. If both, metadata annotations and DD settings, refer to the same configuration aspects, the DD settings overwrite the settings of the metadata annotations. Therefore, it is possible to change the configuration of a bean without the need of source code modifications and recompilation.

Before deployment, the configuration must be finished. This includes, e.g., that the values of simple environment entries are set or that all dependencies are specified in a kind such that they can be fulfilled at runtime by the container or the operational environment. The latter one means, e.g., that all receptacles of a bean are connected to corresponding interfaces provided by session beans, which are part of the same EJB module, which are
already running in the target container or which will be deployed themselves afterwards
before the requiring beans become operational. However, after the deployment of a module
the configurations of beans being part of the module can not be changed.

Finally, an EJB-based system usually consists of several EJB modules. Its architecture
can be interpreted as a directed graph consisting of beans as nodes and connections as the
arcs between the nodes. This is comparable to the system shown in figure 3 on page 17.
Furthermore, nodes representing beans of the same module are grouped since the module
is the unit for deployment.

4.2.2 EJB Module Development and Deployment Life Cycle and its Roles

The EJB specification [DK06a] defines seven distinct roles in the development and de-
ployment life cycle of EJB-based applications. Each EJB Role may be performed by a
different party, but in some scenarios, a single party may perform several roles. In addition
to the roles, contracts are specified that ensure that the product of one role is compatible
with the product of other roles. These roles are described in the following.

1. Enterprise Bean Provider: The enterprise bean provider, or bean provider for short, is the producer of enterprise beans. This role provides the implementation of a bean and specifies metadata of a bean. This metadata can be specified through annotations or through a DD, and it represents configuration information. These information include, e.g., the required and provided interfaces, and other dependencies of the bean. As being concerned with the implementation of the business logic, the bean provider is typically an application domain expert.

2. Application Assembler: The application assembler combines enterprise beans produced by bean providers into deployable application units. The application assembly step occurs before the deployment of enterprise beans. The assembler is a domain expert who composes applications. Therefore, the metadata of each bean and the contract specifications of provided and required interfaces are used in order to build a functioning composition. Since the assembler is concerned with interface contracts, he or she must be familiar with the functionality of each bean, but need not to have any knowledge of the implementation of each bean.

3. Deployer: The deployer takes the output of a bean provider or application assembler and deploys the EJB modules in a specific operational environment. The deployer must resolve all external dependencies of a bean, such that the operational environment fulfills these dependencies, and must follow the application assembly instructions defined by the application assembler. After that, the beans fit into the specific operational environment and they could be deployed in a specific EJB container. The deployer is an expert at a specific operational environment and responsible for the deployment of modules in this environment.

4. EJB Server Provider: The EJB server provider is a vendor of a server that integrates an EJB container. The current EJB specification assumes that the server provider and the EJB container provider are the same vendor.

5. EJB Container Provider: The container provider provides tools that are necessary for the deployment of enterprise beans and the runtime support for the deployed
beans. The container is part of the target operational environment and it provides services, like, e.g., support for transactions or persistence, to the deployed beans. The main task of a container provider is the development of a scalable, secure, transaction-enabled container that is integrated with an EJB server.

6. **Persistence Provider**: The persistence provider provides an implementation of a scalable, transaction-enabled runtime environment for the management of persistence. This implementation must support the *Java Persistence API* [DK06b]. This includes the tools that are necessary for the object/relational mapping of persistent entities to a relational database, and the runtime support for the management of persistent entities and their mappings to the database.

7. **System Administrator**: The system administrator is responsible for the configuration and management of the IT infrastructure of a company that includes the EJB server and container. The administrator has also the task to oversee the behavior of deployed enterprise beans applications at runtime, e.g., to recognize errors or poor performance.

The roles of the *Deployer* and of the *System Administrator* are supported by two *Java Specification Requests* (JSR). The deployment life cycle of applications is defined by the *Java Enterprise Edition 5 Deployment API Specification, Version 1.2 (JSR-88)* [Doc06]. This API is implemented by an EJB container and by tools that enable the deployment of application in the container. Such tools assist the *Deployer* in managing the life cycle of applications. The deployer may perform, among others, the following operations:

- An application may be *distributed* to a container, i.e., the configuration of the module is validated, container specific operations may be performed, and the module is moved to the designated target container.

- A module that has been distributed before might be *started* such that it becomes runnable and available to clients.

- On the other hand, a running module, i.e., a started one, might be *stopped* such that it becomes unavailable to clients.

- An *undeployment* removes a stopped module from the target container.

- As an optional operation, the *Deployment API* specification defines a redeployment, that replaces a currently deployed application with an updated version. Nevertheless, the runtime configuration information of the updated application must remain identical to the application it is updating. Furthermore, this redeployment must happen transparently to the clients of the affected application.

- Finally, several operations are provided to retrieve the modules, which have been deployed in the target container, and their life cycle states.

Besides considering a *very basic type of application redeployment* ([Doc06], p.3), which is not required to be provided by a container, the specification does not address the issue of replacing or updating modules. Nevertheless, such a facility is not prohibited by the specification.
4 Component Orientation and Enterprise Java Beans

While the Deployment API supports the tasks of the Deployer, the Java Enterprise Edition Management Specification, Version 1.1, (JSR-77) [Hra06] addresses the role of the System Administrator. This specification defines an information model for the monitoring and controlling of servers and application components running in the server. Furthermore, it specifies how this model can be accessed. The model consists of Managed Objects that are definitions of units of management information. A managed object represents an entity that is instrumented and managed, like, e.g., an enterprise bean, an EJB module, or a server. A managed object does not deliver much more information than a unique name to identify the entity, but it can provide additional information and functionality. This additional information comprehends state information and performance statistics, and the additional functionality covers an event model and the opportunity to trigger changes in the state of a managed object. But these additional features are not required to be supported by a managed object. Therefore, the Management Specification is hardly helpful in inspecting comprehensively the behavior of an enterprise bean application at runtime. Furthermore, beyond changing the life cycle of modules, opportunities to manipulate an application and its beans at runtime are not considered by this specification [Hra06] or by any other JEE or EJB specifications [Sha, Doc06, DK06a].

4.2.3 Container Services

In the following, some relevant aspects of the EJB component model, namely Interceptors, Transactions, the Timer Service, and support for Persistence, are presented briefly. All of them are services provided by the EJB container that may be used by enterprise beans. In addition, there exist further container services, like, e.g., security support, which are not presented here for brevity.

Interceptors An arbitrary number of Interceptors can be attached to each enterprise bean. If a method should be invoked on a bean, this call is directed to the first interceptor of the sequence of all interceptors attached to the corresponding bean. This interceptor, like all the other interceptors in the chain, has full control over the control flow of the call and over the parameters of the call. Therefore, an interceptor can hand over the control flow to the next interceptor or if it is the last one in the interceptor chain to the bean instance. Likewise, the control flow could be discontinued by each interceptor such that the call does not reach possibly all interceptors and the original bean instance. Besides inspecting and manipulating the submitted parameters, the same can be done with the return value of the call by all interceptors. To sum up, an interceptor intercepts method calls on enterprise beans to impose user-defined functionality. Interceptors can be specified at different levels of granularity. E.g., interception should only be performed for calls of certain methods of a bean, for all calls on a certain bean or even for all methods of all beans of a module. Interceptors can be added to and removed from beans until deployment time. After deployment, the interceptor configuration cannot be changed.

Transactions The EJB component model supports (distributed) transactions. Nevertheless, transactions must be flat, i.e., nested or child transactions are not supported. The bean developer must decide whether the enterprise bean will demarcate transactions programmatically in the business methods by explicitly beginning and committing trans-
actions or whether the transaction demarcation is performed by the container. Thus, the first alternative is called bean-managed-, and the second alternative being the default one container-managed transaction demarcation. With container-managed transaction demarcation, the container demarcates transactions according to instructions that are provided by the developer in metadata annotations or in the DD. Therefore, the transaction attribute can be specified for each bean method. It instructs the container whether it should include a call of a bean method in the transaction - if available - of the invoking client, execute the call in a new transaction started by the container, or run the method without a transaction. Nevertheless, for both alternatives, application-managed and container-managed transaction demarcation, the bean developer is not concerned with implementing transaction management or transaction protocols because the container must provide the necessary transaction support.

**Timer Service**  The **Timer Service** is a reliable and transactional notification service for timed events. It is managed and provided by the container and it allows stateless session beans and message-driven beans to register themselves for timer callbacks. These callbacks are invoked by the container on an instance of the registered bean and are scheduled at a specified time, after a specified elapsed time, or at specified intervals. The intention of the service is to enable the processing of notifications that certain temporal events have occurred without the need of having a client component that monitors the time and that invokes the callback methods if required. Nevertheless, the service is not intended for real-time events, but it enables the execution of business code without triggering the execution through a client.

**Persistence in the EJB Component Model**  The support of persistence within the EJB component model is specified by the **Java Persistence API** [DK06b]. This **Application Programming Interface (API)** defines a query language and an object/relational mapping to combine the world of object-oriented applications with the world of relational databases. An **Entity** is a lightweight persistent object. The persistent state of an entity is represented by instance variables and it is available to clients only through getter or setter methods or other business methods. The mapping of an entity and its instance variables to database tables and columns can be specified through metadata annotation within the entity classes or through XML files. In the same manner, it could be defined which classes represent entities.

A **Persistence Context** is a set of entity instances in which for any persistent entity identity there is a unique instance. Within the persistence context, the entity instances and their life cycle are managed by an **Entity Manager**. Each entity manager instance is associated with a certain persistence context. The entity manager provides methods to persist and remove entities, to synchronize entities with the database, to find entities by their primary key, and to query over entities using the specified query language or native **SQL**. Furthermore, an entity manager provides operations to acquire and release locks to support the optimistic locking technology in order to ensure consistency if entity instances are modified and synchronized with the database concurrently. To make use of persistent entities and of the persistence support provided by a container, enterprise beans interact with persistence contexts through entity managers. The set of entities that can be managed by a given entity manager instance, and therefore, are part of the same persistence context, is defined by a **Persistence Unit**. A persistence unit specifies the set
of all entity classes that are related or grouped by the application, and which must be
colocated in their mapping to a single database. These entity classes can be packaged
like any other class within an EJB module. An EJB module may have several persistence
units, but within all those units each unit must have a unique name. Persistence units
have to be defined in the `META-INF/persistence.xml` file as part of an EJB module.
An entity manager can be container-managed or application-managed depending who is
managing the life cycle of the entity manager and of the corresponding persistence context.
An entity manager for a persistence context can be obtained from an `Entity Manager
Factory`. When container-managed entity managers are used, the application does not
interact with the factory, but it obtains an entity manager from the JNDI or through
dependency injection. Thus, the interaction with the factory is done transparently to
the applications by the container. When application-managed entity managers are used,
the application obtains the entity manager factory from JNDI or through dependency
injection, and it uses this factory to manage the entity manager and persistence context
life cycles. In general, within an EJB environment, container-managed entity managers
are typically used.
5 mKernel: A Manageable Kernel for EJB-based Systems

The mKernel [BW07, BNVW08] system is a generic AC infrastructure for EJB-based autonomous applications. Regarding the architecture of an AC system, as described in section 3.2, it addresses the Managed Layer (see figure 2 on page 14). It enables the management of applications being allocated in the Managed Layer. Therefore, mKernel provides a comprehensive set of sensors and effectors that can be used by the Management Layer for higher level functions of autonomous management.

Figure 4 shows the conceptual view on a mKernel managed system from the perspective of an administrator. The buildings blocks of mKernel are the infrastructure itself, the preprocessing tool managizer, and the Application Programming Interface (API) that makes the Sensors and Effectors accessible for the Management Layer.

![Conceptual View on mKernel Managed System](image)

Figure 4: Conceptual view on an mKernel managed system from the administrator perspective

The infrastructure is a set of enterprise beans and it is itself an EJB-based application that is deployed in an EJB container. mKernel Managed applications have to run in the same container. The implementation of the sensor and effector interfaces, which constitute the API, is provided by the infrastructure. Details of this implementation are discussed in [BW07, BNVW08] and are not relevant for an administrator, because an administrator only needs to work with the API and the preprocessing tool. Therefore, only the API and the managizer are presented here in more detail.

A human administrator or a software entity that represents the Management Layer uses only the API to manage the application with the help of the infrastructure. This API provides a comprehensive set of sensor and effector operations that are used for the inspection and manipulation of managed applications. Through this API, mKernel provides a reflective view, the meta level, of the managed application, the base level. Both levels are causally connected [Mae87]. This reflective view enables the management of the application at three different levels of abstraction, namely the Type Level, Deployment Level, and the Instance Level. These three levels are described in the following.
• Type Level

The Type Level addresses information regarding types of the constituting elements of the managed application. These elements are artifacts that are the result of the development. Artifacts are the Java Archive (JAR) and the class files being part of the JAR file. A JAR file, which has been preprocessed by the managizer, is the unit that can be integrated into the mKernel infrastructure by loading it up to an mKernel administered repository. Within the repository, the artifacts and their properties are represented by an information model. An EjbModuleType represents a JAR file, which contains at least one enterprise bean. A bean is represented by an EjbType that corresponds to its class file. A bean provider defines the Simple Environment Entries, which are reflected by SimpleEnvironmentEntryType, and the receptacles of a bean. A receptacle at this level is an EjbReferenceType that declares a dependency to an implementation of a certain JavaInterfaceType. An EjbType can be either a MessageDrivenBeanType or a SessionBeanType. The latter one implements a certain JavaInterfaceType and provides this functionality as an EjbInterfaceType, i.e., as a business interface. Besides the integration and removal of a JAR file to or from the repository, the information model representing the artifacts of an application can be used for inspection, e.g., it is possible to find all SessionBeanTypes that provide a business interface of a certain JavaInterfaceType.

• Deployment Level

To deploy an instance of an EjbModuleType, called EjbModule, a concrete configuration of a module is required. This is addressed by the Deployment Level. An EjbModule can be deployed in the container and it includes EnterpriseBeans, which are either SessionBeans or MessageDrivenBeans. The configuration of a module comprises the setting of values of the SimpleEnvironmentEntries of its beans, and the connection of receptacles, represented as EjbReferences, to provided interfaces, represented as EjbInterfaces, of SessionBeans. An EjbReference can be bound to an EjbInterface if their types refer to the same JavaInterfaceType. An EjbModuleType can be deployed more than once, but each deployed module is represented by an EjbModule. At the Deployment Level the life cycle of EjbModules can be managed according to the Java Enterprise Edition 5 Deployment API Specification, Version 1.2 (JSR-88) [Doc06], that is described in section 4.2.2. Furthermore, the bindings between EjbReferences and EjbInterfaces can be reconfigured. Such reroutings of connections, i.e., binding a reference to a new target, are specified at the Deployment Level, but they may have different consequences on the Instance Level. Lazy rerouting maintains already established connections between bean instances and only newly created ones point to instances of the new target. In contrast, non-lazy rerouting affects newly created connections and replaces already established ones between bean instances. Of course, the new target session bean of the connection has to provide the same JavaInterfaceType as the original target session bean. Which type of rerouting should be used, depends, among others, on the bean that is the target of a connection. If the target is a stateless session bean, then existing connections between bean instances can be manipulated without difficulty, because target bean instances retain no client-specific state and both beans, the original and the new target, provide the same functionality. On the other hand, if the target bean is a stateful session bean, rerouting an existing connection might cause problems because client-specific states of target in-
stances would get lost unless the internal states are not transferred from the original target instances to the new target instances. The differences between stateless and stateful session beans are discussed in section 4.2.1. Comparable to connections among beans, the values of SimpleEnvironmentEntries can be modified at the Deployment Level. It can be distinguished whether only newly created bean instances are provided with the changed value or, in addition, that also already existing instances are supplied with the changed value.

Finally, mKernel enhances the EJB Interceptor facility (see section 4.2.3) by runtime integration and removal of interceptors realized as session beans to or from session beans whose calls should be intercepted. Thus, even at runtime mKernel based interceptors can be attached to and removed from SessionBeans.

- Instance Level

The Instance Level addresses bean instances and interactions among them. Thus, the actual execution of the managed application is considered. A concrete instance of a deployed EnterpriseBean is represented by an EnterpriseBeanInstance, which is either a SessionBeanInstance or a MessageDrivenBeanInstance. The arrival of a message at a MessageDrivenBeanInstance or a method invocation on a SessionBeanInstance are represented as Calls, which can be a MessageCall or a BusinessCall, respectively. Super- and sub-call relations between calls are captured, which facilitates the analysis of call chains. Furthermore, the life cycle state of a bean instance is monitored and retrievable.

Finally, basic support for the reconfiguration of EJB modules is provided. mKernel may ensure a quiescent state and it enables a state transfer (compare with section 2.3). The whole managed application or parts of it can be placed into a quiescent state. The part of the system that should be quiescent is called QuiescenceRegion. This region can be specified at different levels of granularity. Thus, a quiescence region may consist of a set of EjbModules, including all of their SessionBeans and the corresponding SessionBeanInstances, a set of SessionBeans and their SessionBeanInstances, or even only a set of SessionBeanInstances. To reach a quiescent state, mKernel uses a dynamic algorithm (see section 2.3) that passes relevant call chain information as implicit parameters of a call along the chain. If a quiescence region has been defined, and this region is currently engaged in reaching the quiescent state or is already quiescent, call chain analysis determines whether or not a method call that actually occurs should be blocked. Quiescence is required for the consistent replacement of instances of stateful session beans. Furthermore, the transfer of the Conversational State from the replaced instance to the replacing one is required. Therefore, mKernel enables the access to the state of a stateful SessionBeanInstance to extract or inject the state. It is not required that the bean developer has to implement certain interfaces to enable the state extraction or injection. Furthermore, after replacing a stateful session bean instance, mKernel is able to replace all the references to the replaced instance with references to the replacing one. Therefore, in the information model of the API, a HoldingReference represents a quiescent SessionBeanInstance which is either part of the QuiescenceRegion, or which has been created by mKernel as a completely new instance, that will replace an instance of the QuiescenceRegion. The actual state injection or extraction of a stateful SessionBeanInstance is performed through the corresponding HoldingReference.
Based on this multi-level view provided through the API and the possibility to navigate between the levels, subtle management operations are possible to inspect and manipulate the structure and behavior of the managed application. Especially with the support of quiescence and state transfer, the sensors and effectors of the mKernel API constitute a basis to perform manifold reconfigurations. As discussed in section 4.2.1, the EJB standard limits the configuration of Simple Environment Entries and of connections among enterprise beans to the deployment time, but mKernel enables the modifications of them at runtime. Nevertheless, the EJB specification is not violated or restricted by mKernel.

As mentioned above, before a JAR file can be integrated into the mKernel managed system, it must be preprocessed by the managizer. Regarding the information model of the API, a JAR file is an EjbModuleType. Therefore, the managizer, receives an EJB module type that must be compliant to the EJB specifications [DK06a, DK06b] as input. The module, the included enterprise beans, and the Deployment Descriptor (DD) are analyzed and enriched to enable the autonomous management of the beans through the mKernel infrastructure. Enrichments incorporate management aspects into the module by adding classes, which are provided by the tool, into the module and by modifying the byte code of original classes, which are provided by the application developer. Though these enrichments, the module and its beans remain compliant to the EJB standard. The output of the tool is an enriched EJB module type that might be integrated into the mKernel system through the API (see figure 4 on page 25).

The preprocessing of a JAR file by the tool is executed without requiring additional information from the developer or user assistance. Bean Providers do not even have to consider the management of their beans during implementation or follow special guidelines beyond those of the EJB standard to enable the manageability of their beans through mKernel. Thus, the developer can solely focus on the application logic and is not concerned with management aspects, which are woven into the EJB module by the preprocessing tool. This approach maintains the idea of Separation of Concerns to separate application logic from crosscutting concerns, such as management. Additionally, the infrastructure for autonomous management provided by mKernel is realized as a plugin for the Glassfish Application Server [gla] without the need to adapt the application server implementation. Thus, neither the Enterprise Bean Provider, Application Assembler, EJB Server Provider, EJB Container Provider, nor the Persistence Provider (see section 4.2.2) are concerned with enabling the management of EJB-based applications through mKernel. Therefore, the EJB component model and AC management aspects are integrated seamlessly and the Deployer and System Administrator may use mKernel for comprehensive management of EJB-based applications.
6 A Model for Autonomous Reconfiguration of EJB-based Enterprise Applications

The approach of seamless reconfiguration, which is presented here, addresses component-oriented Enterprise Applications (EAs) being realized using EJB technology. The modular architecture of component-based software systems, as discussed in section 4, considers the complexity of EAs and facilitates the development and deployment of components. This modularity assists also the reconfiguration of EAs, because each component can be reconfigured individually without reconfiguring the whole system. This reduces, e.g., the complexity of an upgrade cycle. Nevertheless, the EJB component standard, presented in section 4.2, addresses only the development and deployment of EJB-based EAs, but does not specify aspects regarding the execution of applications, like, e.g., the inspection and manipulation of applications at runtime. Thus, post-deployment aspects are completely left to EJB Container Providers and support for comprehensive runtime management or even for reconfiguration of EJB-based applications might be poor. This issue is discussed in section 4.2.2 with the limitations of the EJB specifications. Consequently, despite the modularity of component-based software systems, reconfiguration of EJB-based EAs remains a challenge, especially when reconfiguration should occur at runtime. Therefore, an administrator should be supported to facilitate reconfiguration, e.g., by automating as many reconfiguration tasks as possible. Here, the vision of Autonomic Computing (AC) that is described in section 3 is helpful. It is dealing, among others, with this issue of automating system maintenance tasks in order to disburden human administrators. With the mKernel system a generic AC infrastructure is available, which is described in detail in section 5. To facilitate runtime reconfiguration, the mKernel system is helpful for two reasons. First, it supports the comprehensive management of EJB-based applications, even at runtime, which is neglected by the EJB standards. Second, these facilities are provided through an Application Programming Interface (API) that abstracts from fine-grained management tasks in order to simplify and enable the management of EJB-based applications at a higher level. Additionally, the facilities of mKernel provide runtime support for parameter and compositional adaptations that are described in section 2.2. Thus, mKernel is a basis for the runtime reconfiguration of EJB-based EAs. With this approach using EJB and mKernel, the concept of Component Orientation (CO) and the vision of AC are combined seamlessly. Moreover, McKinley et al. [MSKC04] consider three technologies, namely Separation of Concerns, Computational Reflection, and Component-based Design as keys to reconfigurable software design, which are applied by mKernel or EJB:

- **Separation of Concerns** enables the separate development of the functional behavior, i.e., the business logic, of an application and of the code for crosscutting concerns, like, e.g., quality of service or management. This separation simplifies development and maintenance and it promotes software reuse. EJB follows this idea by requiring from an EJB container to provide container services, like, e.g., transaction and persistence support, that may be used by enterprise beans. Thus, bean providers do not have to implement such concerns for every bean or module, but they can concentrate on realizing the business logic. mKernel maintains this idea by providing support for the management of enterprise beans transparently to bean providers. Consequently, bean providers do not have to contribute code at development time to enable the management of their beans.
• **Computational Reflection** refers to the ability of a program to reason about itself and to alter its own behavior [Mae87], which facilitates adaptations. EJB applications have themselves no reflective capabilities. Even the EJB specification limits the use of reflection features being available with the Java programming language within enterprise bean code [DK06a]. Nevertheless, mKernel adds reflective capabilities to managed EJB-based applications without violating the EJB specification.

• The modularity and loose coupling of **Component-based Design** facilitates runtime reconfiguration if late binding is supported, which is addressed by Java and EJB. Thus, changes in the architecture of a component-based system, like, e.g., the addition or removal of a component, is feasible without having to change the overall system. Focusing only on EJB-based applications and their reconfigurations, the requirement of component-based design is met.

Consequently, EJB and mKernel are a promising foundation for a reconfiguration approach. This approach focuses on the reconfiguration of EJB-based systems, i.e., it addresses the application of changes to systems (compare to section 2). Thus, only a part of an iteration of a software evolution is considered. An iteration of a software evolution can be mapped to the control loop of AC systems that is presented in section 3.2, and it may consist of the following activities: detecting a critical incident within an EJB-based application (Monitoring), analyzing the incident and identifying its cause (Analysis), finding a solution for the incident (Planning), and finally, applying the solution (Execution). Focusing on reconfiguration, this approach starts at the point when a solution has been found and its subject is to apply the solution. Thus, the **Execution** stage of the control loop is addressed.

This approach supports runtime reconfiguration that applies anticipated and unanticipated changes to an EJB-based application without the need to shutdown the system. Thus, the application remains available to its clients. Applied changes comprehend parameter and compositional adaptations, which are discussed in the context of EJB in section 6.1, such that, among others, the architecture of an EJB-based application can be modified. Structural changes of an architecture are critical for the consistency of the application. Therefore, this approach considers the consistency during a reconfiguration, such that a consistent application state survives a reconfiguration. Moreover, changes are applied seamlessly, such that clients of the reconfigured application may notice delays in the response time but no failures. Thus, a reconfiguration is almost transparent to clients. Moreover, the disruption of the system may be influenced by the way an application is reconfigured. This approach does not limit the ways to perform a reconfiguration, i.e., it does not prescribe a certain reconfiguration strategy. Instead, this approach provides several reusable strategies and it is extensible w.r.t. the opportunity to integrate new ones or to modify existing ones. Furthermore, code written by an administrator that performs certain reconfiguration tasks may be integrated into the approach. This flexibility to extend the capabilities of the approach addresses the need to fulfill different requirements of concrete reconfigurations. These requirements depend, among others, on concrete reconfiguration needs and on concrete applications that should be reconfigured. A strategy is reusable and it abstracts from fine-grained reconfiguration tasks. Thus for a concrete reconfiguration, an administrator is freed from handling these tasks because he or she is only concerned with choosing an appropriate strategy which serves as a template that eases the subsequent preparation and execution of a reconfiguration. Thus, this approach simplifies a reconfiguration of an application for an administrator. Furthermore, a reconfiguration is executed autonomically without any need for interaction with an administrator, such that
the degree of automation in system reconfiguration increases. Additionally, a potential reconfiguration of an application need not to be anticipated during the development of the application. Thus, this reconfiguration approach is transparent to the Enterprise Bean Provider and Application Assembler, which maintains the idea of separation of concerns.

The flexibility and extensibility of the approach are enabled through a modular reconfiguration model that specifies the design of the approach. The building blocks of the model are reusable and some of them are even replaceable or extensible. Thus, instead of starting from scratch with each concrete reconfiguration, the administrator can rely on already existing entities of the building blocks. An overview of model is given in section 6.2 and it is described in detail in the sections 6.3 and 6.4. This reconfiguration model has been implemented (see section 7) and it provides four reusable strategies, which are presented in detail in section 8. The implementation consists of an API and of an infrastructure that realizes the API. This API supports an administrator in creating, modifying and using strategies, in preparing and executing a reconfiguration, and in implementing and integrating custom code into the infrastructure that may be executed during a concrete reconfiguration.

Before starting with the reconfiguration model, a discussion of parameter and compositional adaptations in the context of EJB is given in the next section.

6.1 Parameter and Compositional Adaptation of EJB Applications

To enable various reconfigurations of EJB-based applications in order to correct, adapt, or perfect applications, parameter and compositional adaptations are supported. These adaptations are introduced in section 2.2.

In the context of EJB, parameter adaptation is performed through setting and changing values of Simple Environment Entries provided by an enterprise bean. E.g., a shipping company runs a system that computes routes for the delivery of goods. The objective is to find the least-cost route, which is known as the Traveling Salesman Problem. Therefore, an enterprise bean implements, e.g., two algorithms and it provides a Simple Environment Entry that enables the switch of the algorithm being used. The first algorithm might be a brute-force search that always computes the optimal route but at high costs in terms of performance. The second algorithm might be a more efficient one, like, e.g., a greedy algorithm. Nevertheless, the greedy approach does not guarantee to find the optimal route. Consequently, there is a trade-off between the effectiveness and the efficiency when selecting one of the algorithms. If the problem size is appropriate, both algorithms can be applied and a decision for one of them might be based on the workload of the system. I.e., if the system gets overwhelmed with requests, it might be sufficient not to get the least-cost route, but to get a reasonable route on time. Thus, in case of an increased workload, the greedy algorithm satisfies the requirements. Otherwise, the expensive brute-force algorithm is selected. Thus, depending on the workload, an administrator or a piece of software, monitoring the workload, changes the value of the Simple Environment Entry accordingly in order to switch the algorithm that should be used by the bean. To support such a scenario, the bean must implement several algorithms, provide a Simple Environment Entry, and react appropriately to modifications of the entry value. Therefore, the adaptations are anticipated during bean development. Hence, parameter adaptations through Simple Environment Entries only supports anticipated changes.
In contrast, compositional adaptation is more powerful and it covers unanticipated changes. It addresses the deployment and undeployment of EJB modules and the manipulation of connections among enterprise beans. Thus, the architecture of an application, consisting of several EJB modules, might be changed. New EJB modules can be integrated into the application and modules, which are not needed any more, can be removed from the application. E.g., a shop application may consist of modules, each of them covering an aspect, like, e.g., customer management, product management, and accounting. After the web-shop has become operational, it can be extended, e.g., with features of a recommender system to recommend products to customers. Thus, a module for recommendations might be integrated into the application and this module might interact with the other modules of the shop application.

Furthermore, deployment and undeployment of modules enable the replacement of a module. The replacing one may be a more efficient implementation than the replaced one, it may correct failures in the replaced module, or it may provide additional functionalities than the replaced one. All structural changes of an architecture are critical for the disruption to and consistency of an application, which is described in section 2.3. System disruption is minimized by performing parameter and compositional adaptations at runtime without shutting the system down. The concepts of quiescence and state transfer guarantee that structural changes do not break the consistency of an application. Thus, the approach considers the opportunity to place modules of an application and their constituting beans and bean instances, which are affected by structural reconfigurations, into a quiescent state. Furthermore, if instances of stateful session beans need to be replaced by instances of the replacing bean, their Conversational States can be transferred. Considering instances of enterprise beans and interactions among them, the approach addresses the Instance Level of the EJB-based applications, while the Deployment Level is incorporated, e.g., through the deployment and undeployment of EJB modules and the manipulation of connections among beans. To sum up, the desired objectives of a reconfiguration, namely minimizing system disruption and preserving consistency of reconfigured applications are addressed when structural changes are applied.

6.2 A Conceptual Overview of the Reconfiguration Model

The reconfiguration model specifies the design of the approach. The model is structured modularly in order to enable the required flexibility to support various types of changes and various reconfiguration strategies and to be extensible by an administrator. Extensibility of the model is important such that the model is able to fulfill specific requirements of a concrete reconfiguration that cannot be met generically. Additionally, the building blocks of the model are reusable to help an administrator with a reconfiguration of a concrete application. This avoids, e.g., that he or she always needs to write code from scratch to reconfigure a concrete reconfiguration, which simplifies his or her work. The model is shown in figure 5 on page 33.

The model is based on so called Steps. Each step is reusable and specifies a certain reconfiguration task, like, e.g., the deployment of an EJB module or the establishment of connections between enterprise beans. A complete and comprehensive set of steps is specified and provided by this approach. Steps are realized by Step Executors, or Executors for short, that might use the mKernel API. For almost each provided step, the implementation of the approach provides default executors. Nevertheless, administrators have the
freedom to implement *custom* executors for arbitrary steps. These custom executors can be integrated into the model and they may replace the default ones. In this way, special requirements for performing concrete changes to a concrete application can be fulfilled.

The provided steps are the basis for the *strategies*. Therefore, steps are combined into generic and autonomous reconfiguration procedures. A procedure is an arrangement of steps that together specify a certain kind of reconfiguration. The reusability of each step, the possibility to use only the relevant steps, and the flexibility in ordering the steps enable various combinations of steps into procedures. Each procedure realizes a certain reconfiguration strategy, i.e., a certain way to perform a reconfiguration. Therefore, in addition to the strategies that are already *provided* by the approach, administrators can develop *custom* strategies, that may be derived from others or that may be completely new ones. Even a dynamic arrangement of steps during runtime is conceivable, resulting in *ad-hoc* strategies. The ability to have several strategies instead of prescribing one certain strategy accounts for the various kinds of changes that are conceivable and that should be done to an application. Theses changes have different origins, namely, corrective, adaptive, and perfective (see section 1.1). Thus, it is difficult to predict all possible changes that may happen (cf. unanticipated changes in section 2.1) and how they should be applied. Thus, limiting the way how a reconfiguration is applied is not appropriate. Different types of changes may require different ways and therefore different strategies, also depending on the concrete application.

For a concrete reconfiguration, a strategy serves as a template for easing the planning and execution of the reconfiguration. To reconfigure an application an administrator must provide a *Reconfiguration Plan*, or *Plan* for short, i.e., a strategy must be selected, instantiated and configured. Afterwards, a configured plan can be executed autonomically. The execution of a plan performs the reconfiguration.

The following section goes into the details of the reconfiguration model by describing each part of the model and their relation among each other in detail.

### 6.3 The Reconfiguration Model in Detail

After describing the *steps* that are specified and provided by this approach, the concept of *step executors* is discussed. Then, the combination of steps into *procedures* is presented. A procedure is a cooperation of steps that realizes a certain reconfiguration strategy. Finally, the execution of a reconfiguration by running a *plan* is addressed.
6.3.1 Specification of the Provided Set of Steps

A step is the basic unit of a reconfiguration procedure. It is reusable and it specifies a certain reconfiguration task. To meet various reconfiguration needs, a comprehensive set of steps has been specified, which are provided by the approach. This set can be divided into seven groups. The bold terms represent the names of each group. Steps of the first group address the EJB Module Life Cycle. The group Parameter and Connection cover Simple Environment Entries of and connections among enterprise beans, respectively. The next two groups comprehend the concepts of Quiescence or State Transfer. A reconfiguration of a Database is considered, and finally, Supporting Steps are specified that may assist the other steps or that support the cooperation of steps. In the following, all groups and their steps are described by presenting the reconfiguration tasks of each step. Within the subsequent paragraphs, names of interfaces refer to the mKernel API that is presented in section 5.

EJB Module Life Cycle The steps of this group, namely Module Creation, Module Deployment, Module Starting, Module Stopping, Module Undeployment, and Module Destruction, manage the life cycle of an EJB module. Each step performs a transition in the life cycle state of a module. This life cycle is depicted in figure 6. It is based on the Java Enterprise Edition 5 Deployment API Specification, Version 1.2 (JSR-88) [Doc06], but it is extended to reflect specifics of mKernel.

![Ejb Module Life Cycle Diagram](image)

Figure 6: Ejb Module Life Cycle

The following list describes the steps of this group and the specification of their tasks.

- **Module Creation**: mKernel offers a repository that administrates EJB modules at the Type and Deployment Level. EJB modules at the Type Level (EjbModuleTypes) are modules that have been loaded up to the mKernel repository. Therefore, an EjbModuleType can only be in the state UPLOADED. Otherwise, it is not known to mKernel. To deploy an EJB module, a configurable EJB module at the Deployment Level is necessary. Therefore, this step retrieves an EjbModuleType at the Type Level from the repository and creates instance of it. The resulting instance is an EjbModule at the Deployment Level in the state EXISTS that, e.g., might be configured and deployed afterwards.

- **Module Deployment**: This step deploys an EjbModule that is in state EXISTS in an EJB container of the target operational environment. After deployment, the module is not runnable and not available to clients, but it is integrated into the container. It is in the STOPPED state.
- **Module Starting**: Using this step, an EjbModule in state STARTED can be started. After that, the module and its beans are runnable and available to clients. The module switches to the state STARTED and it becomes operational if clients start requesting and interacting with instances of its bean.

- **Module Stopping**: This step makes a running EjbModule unavailable to clients and stopped. After that, the module and its beans are no longer runnable. Consequently, the operations of the module are terminated. The module is now in state STOPPED.

- **Module Undeployment**: Using this step, an EjbModule in state STOPPED can be undeployed and removed from the EJB container of the target operational environment. After that, the module at Deployment Level is no longer associated with the EJB container. A representation of the module only exists in the mKernel repository. Therefore, the module is in state EXISTS.

- **Module Destruction**: An EjbModule in state EXISTS that is not needed any more might be destroyed through this step. This destruction removes the representation of the module from the mKernel repository. The EjbModule moves to the state DESTROYED which indicates the end of its life cycle. Thus, the module can not be deployed any more. Nevertheless, the EjbModuleType from which the EjbModule has been derived remains in the repository.

**Connection**  
This group of steps addresses connections among enterprise beans. A connection is a binding from a receptacle of a bean to a provided business interface of a session bean. In terms of the mKernel API, it is a binding from an EjbReference to an EjbInterface. Thus, a binding connects two corresponding enterprise beans. The following steps configure and reconfigure connections among beans and consider the Deployment and the Instance Level of an EJB-based application.

- **EJB References Connection**: The receptacles of EnterpriseBeans of a module in state STOPPED must be connected to corresponding interfaces provided by other SessionBeans. These connections should be defined before the module that contains the beans with disconnected receptacles has been started. Moreover, receptacles of the target SessionBeans of connections have to be also bound to provided interfaces, recursively. Additionally, all modules being part of the transitive closure obtained though following connections among beans have to be started in an appropriate order, which is performed by other steps. Otherwise, the EnterpriseBeans would become available to clients, but their dependencies to other beans are not fulfilled comprehensively. Consequently, beans might not be able to work appropriately. Nevertheless, it is possible that some receptacles need not to be connected, e.g., if they are optional ones or if only a subset of the provided functionality of a bean will be used that does not require certain receptacles of the corresponding bean to be bound [AP04]. To sum up, the task of this step is to connect receptacles of beans, whose modules are in state STOPPED, to provided interfaces of session beans.

- **Newly Established Connection Rerouting**: The tasks of this step is to reroute connections at the Deployment Level to a new target, i.e., to a different SessionBean that provides an EjbInterface of the same JavaInterfaceType like the provided
EjbInterface of the original target SessionBean. This rerouting affects only the establishments of connections and not already existing connections among bean instances. Thus, this step performs the lazy rerouting of connections that is provided by mKernel and described in section 5.

- **Existing Connection Rerouting:** In contrast to the step *Newly Established Connection Rerouting*, this step also considers already existing connections between bean instances. Thus, rerouting affects all connections, i.e., connections that will be created and those that already exist. Therefore, this step performs the non-lazy rerouting of connections that is provided by mKernel and described in section 5.

The steps of the group **Connection** together with the steps of the group **EJB Module Life Cycle** address compositional adaptation, because using these steps, an architecture of an application might be modified structurally.

**Parameter** This group covers parameter adaptations of EJB-based applications, i.e., it addresses *Simple Environment Entries* being provided by enterprise beans. Therefore, one step is defined.

- **Simple Environment Entry Modification:** This step sets or changes the values of SimpleEnvironmentEntries of deployed EnterpriseBeans as specified. Either, this step can be used for entries of beans whose EjbModules are in state STOPPED or in state STARTED. The first case covers initial settings of entry values before the corresponding EjbModule is started. Thus, all entries have been assigned values and, after the EjbModule has been started, bean instances are able to work appropriately using these values. The second case modifies values of entries because the EjbModule is in state STARTED and there might exist bean instances that currently use the original entry values. If a value of an entry is changed, it should be distinguished whether these instances should be provided with the changed value or not. But, bean instances that are instantiated after the change of the entry value should always be provided with the new value.

**Quiescence** The concept of quiescence is addressed by this group of steps. As discussed in section 2.3, consistency of an application should be preserved during and after the reconfiguration of an application. This may require that the part of an application which is affected by the reconfiguration should quiescent before changes are applied. A quiescent part of an application is called **QuiescenceRegion**. The life cycle of a quiescence region, which is depicted in figure 7 on page 37, is managed by the following steps.

- **Quiescence Region Declaration:** With this step, a QuiescenceRegion is defined. The region consists of those EjbModules, SessionBeans, or SessionBeanInstances that should be transferred into a quiescent state at a later point in time. If an EjbModule is added to the region, then all of its session beans and all of the instances of those beans will become quiescent. Likewise, if a SessionBean is added to the region, all of its instances will become quiescent. Furthermore, it is possible that only certain SessionBeanInstances should become quiescent, i.e., it need not to be the case that all instances of a certain SessionBean have to be placed into a
quiescent state. After the definition of a quiescence region through this step, the region is in its initial state OFF.

- **Quiescence Region Tracking**: For a reconfiguration of an EJB-based applications, it might be necessary to handle existing bean instances (see section 2.4). Therefore, those instances must be known and they must be accessible. This step initializes the tracking of those SessionBeanInstances that are part of the QuiescenceRegion. The QuiescenceRegion moves to the state TRACKING. Consequently, the instances that have been tracked because of this step might be accessed and reconfigured afterwards.

- **Quiescence Region Blocking**: This step initializes the blocking of relevant method calls between EnterpriseBeanInstances, such that the whole QuiescenceRegion is able to become quiescent. How quiescence can be reached is described in section 2.3. Nevertheless, already running method calls on EnterpriseBeanInstances being part of the QuiescenceRegion must be finished. As long as at least such a call is still running, the QuiescenceRegion is in the state BLOCKING.

- **Waiting for Quiescence**: As mentioned in the previous step Quiescence Region Blocking, it may take some time until already running method invocations have finished and the QuiescenceRegion reaches a quiescent state. This step checks whether or not quiescence has been reached. It delays the further execution of a reconfiguration until the QuiescenceRegion moves from the state BLOCKING to the state QUIESCENT. In contrast to the other steps of this group that cause actively transitions of a QuiescenceRegion state, this step waits until a QuiescenceRegion changes its state itself. Thus, this step only monitors the state of a QuiescenceRegion.
• **Quiescence Region Release**: If the quiescent state of a QuiescenceRegion is not required any more, this step resolves the quiescent state. As a consequence, blocked method invocations are released and resume their operation and newly initialized invocations are not blocked. Therefore, interactions with and within bean instances of the QuiescenceRegion are not affected any more. The QuiescenceRegion switches to its initial state OFF.

• **Quiescence Region Destruction**: If the QuiescenceRegion that has been defined with the step *Quiescence Region Declaration* is not needed any more, this step is used to destroy it. The QuiescenceRegion moves to the state DESTROYED, which indicates the end of its life cycle. Afterwards, the QuiescenceRegion does not exist in the mKernel system anymore.

**State Transfer** In addition to quiescence, a state transfer might be required to ensure consistency of an application if running elements should be replaced. This issue is discussed in section 2.3. In case of EJB, if stateful session bean instances should be replaced, their Conversational States have to be transferred from the original instances to the replacing ones. Such a state transfer is enabled by this group of steps.

• **Conversational State Extraction**: This step extracts Conversational States from stateful session bean instances. These instances are part of a QuiescenceRegion. The region should be in the state QUIESCENT to ensure consistency and to prevent multiple invocations from being active on one instance at the same time. The latter aspect is important because bean instances are per definition non-reentrant (see section 4.2.1) and if more than one method call is active on one bean instance at the same time, an exception will be thrown. In case of an exception the container destroys the bean instance, which is not desired because the instance and its state would get lost and the application consistency might be affected. Therefore, during the extraction of the state of an instance, no business methods should be invoked on the corresponding instance, which is guaranteed if the instance is quiescent. Furthermore, a quiescent state guarantees consistency of reconfigured applications because of another reason. E.g., it prevents that, after extraction of the state from a stateful session bean instance, a business method will be invoked on this instance, which might change the conversational state, such that the extracted state and the state held by the bean instance might diverge.

• **Conversational State Conversion**: This step is used if the state types of the replacing and of the replaced stateful session beans do not match. There might be syntactic or semantic differences that circumvent a simple copying of the state from a replaced instance to a replacing one. Therefore, the states, which have been extracted with the step *Conversational State Extraction* can be adapted, modified, enhanced or reduced such that they match the state type of the replacing stateful session bean. Consequently, after the conversion, the extracted states should be of an appropriate type such that they can be injected into replacing stateful session bean instances.

• **Conversational State Injection**: This step injects Conversational States, which have been extracted with the step *Conversational State Extraction* and which might have been converted with the step *Conversational State Conversion*, into replacing
instances of stateful session beans. Thus, this step is only responsible for setting states into stateful session bean instances.

- **Reference Publication**: After a stateful session bean has been replaced and all running instances of the replaced bean have been replaced to instances of the replacing stateful session bean, all clients holding references to replaced instances must be provided with references to the corresponding replacing instances. In other words, the references to replacing instances must be published in the system, such that clients only hold valid references. No client should hold any reference to a replaced stateful session bean instance. This exchange or publication of references is the task of this step.

The partition of the state transfer into the three steps for extraction, conversion, and injection aims to improve the reusability. The steps of state extraction and injection are independent of a concrete application and therefore, their implementations might be completely reusable. In contrast, the operations performed within the state conversion step usually depend on the business logic of the reconfigured application. Consequently, an implementation of this step makes use of business objects of a concrete application, such that, in case of a reconfiguration, a specific implementation has to be provided for each concrete application.

**Database**  Managing the *Conversational States* of stateful session beans being part of the EJB module that should be reconfigured might not be sufficient. The relevant application state of an EJB module might cover these *Conversational States* and the data stored in the database the EJB module is working on. Therefore, reconfiguration of a database might be necessary, which is addressed by this group of steps. Different database reconfigurations are conceivable, like, e.g., a migration of a database to a new server or changes of the database schema. Nevertheless, this approach currently does not cover the reconfiguration of a database. Therefore, the following steps are not specified exhaustively and they serve as kind of placeholders for database reconfigurations in order to keep the set of step comprehensive. The design of these steps is comparable to the steps that address the transfer of *Conversational States*.

- **Database Extraction**: This steps extracts data from a database.
- **Database Conversion**: Adaptations, modifications, enhancements or reductions of the data being extracted from a database are performed by this step.
- **Database Injection**: The injection of the extracted and maybe converted data is considered by this step.

**Supporting Steps** Finally, the last group consists of three steps that specify assisting tasks that support the tasks of the other steps that have been described above.

- **Delay**: As a reconfiguration is a cooperation of several steps, it may be required that the reconfiguration process should pause after having executed some steps, but before the next steps will be performed. This is the intention of this step. It delays the execution of the reconfiguration for a specified time interval. One
possible application of this step is to define the temporal length of the TRACKING phase of a QuiescenceRegion. The step Quiescence Region Tracking initializes only the tracking (see above), but it does not specify the time interval during which bean instances are tracked. Using this step after initializing the tracking, such a time interval might be specified, before the QuiescenceRegion, e.g., enters the BLOCKING state through the step Quiescence Region Blocking.

- **Analyzer**: This step performs arbitrary analysis of the application that should be reconfigured or of the application environment. E.g., the structure of an application can be inspected. The results of an analysis are required by other steps for their appropriate execution. As some knowledge gained through a certain analysis might be needed by several other steps, this analysis should not be repeated by each of those steps. In contrast, the analysis can be outsourced to an own step. This step is intended for such scenarios.

- **Adapter**: Performing a complex reconfiguration requires that different steps exchange information. Such a cooperation is only possible if the steps are compatible to each other. Therefore, the integration of steps might be necessary to enable reconfiguration, e.g., to adapt data that is sent from one step to another, because the interfaces of both steps do not match. Such integration tasks are performed by this step. From a technological point of view, an adapter step provides some glue to connect other steps. Thus, the Adapter step can be compared to the Adapter pattern [GHJV04].

### 6.3.2 Step Implementations: Executors

A step executor is an implementation of a step, i.e., it implements the tasks that are specified for a certain step. A step executor might use the mKernel API whose sensors and effectors are the basis for the inspection and manipulation of the application that should be reconfigured. Furthermore, it is conceivable and possible that other APIs and tools are used by steps. E.g., the steps for database reconfiguration may utilize APIs and tools from a database vendor or persistence provider. Steps are introduced by the reconfiguration model (see section 6.2 and figure 5 on page 33) because they abstract from a concrete implementation. Therefore, exchanging the implementation of a step is facilitated.

To be (re)usable, an executor has to specify the step it is implementing, a description, and it may specify input and output parameters. The realized step has to be provided to map an executor to its step or to retrieve all executors that implement a certain step. Thus, associations between a step and its implementations are built. A description should provide meta information about details of what the executor is actually doing. A step gives only a generic description of its tasks, but an executor may make some assumptions that need to be fulfilled in order to be used appropriately. Finally, a specification of an arbitrary number of parameters might be provided. Comparable to a parameter of a method that is defined in an interface, a specification consists of the data type of a parameter, which addresses syntactic aspects, and of the documentation about the semantics of a parameter. Input parameters represent information that are required for an appropriate execution of an executor. During execution, the input values are processed by an executor. Likewise, information about execution results are provided through outputs. Output values are generated during execution. Furthermore, parameters can be declared as optional, i.e.,
values for them do not have to be provided at runtime. Optional inputs should have a default value corresponding to some default behavior of the executor, which could be changed by indicating an input value other than the default one. Optional outputs need not to be available after an execution, e.g., because of a fork in the activity of an executor, the code that would have generated the optional output value has not been reached. Thus, other executors should not rely at runtime on the availability of values for optional outputs.

Finally, figure 8 shows the schematic view on a step. The SampleStep is implemented by the SampleStepExecutor. This executor has a description and it requires two input parameters, namely in1 and in2 for its appropriate execution. As a result of its execution, the executor returns the two output parameters out1 and out2. For the visualization of required input and provided output parameters, the notation of the Unified Modeling Language (UML) [Obj07] for required and provided interfaces of components is used. Thus, in this figure, parameters should not be confused with interfaces.

Figure 8: Schematic view on a step and on an associated step executor

To perform a certain step, a corresponding executor must be associated to the step. The step has to require or provide the input or output parameters of its associated executor. This is shown in figure 8 through the ports that belong to a step and that represent executor parameters. Thus, when it comes to the execution of a step or precisely of the corresponding executor, the distinction between step and the associated step executor gets blurred and both can be seen as one unit. Nevertheless, the distinction has the advantage to separate the design of a reconfiguration from the execution of a reconfiguration. For the design, an administrator is only concerned with the concepts of the steps and not with step implementations. He or she decides what steps should cooperate and should be used for which parts of the application that should be reconfigured. Afterwards, to perform the reconfiguration, execution details must be considered. Each step that has been selected must be associated with an executor. Perhaps, appropriate executors are already available or must be implemented to meet the requirements of the design. Thus, steps can be associated to the design of a reconfiguration, while executors are important for the implementation and execution of a reconfiguration.

6.3.3 Reconfiguration Procedures and Strategies

The steps, presented in section 6.3.1, can be combined into generic reconfiguration procedures. Each procedure realizes a certain reconfiguration strategy, i.e., a certain way to perform a reconfiguration. Thus a strategy is a cooperation of several steps. To have a concrete strategy, for each of its steps an executor must be assigned. Thus, it is known,
which inputs and outputs are required or provided by a step because of the associated executor. Figure 9 shows an example of a concrete strategy, called SampleStrategy, that consists of four steps. To each of those steps an executor has been assigned. The executors are not depicted in the figure, because for each step only an external view, i.e., the required and provided parameters, is relevant (compare to figure 8 on page 41). To enable the cooperation of steps at runtime, information constituting of parameter values have to be passed between steps or precisely between the corresponding step executors. Therefore, parameter mappings have to be specified for existing parameters. Outputs of a step can be mapped to inputs of subsequent steps. E.g., in figure 9, the output step2_out1 of Step2 is connected to the input step3_in1 of Step3. Optional parameters, like, e.g., step1_in2 or step1_out2, need not to be part of a mapping. A mapping specification has to consider two constraints to ensure the practicability of a step cooperation.
Regarding the multiplicities of a connection between parameters, a step output can be the source of none or an arbitrary number of mappings. E.g., the output \texttt{step1\_out1} is connected to two inputs of other steps. On the other hand, a step input can only be the target of at most one mapping. Otherwise, it is not clear, which value should be used at runtime or one value will overwrite the others.

Another constraint is that both values of the parameters that are part of one mapping have to be of the same type in terms of the formats. If this is not the case, the executor of the step whose input is the target of the mapping is not able to understand the received value at runtime. Nevertheless, if the formats of parameters mismatch, the \textit{Adapter} step (see section 6.3.1) as an intermediate can be used to adapt the formats.

A mapping between parameters of two steps is also a dependency between these two steps. The step, to which the target parameter of a mapping belongs, depends on the step that provides the corresponding source parameter. E.g., Step3 depends on Step2 because of the mapping from \texttt{step2\_out1} to \texttt{step3\_in1}. Thus, at runtime, the executor of Step2 has to be executed before the executor of Step3 such that the results of the Step2 executor, namely the value of \texttt{step2\_out1}, can be used by the Step3 executor (see figure 9 on page 42). Thus, dependencies can be used to find basic restrictions in ordering the steps. Furthermore, they show potentials for the parallel execution of step executors as long as the dependencies are fulfilled. E.g., regarding the strategy shown in figure 9, the executor of Step4 can be executed in parallel to the executors of Step2 and Step3. In addition to the dependencies among steps that originate from parameter mappings, it is conceivable that there are further dependencies which have not yet been covered. The reason is that a step may depend on another one though there is no parameter mapping between both of them. E.g., some constraints may require that the executor of Step4 has to be executed after the Step3 executor, which is a further restriction in ordering the steps. Thus, to meet the constraints, an explicit dependency from Step4 to Step3 might be added. This is not depicted in figure 9. A concrete example for the requirement of explicit dependencies is the constraint that an EJB module has to be deployed before it is able to be started. Thus, the step \textit{Module Deployment} has to be performed before the step \textit{Module Starting} because both steps address the same EJB module. Having no parameter mapping from \textit{Module Deployment} to \textit{Module Starting}, there is no dependency the other way around. Thus, a dependency has to be specified explicitly between these two steps to ensure the correct order of them at runtime. Covering such dependencies automatically is not trivial, because they are influenced by the concrete set of EJB modules and enterprise beans each step is addressing and by the concrete operations being performed on the sets through each step. The current state of the reconfiguration model imposes on an administrator to anticipate all relevant dependencies among steps before runtime. Thus, there might be the requirement that, besides parameter mappings, dependencies need to be specified explicitly among steps. Knowing all step dependencies before runtime enables the proof whether or not there are cycles among step dependencies of a strategy. A cycle would lead to a deadlock at runtime. Consequently, the problem of deadlocks can be avoided at runtime. The test for the validity of a strategy that is described later on includes the check for cycles among step dependencies.

Here, a further advantage of separating steps from step executors becomes apparent. When the executor should be replaced for a step, because a different implementation should be used, the executor associated to the step will be removed. This removal deletes all required and provided parameters from the step (compare to figure 8 on page 41) and
all corresponding parameter mappings these parameters are part of. Nevertheless, the
step itself and explicit dependencies to or from it remain within the strategy. Thus, the
initial design of the strategy is not broken or changed. Otherwise, if the step would also
be removed, the reconfiguration tasks specified by the removed step would get lost which
would change the intended overall behavior of the strategy. Finally, this does not prevent
the removal of step from a strategy if the strategy design should be changed intentionally.

To perform a step appropriately by running its executor, each required input parameter
has to be provided with a corresponding value. Thus, each non-optional input has to
be the target of a parameter mapping. Nevertheless, for some inputs no corresponding
outputs of other steps might be available. E.g., the step that is intended to be first one
to be performed cannot rely on other steps. Thus, certain input requirements have to
fulfilled from outside. Therefore, parameter inputs and outputs can be specified for a
strategy. The SampleStrategy shown in figure 9 on page 42 requires two inputs, namely
strategy_in1 and strategy_in2, and provides two outputs, namely strategy_out1 and
strategy_out2. Strategy inputs can be connected to inputs of steps to meet the step
requirements. E.g., the strategy input strategy_in1 is mapped to the input step1_in1
of Step1. Likewise, strategy outputs provide information about execution results of an
instantiated strategy. Therefore, outputs of steps can be connected to strategy outputs,
like, e.g., step4_out1 to strategy_out1. Thus, results of a reconfiguration are available
at the strategy level and need not to be collected from several steps. Strategy parameters
can also be declared as optional. Optional strategy inputs should have default values
that might be changed if necessary. Strategy outputs that are defined as optional do not
guarantee that after the execution of an instantiated strategy values for these outputs are
available. Regarding parameter mappings that involve strategy parameters, there are also
constraints that have to be considered.

- Each strategy parameter should be part of a parameter mapping. Otherwise, it is
  needless and need not to be specified. Thus, it could be removed. A strategy input
can be source of one or more mappings. On the other hand, a strategy output can
only be the target in at most one mapping. Otherwise, at runtime, the value that
is assigned to a strategy output will be overwritten by assignments of other values
originating from other mappings.

- Likewise to a parameter mapping between two steps, the values of two parameters
  being part of one mapping must be of the same type. Nevertheless, the need for
an Adapter step might be avoided, because a strategy parameter only consists of a
specification which may define the appropriate type that matches the requirements
of the other parameter of the mapping. Nevertheless, the use of an Adapter is not
forbidden. E.g., if a strategy input is connected to a step input and it should be
connected to another step input that has a different type than the strategy input,
an Adapter step might be introduced to enable the additional parameter mapping.

In summary, the specifications of strategy parameters, parameter mappings and of explicit
dependencies among steps enable the cooperation of steps, because the requirements of
each step, i.e., the required inputs, can be fulfilled. At runtime, these specifications are
used to orchestrate the executions of step executors.

This orchestration is implemented by so called Plan Executors. A Plan is an instance
of a fully specified strategy and it is executed by a plan executor to perform a concrete
reconfiguration. Plans and plan executors are described in detail in section 6.3.4. Here, it is important to know that different plan executors are conceivable. E.g., a plan executor may always perform all steps sequentially, while another plan executor is able to make use of the potentials for parallel execution of step executors. Another distinction might be whether a plan executor performs the reconfiguration synchronously or asynchronously to its client. As a strategy may influence the way an instance of it should be executed, a strategy should define its plan executor. Therefore, the plan executor that should be used for a reconfiguration has to be assigned to a strategy.

Finally, a concrete strategy consists of a set of steps together with their executors, specifications of input and output parameters at the strategy level, mapping specifications between parameters, declarations of explicit dependencies among steps, and an associated plan executor. All those parts are used to test the validity of a strategy by analyzing the specifications. The checked aspects are independent of a concrete execution of a strategy instance with concrete values. A strategy is valid if the following points are met.

1. At first, step dependencies being either declared explicitly or through parameter mappings are analyzed. As already discussed, circular dependencies among steps have to be avoided. Otherwise, the consequence will be a deadlock at runtime and the instance of the strategy cannot be executed. Therefore, considering the dependencies, all steps of a strategy must have together the ability to be linearized in a sequence of steps that is potentially executable. Nevertheless, a concrete sequence need not to be specified explicitly and there is no restriction that all steps have to be executed sequentially. This check avoids the problem of deadlocks at runtime when an instance of a strategy is executed.

2. Each step of the strategy has to be completely configured. This includes two aspects. First, an executor has to be assigned to each step. Second, all non-optional input parameters of a step and its executor must be the target of a parameter mapping. Thus, input values will be available at runtime and the requirements of the step executors are fulfilled.

3. The constraints, which have been discussed above, are also checked:
   (a) Each parameter mapping defined within the strategy connects two parameters whose values have to be of the same type in terms of the same format.
   (b) A step input parameter or a strategy output parameter can only be the target of at most one parameter mapping.
   (c) Each strategy parameter has to be part of at least one parameter mapping.

4. A plan executor must be associated to the strategy. This executor will be used for the execution of an instance of the strategy.

A valid strategy excludes some errors that potentially would have occurred during execution of an instance of it. Thus, a valid strategy is the basis for performing a concrete reconfiguration, which is described in the following section.
6.3.4 Reconfiguration Plan

For a concrete reconfiguration, an administrator has to provide a reconfiguration plan. Therefore, an appropriate strategy must be chosen, instantiated and configured. The choice of the strategy depends on the concrete reconfiguration need and on the concrete application that should be reconfigured. Nevertheless, the strategy must be a valid one. An instance of a concrete strategy is a plan. While a strategy is only a specification, a plan conforms to this specification and it may have concrete values, e.g., for instances of parameters being specified for a strategy. This is comparable to Java, where the concept of a strategy corresponds to the notion of a class, a concrete strategy to a concrete class, and finally, a plan to an object instance of the concrete class. Before a plan can be executed by a Plan Executor, it has to be configured. Plan configuration is limited to the provision of values for input parameters that are specified by the corresponding strategy of which this plan is an instance. Thus, an administrator only needs to know what a strategy is doing, but not how it is realized, i.e., the internals of a strategy are not relevant. After the configuration of a plan and before its execution, the plan should be validated. A plan is valid if the following points are met.

1. The plan executor that is designated to be used for the execution of the plan must be runnable. Since this approach focuses on the EJB technology, executors run within an EJB container. Therefore, an executor is runnable if the EJB module the executor is part of has been deployed and started in the container.

2. Likewise to the plan executor, each step executor that is associated to a step of the plan must be runnable. Step executors run also in the EJB container. Therefore, the EJB modules the executors are part of have to deployed and started in the container.

3. For each input parameter specified by the strategy from which the plan has been derived concrete values have to be provided. If an input is optional, an administrator need not to provide a value for it, but then the strategy should define a default value that will be used.

In contrast to the strategy validation, which is described in section 6.3.3, the validity check of a plan addresses aspects that should be checked just before the plan is executed. Only a valid plan should be executed. For execution, a plan is sent to its plan executor that is responsible for the orchestration and execution of the steps and their step executors being part of the plan. Therefore, considering the dependencies among steps, a plan executor invokes the corresponding step executors and provides them with values for their required inputs. Values for required inputs of step executors can be obtained from two sources. They are either results of step executors that have already been executed, or plan input values that conform to the specifications of corresponding strategy inputs and that are provided with the configuration of the plan. Thus, a plan executor propagates parameter values according to parameter mapping specifications which are part of the strategy from which the plan has been derived. This enables that the reconfiguration can be executed automatically and without any interaction need. Finally, after all step executors have been executed, a plan executor returns values for plan outputs according to the specifications of the corresponding strategy output parameters.
6.4 Roles in the Reconfiguration Development and Application

The development and application of a reconfiguration involve several roles that are called reconfiguration roles. These roles enhance and do not replace any of the roles that are defined by the EJB specification and that are described in section 4.2.2. It is conceivable, that a single party that performs an EJB role, also performs a reconfiguration role because both roles may require, among others, the same knowledge or skills from a party. Therefore, based on the reconfiguration model presented in the sections 6.2 and 6.3, four roles are defined. These roles are the Strategy Creator, Reconfiguration Planner, Reconfiguration Operator, and the Executor Provider. The reconfiguration model is the basis for the contracts that ensure that the product of one role is compatible with the product of other roles. A product is an entity of the reconfiguration model and it is either a strategy, a reconfiguration plan, or an executor.

6.4.1 Strategy Creator

The Strategy Creator combines several steps, which are presented in section 6.3.1, into a reconfiguration procedure that realizes a certain strategy, which is discussed in section 6.3.3. The output of his or her work is a concrete and valid strategy.

The work of a strategy creator can be divided into two phases, namely the design and the specification of a strategy. During the design phase a strategy creator is concerned with the set of provided steps, which is described in section 6.3.1, and how these steps can be combined to a strategy that fulfills actual or potential reconfiguration needs. Therefore, a strategy creator considers a strategy at an abstract level, because he or she is not concerned with the realizing step or plan executors. Appropriate executors are either already available or they can be provided on demand by an Executor Provider. The role of an Executor Provider is specified in section 6.4.4. Thus, executors should not influence the design of a strategy. The strategy creator must be able to understand reconfiguration needs and concepts of reconfiguration, like, e.g., quiescence. Moreover, he or she must be familiar with the provided set of steps and must know which kind of step combination meets the reconfiguration needs while considering other objectives, like, e.g., minimizing the system disruption.

After designing a strategy, executors play a role in the specification phase to obtain a concrete strategy. Therefore, a step executor obtained from an Executor Provider has to be assigned to each step of the strategy. A strategy creator may choose executors that are already available or he or she may demand appropriate executors from an Executor Provider who is going to implement them. Nevertheless, the strategy provider is not concerned with the implementations of the step executors, because only the contractual information about an executor, i.e., what an executor is doing and the specification of its required and provided input and output parameters, are relevant. These information are used by the strategy creator to specify strategy parameters, parameter mappings and additional dependencies among steps. Moreover, a plan executor, also obtained from the Executor Provider, has to be associated to the strategy. Comparable to step executors, the strategy creator need not to have any knowledge about the concrete implementations of plan executors. Finally, the created strategy has to be validated to ensure that the output of the strategy creator is a valid strategy.

A strategy can be created in advance to a concrete reconfiguration need, because several reconfiguration needs may have the same requirements on the application of a reconfigu-
ration. Thus, a strategy that fulfills these requirements can be seen as a reusable pattern, that can be applied in several reconfiguration situations. Such strategies correspond to the provided strategies as described in section 6.2. Nevertheless, if no appropriate strategy is available to meet a concrete reconfiguration need, custom strategies in addition to the provided ones can be created. Finally, if a more dynamic approach is necessary, it is conceivable that the strategy is created gradually at runtime, e.g., in interaction with an administrator, which results in an ad-hoc strategy. However, in-depth knowledge about the business domain of the application whose reconfiguration will be based on a certain strategy is not required for a strategy creator.

6.4.2 Reconfiguration Planner

The Reconfiguration Planner prepares the application of a concrete reconfiguration. The need for a reconfiguration of an EJB-based system originates from a problem that may be identified by the party performing the EJB role of the System Administrator, who is responsible for the runtime monitoring of applications, or by clients of the system who are complaining about errors or poor performance. Afterwards, the problem has to be analyzed and a solution that solves the problem has to be found and implemented. This may involve several parties, like, e.g., the Enterprise Bean Provider or the Application Assembler because the cause for the problem may be the implementation of enterprise beans or the assembly of the application, respectively. After both roles provide a solution, like, e.g., a new EJB module that should replace the faulty one, the work of the reconfiguration planner starts.

The reconfiguration planner has to find a way to apply the solution to the system by reconfiguring it. Therefore, depending on the concrete reconfiguration need and on the application that should be reconfigured, he or she must find an appropriate strategy. Consequently, this might require knowledge about the reconfigured application from the reconfiguration planner. An appropriate strategy is either already available or the Strategy Creator is requested to provide one. Having found a strategy, the planner must instantiate the strategy to a reconfiguration plan and he or she must configure the plan. Plan configuration is the provision of values for input parameters, which are specified by the chosen strategy, to the plan. Finally, the configured plan should be validated. Thus, the output of the reconfiguration planner is a configured and a valid plan.

Plan validity requires that the reconfiguration planner has to ensure that the chosen plan and step executors of the plan are runnable in the EJB container. Thus, either the modules that contain the relevant executors have already been deployed and started, or the reconfiguration planner has to deploy and start them. The EJB role of the Deployer might assist the reconfiguration planner with the deployment and starting of modules.

6.4.3 Reconfiguration Operator

The Reconfiguration Operator takes the output of the Reconfiguration Planner and executes the valid plan. Moreover, the execution of the plan should be monitored in order to recognize and handle potential problems. Additionally, a plan execution might require support from the operator. The ideal case is that the reconfiguration could be executed autonomically and without any errors. However, it is conceivable, that interaction with the operator is required, e.g., to make a decision or to provide information that could
not be retrieved by the running executors. Therefore, it might be required that the operator is familiar with the reconfiguration strategy that is applied, with the reconfigured application and with the application domain.

6.4.4 Executor Provider

The *Executor Provider* is the developer of executors. An executor can be either a plan or a step executor.

To implement a plan executor, the executor provider should be familiar with the concepts of the reconfiguration model, which is discussed in detail in section 6.3, because he or she must implement facilities that use the information contained in a reconfiguration plan and that enable the cooperation of step executors at runtime. Thus, a plan executor realizes the orchestration of step executions. Usually, a plan executor does not depend on a concrete reconfiguration or on a concrete application that should be reconfigured. Therefore, a provider of a plan executor need not to have such knowledge and a plan executor is completely reusable in several reconfiguration situations.

In contrast, an implementation of a step through a step executor may depend on a concrete application. E.g., an executor for the step *Conversational State Conversion* involves application-specific code that works with objects from the application domain. Therefore, a provider of certain step executors must be either an application domain expert who is familiar with the business logic of the application that should be reconfigured, or he or she is assisted by such an expert. Consequently, the provider of such executors can be the same party as the party performing the EJB role of the *Enterprise Bean Provider*. On the other hand, some step executors are independent of a concrete application. E.g., the executors for managing the life cycle of EJB modules are completely generic and include no application-specific code. Thus, knowledge about the application domain is not required.

Besides writing the code for a step executor, the executor provider must provide a description for it that contains specifics about the implementation, like, e.g., preconditions that must be fulfilled in order to use the executor appropriately. This description is intended to be read by a *Strategy Creator* who must select appropriate step executors for the steps of the strategy he or she is creating. Moreover, the executor provider must specify input and output parameters the executor requires or provides, respectively. This specification is also used by the *Strategy Creator* for the specification of parameter mappings. The role of the *Strategy Creator* is described in section 6.4.1.

Finally, every executor provider should be familiar with the EJB component model, because his or her code directly affects EJB-based applications. Furthermore, executors might use the *mKernel* API extensively. Thus, an executor provider must be familiar with the API of the *mKernel* system, while internals of *mKernel* are transparent to him or her.
7 Implementation of the Reconfiguration Model

This section provides information about the implementation of the reconfiguration model which is presented in section 6. The implementation consists of an Application Programming Interface (API) and a reconfiguration infrastructure that provides the API. The following section 7.1 presents the API and how the different roles of the reconfiguration development and application use the API to perform their tasks. These roles are described in section 6.4. Afterwards, in section 7.2, some selected and relevant internals of the infrastructure are discussed.

7.1 API Provided by the Implementation

The reconfiguration infrastructure provides an API that is intended for the Strategy Creator, Reconfiguration Planner, Reconfiguration Operator, and for the Executor Provider. These roles are described in section 6.4. All four roles use the API to perform their tasks, and additionally, an Executor Provider has to provide code, i.e., implementations for executors, to the infrastructure. The API consists of the mKernel.reconfiguration package and its sub-packages. At first, the API used by the Strategy Creator, Reconfiguration Planner, and Reconfiguration Operator is presented. Afterwards, the API that is intended for the Executor Provider together with aspects he or she has to consider for implementing executors are discussed.

7.1.1 Strategy Creator API

The task of a Strategy Creator is to design and specify strategies, which is discussed in section 6.4.1. He or she uses parts of the package mKernel.reconfiguration provided with the API to perform this task. Within this package, the Reconfiguration interface is the point of entry for a Strategy Creator in order to create a new or retrieve an existing strategy. A new one has to be designed and specified, while the design and specification of an existing one might be modified. Therefore, the API depicted in figure 10 on page 51 is utilized. This API provides a view that is equivalent to the concept that describes the combination of steps into strategy realizing procedures and that is presented in section 6.3.3.

A Strategy has a name and a description and it consists of at least one Step. A Step represents one of the provided steps described in section 6.3.1 and the stepType identifies the represented step. A Step belongs to exactly one Strategy and it has a unique name within this Strategy. Steps can be added to and removed from a Strategy. A step executor is assigned to each Step and a plan executor to the Strategy. In order to find appropriate executors, a Strategy Creator uses the API representation of plan and step executors that are provided by the Executor Provider. This API representation is described in section 7.1.4 and depicted in figure 12 on page 59. Consequently, there exist associations from a Strategy to a PlanExecutor and from a Step to a StepExecutor. These associations are not depicted in the figures.

A ParamOwner is either a Step or a Strategy that may have Parameters. A Parameter is either an InputParameter or an OutputParameter and it belongs either to a Step or to a Strategy. The interfaces InputParameter, OutputParameter and Parameter
have a type parameter Owner to specify whether the owner of a parameter instance is a Step or a Strategy. A Strategy Creator does not specify Parameters for Steps because Parameters of Steps represent the parameters (see ExecutorParameter in figure 12 on page 59) of their associated StepExecutors (compare to figure 8 on page 41). In contrast, Parameters can be specified for a Strategy. A Parameter has a globally unique identifier (uniqueIdentifier), that is generated internally and automatically, and a key which is its local identifier within a Step or a Strategy. The semantics of a Parameter are specified through a description and a valueType defines the type to which concrete values of the Parameter must conform. Finally, a Parameter can be an optional or a mandatory one, which is specified by the optional property.

Parameters are used to specify parameter mappings. A parameter mapping is represented by an IORelation which can be added to and removed from a Strategy. An IORelation connects a source Parameter to a target Parameter and it has the two type

---

**Figure 10:** Simplified Strategy Creator API
parameters Source and Target. Source and Target refer to the concrete ParamOwner of the source Parameter or of the target Parameter, respectively. An IORelation between two Parameters that both belong to Steps constitute a dependency between the corresponding Steps. Additionally, explicit dependencies to Steps can be added to and removed from a Step (addDependency() and removeDependency()). All dependencies of a Step, either originating from parameter mappings or from explicit dependencies, can be obtained with the getDependencies() method. Nevertheless, a Strategy Creator need not to synchronize the parameter mappings between step parameters and the corresponding dependencies between steps. E.g., if an IORelation between two Parameters of different Steps is removed from a Strategy, the corresponding dependency between the Steps is removed, too. Even, the removal of a Step from a Strategy causes the removal of all IORelations that involve a Parameter of the removed Step. Moreover, the API assists a Strategy Creator in specifying a valid Strategy. E.g., an IORelation can only be specified for two Parameters that have the same valueType. The characteristics of a valid strategy are discussed in section 6.3.3. A Strategy Creator may even validate a strategy on demand using the validate() operation of the Strategy. Finally, a Strategy can be persisted to and deleted from a persistent storage (save() and remove()). Persisting a Strategy is relevant for its reusability in order to make it available at a later point in time, e.g., for a Reconfiguration Planner who searches appropriate strategies for a concrete reconfiguration need.

7.1.2 Reconfiguration Planner API

The task of the Reconfiguration Planner, which is described in detail in section 6.4.2, is to provide a valid reconfiguration plan for a concrete reconfiguration need, i.e., a strategy must be chosen, instantiated and configured. Thus, he or she must find an appropriate reconfiguration strategy that has been provided by the Strategy Creator. Persisted strategies can be obtained through the Reconfiguration interface. Then, the inspection facilities of the API presented in section 7.1.1 and shown in figure 10 on page 51 is used to find an appropriate strategy. Having found one, a plan can be generated from the chosen Strategy using its generatePlan() method. The strategy is validated internally in order to prevent the creation of a plan if the corresponding strategy is not valid. Otherwise, the strategy is instantiated to a plan which results in the usage of the API depicted in figure 11 on page 53. Consequently, there exist an association from the Strategy to the PlanManagement interface which is not depicted in the figures. From a conceptional point of view, Strategies, Steps, Parameters and IORelations are specifications and Plans, PlanSteps and PlanParameters are instances of these specifications. Instances of IORelations between PlanParameters are integrated into Plans (getRelations()) and not represented as first class entities. Since these instances have to conform to the specifications, it is not possible to modify them, like, e.g., removing a PlanStep from a Plan. Thus, these interfaces contain mostly operations for inspection, i.e., getter methods.

A PlanManagement contains the generated Plan if the strategy from which the plan has been derived is valid. Otherwise, there is no Plan available. Nevertheless, information about the strategy validation and plan generation can be obtained with the getReport() method. A Plan, a PlanStep and a PlanParameter contain only the information of a Strategy, Step and Parameter, respectively, which are relevant for the execution of a reconfiguration (compare the interfaces and their operations of the API depicted in figure 10 on page 51 and in figure 11 on page 53). Thus, information about the steps of a
Implementation of the Reconfiguration Model

Figure 11: Simplified Reconfiguration Planner API

plan, dependencies among these steps, parameters of the plan and of the steps, and about mappings between these parameters are available. Additionally, mapped names of plan and step executors are part of a plan. Executors are realized by an Executor Provider, which is described in section 7.1.4. Within the reconfiguration infrastructure, executors become session beans, which is the reason for having mapped names assigned to step and plan executors. Thus, to identify certain executors that should be used for the execution of a reconfiguration, it is sufficient to know the mapped names under which they are bound in the naming directory of the EJB container.

To configure a Plan, a Reconfiguration Planner has to provide values for PlanParameters that are inputs of the Plan and that are obtained with the getInputs() method of the corresponding Plan. These parameters are instances of strategy input parameter specifications of the corresponding strategy. Thus, in contrast to a Strategy, a Plan considers a concrete reconfiguration, such that the Parameters of a Strategy do not have concrete values, while PlanParameters of a Plan may have concrete values (see setValue() and getValue() methods). Additionally to the plan configuration, a Reconfiguration Planner has to ensure that the executors which will be used during plan execution are available and accessible in the EJB container. This can be checked and reached with the API that is described in section 7.1.4. Finally, a Plan might be validated to guarantee that the
Reconfiguration Planner hands over a valid plan to the Reconfiguration Operator, because only valid plans should be executed. Validity of a plan is discussed in section 6.3.4.

7.1.3 Reconfiguration Operator API

The Reconfiguration Operator is responsible for the execution of a valid plan. Therefore, he or she uses the same API as the Reconfiguration Planner, which is shown in figure 11 on page 53. The PlanManagement interface provides the method runPlan to start the execution of a Plan. Before a Plan is submitted to its associated plan executor, i.e., to the plan executor with the mapped name Plan.getPlanExecutorMappedName(), it will be validated internally. An execution of an invalid plan, which may result in failures, is prevented. Otherwise, a Plan is executed by a plan executor that returns a PlanResult as the result of the execution. This result contains a report about the plan execution (getPlanReport()) and the executed plan that has been sent previously to the plan executor (getExecutedPlan()). Nevertheless, after execution, the output PlanParameters of the Plan and of its PlanSteps contain values that actually represent the results of the reconfiguration. The current API does not support the task of the Reconfiguration Operator to monitor the execution of a reconfiguration. Nevertheless, it is conceivable that, e.g., plan executors throw events through a Java Message Service (JMS) [jms] topic to provide information about the current status of the reconfiguration. The Reconfiguration Operator might consume these events, and furthermore, he or she might directly use the mKernel API to inspect the behavior and structure of the reconfigured application.

7.1.4 Executor Provider API

An Executor Provider is the developer of step and plan executors. The role of the Executor Provider is discussed in section 6.4.4. This section describes aspects, a provider has to consider for the implementation of executors and for the integration of executors into the reconfiguration infrastructure. Thus, the infrastructure can be seen as a framework, because it might be extended with executors, i.e., with additional code which is provided by any party that performs the role of the Executor Provider. To be integrable into the infrastructure, an executor implementation must conform to the API that consists of the package mKernel.reconfiguration.executor and its sub-package. At first, the API for step executor implementations is presented, and afterwards the API for plan executor implementations. Finally, common restrictions for step and plan executor implementations are discussed, and the integration of executors into the infrastructure is presented.

Step Executor API  A step executor realizes the tasks that are specified by one of the steps presented in section 6.3.1. Thus, an implementation of a step executor includes the code that performs the tasks of the realized step. Additionally, required input and provided output parameters might be specified, and meta information about the executor, like, e.g., a description or which step it is realizing, must be provided. This corresponds to the conceptual description of step executors in section 6.3.2. The API that must be considered by step executor implementations consists of an interface and of metadata annotations. Listing 1 on page 55 shows the stub of a class for an example step executor that realizes the step Module Deployment. This example is used to describe the API.
Listing 1: An Example for a Step Executor

```java
package sample.executors;

import mKernel.ejb.Container;
import mKernel.ejb.DeploymentState;
import mKernel.reconfiguration.StepType;
import mKernel.reconfiguration.executor.IStepExecutor;
import mKernel.reconfiguration.executor.annotation.Input;
import mKernel.reconfiguration.executor.annotation.Output;
import mKernel.reconfiguration.executor.annotation.StepExecutor;

@StepExecutor(
    name = "ModuleDeployer",
    description = "Deploys an EJB module into the container.",
    stepType = StepType.MODULE_DEPLOYMENT)
public class DefaultModuleDeployer implements IStepExecutor {
    @Input(description = "Identifier of the EJB module at the " + "Deployment Level that should be deployed.")
    private String ejbModuleId;

    @Input(description = "Enables the logging", optional=true)
    private boolean verbose = false;

    @Output(description = "The deployment state of the " + "corresponding EJB module after trying to deploy it",
            optional=false)
    private DeploymentState ejbModuleState;

    @Override // the only method from the IStepExecutor interface
    public void runStep(Container container) {
        // code for deploying the EJB module
    }
}
```

Meta information is provided with the @StepExecutor annotation that is applied to the step executor class and that has three attributes (see lines 11 to 14 of listing 1).

- Optionally, a **name** can be defined for the executor. If no name is specified, then the default name is the unqualified name of the step executor class.

- A mandatory **description** of the step executor that may include, among others, information about its usage, like, e.g., pre- and postconditions or assumptions that are made by the step executor implementation.

- The mandatory **stepType** attribute refers to the step the executor is realizing. Therefore, the enumeration **StepType** is used that represents the provided set of steps which is described in section 6.3.1.

Input and output parameters of a step executor are specified with the @Input and @Output annotation, respectively. Both annotations are applied to fields of the executor class, but a field can be either an input or an output. Two attributes can be optionally specified for both annotations.
The semantics of a parameter are specified with the `description` attribute. This attribute is optional and its default value is the empty string. Though a description is not required, it is recommended to provide one.

A parameter can be either optional or mandatory. This can be specified with the boolean attribute `optional`. This attribute is not required to be provided by the developer and its default value is `false`, i.e., the parameter is, per default, a mandatory one. Thus, e.g., the first input parameter of the step executor class shown in listing 1 is non-optional and a value has to be provided for it at runtime.

For optional input parameters, default values should be defined by the `Executor Provider`, i.e., the corresponding field should not only be declared but also be initialized with a value. Consequently, there is no need to provide a value for an optional input parameter at runtime, except for the case, that a value different from the default one is desired. E.g., the second input parameter of the example step executor is an optional one and its default value is `false`. Thus, the logging of the step executor is disabled if no input value, precisely the value `true`, is provided at runtime (see lines 21f. in Listing 1). Additionally, the type of a parameter is known through the Java type of the field. This type must be a primitive data type, i.e., a `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, or a `char`, or of any complex object type that is serializable. If the type is a complex object, this includes recursively that all of its referenced objects are also serializable or the references are transient. Furthermore, the name of the field that is unique within an executor class serves as a local identifier for the parameter within the step executor class.

Finally, the executor class must implement the interface `IStepExecutor` that specifies only one generic method, called `runStep()`. The method has one parameter which is the point of entry to the `mKernel` API. During execution, before this method will be invoked, values for the specified input parameters are injected into the corresponding fields by the infrastructure. Thus, a required input is seen as a dependency and the concept of `Dependency Injection` is applied. Therefore, an `Executor Provider` can rely on the availability of input values within the `runStep()` method. Within this method, the tasks of the realized step have to be implemented. For the example step executor of listing 1 the task is the deployment of an EJB module. Therefore, input parameter values might be processed and the `mKernel` API might be used. If non-optional output parameters have been specified, values for them have to be assigned to the corresponding fields by this method. Likewise, for optional outputs, if there exist any, values might be assigned to the corresponding fields. Consequently, after the `runStep()` method has finished, output values are stored in the corresponding fields and the infrastructure makes these values available to other step executors. Thus, an `Executor Provider` can concentrate completely on the implementation of the tasks of the realized step and he or she is not concerned with the retrieval of required input values or the provision of output values to other step executors. Consequently, the work of an `Executor Provider` is simplified.

**Plan Executor API** Similar to step executors, implementations of plan executors have to consider interfaces and metadata annotations of the API. Nevertheless, a plan executor does not specify input or output parameters, but a plan executor class implements a certain interface and provides meta information about itself through metadata annotations. Listing 2 on page 57 shows the stub of a class for an example plan executor.
Listing 2: An Example for a Plan Executor

```java
class DefaultPlanExecutor implements IPlanExecutor {
    @Override
    public PlanResult executePlan(Plan plan) {
        // code for plan execution
    }
}
```

Metadata annotations are used to specify meta information about a plan executor. Therefore, the `@PlanExecutor` annotation is applied to the plan executor class and it consists of two attributes (see lines 8 to 10 in listing 2).

- Optionally, a `name` can be defined for the plan executor. If no name is specified, then the default name is the unqualified name of the plan executor class.

- A mandatory `description` of the plan executor that includes information about specifics of the plan execution, e.g., any assumptions that are made by the implementation or if step executors are executed sequentially or, if possible, in parallel.

Additionally, a plan executor class has to implement the interface `IPlanExecutor` that specifies a generic method, called `executePlan()`, for plan execution, i.e., for running the reconfiguration. The return value `PlanResult` and the argument `Plan` of this method are depicted on figure 11 on page 53. The `Plan` is created and configured by the `Reconfiguration Planner` and sent to the plan executor by the `Reconfiguration Operator` (see sections 7.1.2 and 7.1.3, respectively). The `Plan` contains all relevant information for performing a reconfiguration: all steps (`PlanSteps`) and their step executors together with required input and provided output parameters of the executors, which are represented as `PlanParameters`, dependencies among steps, input and output parameters of the plan, which are represented as `PlanParameters`, and finally, mappings among `PlanParameters`. Additionally, input parameters of the `Plan` contain concrete values which have been assigned by the `Reconfiguration Planner`. Thus, the task of a plan executor is to orchestrate the execution of step executors being part of the plan while taking step dependencies and parameter mappings into consideration. An invocation of a step executor includes the provision of values for the required input parameters of the invoked step executor. As a result, the invoked step executor returns values for its provided output parameters. To invoke a step executor, the plan executor uses the interface `IPlanStepExecutor` that is depicted in listing 3 on page 58. This interface specifies one generic method, called `executePlanStep()`. The second argument of this method is the point of entry to the `mKernel` API that might be used by a step executor. The first argument, called `input map`, of this method are values for input parameters that are required by the step executor for
an appropriate execution, and the result, called *output map*, of this method are values for output parameters provided by the step executor. An entry of the *input map* corresponds to an input and an entry of the *output map* to an output *PlanParameter* of a *PlanStep*. Thus, keys of both maps are identifiers for the corresponding *PlanParameter* which can be obtained from the method *PlanParameter.getKeyName()*.

Values of both maps are concrete parameter values that correspond to *values* of the related *PlanParameters*. These values are byte arrays to avoid the need that all classes that are used as step executor parameters have to be part of the class path of the plan executor.

Listing 3: IPlanStepExecutor interface

```java
package mKernel.reconfiguration.executor;
import java.util.Map;
import mKernel.ejb.Container;

public interface IPlanStepExecutor {
    Map<String, byte[]> executePlanStep(Map<String, byte[]> input, Container mKernelContainer);
}
```

In order to invoke a step executor of a *PlanStep*, all relevant input *PlanParameters* are obtained through *getStepInputs()* of the corresponding *PlanStep*. For each of these *PlanParameters*, an entry is added to the *input map*, such that the requirements of the step executor are fulfilled. The *executePlanStep()* method can be invoked. After the invocation, the resulting parameter values that are contained in the *output map* can be set to the corresponding output *PlanParameters* that are obtained with the method *getStepOutputs()* of the *PlanStep* whose step executor has been invoked. To exchange information among step executors through forwarding parameter values, mappings among *PlanParameters* are used. The value of the source *PlanParameter* of a mapping can be obtained with *getValue()* and this value can be set to the corresponding target *PlanParameter* with *setValue()*.

Thus, a *PlanParameter* is a kind of container that maintains the relation between meta information about a parameter and a concrete value of the corresponding parameter throughout the reconfiguration in order to enable parameter mappings and an appropriate usage of the *input- and output map*. The result of a plan execution, i.e., a *PlanResult*, is created from the submitted *Plan* by a plan executor. A *PlanResult* contains the executed *Plan* whose output *PlanParameters*, obtained through *getOutputs()*, contain concrete *values* that represent the results of the reconfiguration. Additionally, a plan executor may generate a report, as a list of messages, about the plan execution, which might be part of the *PlanResult*.

Finally, the infrastructure guarantees that an invocation of *executePlanStep()* on a certain step executor results in an invocation of the corresponding *IStepExecutor.runStep()* method, while managing the mappings between the contents of the *input- and output map* to the corresponding input and output fields of the step executor class (compare to the step executor API described above).

**Common Implementation Restrictions** Since executors are running in the EJB container, an *Executor Provider* has to follow the guidelines of the EJB standard, like, e.g., the programming restrictions (see page 545f. in [DK06a]), when implementing an executor. E.g., it is not allowed to perform any kind of thread handling or to access the file
system of the machine that hosts the container. Thus, an *Executor Provider* must follow the same guidelines as the *Bean Provider*. Indeed, within the current implementation of the reconfiguration infrastructure, executors become session beans. Nevertheless, this is transparent to an *Executor Provider*, such that he or she need not to specify or implement any EJB-related aspects, like, e.g., an EJB *Deployment Descriptor* or EJB metadata annotations.

**Integration of Executors** Having implemented executors, an *Executor Provider* packages them together with their related classes into a *Java Archive* (JAR), called *Executor Module*. An executor module must contain at least one executor, either a step or a plan executor. An *Executor Provider* may integrate a module into the reconfiguration infrastructure through the `mKernel.reconfiguration.ExecutorModuleUpload` interface provided by the API. During the integration of a module, the metadata annotations and the executor classes are processed in order to obtain a representation of the module and its executors. This representation can be accessed through the API and it is shown in figure 12.

![Figure 12: Simplified API Representation of an Executor Module and its executors](image-url)
This representation can only be inspected, but not manipulated, because it is defined by
the source code, including metadata annotations, of the executors and by the packag-
ing of executor classes into a module. Thus, modifications would require changes of the
source code or of the executor module assembly. Consequently, only the life cycle of an
executor module can be manipulated with the ExecutorModule interface (see figure 12
on page 59). The life cycle is comparable to the life cycle of a standard EJB module (see
JSR-88 [Doc06]), and in addition, the remove() method deletes the executor module and
its representation from the reconfiguration infrastructure. The current state of the life
cycle is represented by the enumeration ExecutorModuleState and it might be obtained
through the getDeploymentState() method.
To complete the explanation of figure 12 on page 59, unique identifiers for each entity
of the representation are generated internally and automatically. The same holds for
mapped names of Executors. Each executor becomes a session bean which requires
mapped names for the bindings of executors in the naming directory of an EJB container.
The className of an Executor is the fully qualified name of the corresponding executor
class. The name and the description of an executor are defined through the metadata an-
tnotations @StepExecutor or @PlanExecutor. With the @StepExecutor annotation, the
stepType of a StepExecutor is specified. The properties of an ExecutorParameter are
obtained from the corresponding step executor class field that is annotated with @Input
or @Output. The key and the valueType of an ExecutorParameter are the name and the
fully qualified Java type of the corresponding field, while the other properties are speci-
fied through attributes of the @Input or @Output annotations, that have been described
above.
The API representation of executors is primarily used by the Strategy Creator, being
described in section 7.1.1, for the specification of a strategy, when an appropriate step
executor must be assigned to each step of the strategy and an appropriate plan executor
must be assigned to the strategy.

7.2 Internals of the Implementation

This section addresses internals of the reconfiguration infrastructure that provides the
API being presented in section 7.1. Parts of the infrastructure, as a set of enterprise
beans, run in an EJB container, while other parts have to run outside the container
because they perform, among others, bytecode modifications. Since the functionality of
the infrastructure has already been presented with the API in section 7.1, this section
discusses three selected internal aspects that are hidden by the API.

At first, in section 7.2.1, the integration process of executors into the reconfiguration
infrastructure is presented. Executors are implemented by Executor Providers. The inte-
gration of executors is required, such that these executors might be used during a reconfig-
uration. The second and third aspects consider the problem of transferring Conversational
States among stateful session bean instances. Therefore, section 7.2.2 shows how the life
cycle of stateful session bean instances can be utilized to ensure that a conversational state
is able to be transferred. This life cycle is defined by the EJB standard [DK06a] that
requires that a conversational state of a bean instance must be serializable when the in-
stance is in a certain stage of its life cycle. Section 7.2.3 addresses the concrete question of
transferring instances of entity managers, entity manager factories and entities as part of
a conversational state, which requires special treatments during the state transfer process.
7.2.1 Integration of Executor Modules into the Infrastructure

The reconfiguration infrastructure can be compared to a framework, because Executor Providers develop step and plan executors that might be integrated into the infrastructure. Thus, the infrastructure might be enhanced with additional code. The API for the development and integration of executors is presented in section 7.1.4. Executors are packaged by an Executor Provider into a Java Archive (JAR), called Executor Module. Such a module must contain at least one executor and it is the unit for integrating executors into the infrastructure through the interface mKernel.reconfiguration.ExecutorModuleUpload. This section describes the internal integration process of an executor module into the infrastructure. In order to use the executors, being part of an executor module, during a concrete reconfiguration, a preprocessing of executors is required, which is performed during the integration process.

At first, the module is scanned for executors. How executors have to be implemented is described in section 7.1.4. A step executor must be annotated with @StepExecutor and it must implement the mKernel.reconfiguration.executor.IStepExecutor interface. A plan executor must implement the interface IPlanExecutor of the same package, and it must be annotated with @PlanExecutor. Thus, at least one step executor or plan executor that fulfills these conditions must be part of the executor module. Otherwise, the module contains no executor and the integration of the module into the infrastructure is aborted. If at least one executor has been found, the bytecode of each executor class and their metadata annotations are analyzed to obtain a representation of the executor module and of the executors being part of the module. This representation can be accessed via the API as shown in figure 12 on page 59. The analysis of executor classes is performed with the Pluggable Annotation Processing API [Dar06] that is part of the Java Standard Edition 6. Two annotation processors have been realized, one for plan and one for step executors. The analysis is started by running internally the Java Compiler that scans the bytecode of the uploaded module for annotations and invokes the corresponding processor if an @StepExecutor or an @PlanExecutor annotation has been found. Therefore, each processor implements a callback method that analyzes the annotations and the bytecode of executor classes with the help of the javax.lang.model.* packages that are part of the Pluggable Annotation Processing API and that provide a mirror-based model of the Java programming language. After the processors have executed, an XML-based representation, called Executor Descriptor (ED), of the uploaded executor module has been generated using the Java Architecture for XML Binding (JAXB) [Kaw06]. E.g., listing 4 on page 62 shows such a representation for an example executor module, called sampleExecModule.jar, that contains the step executor being depicted in listing 1 on page 55 and the plan executor being depicted in listing 2 on page 57.

Comparing the source code of the executors and the Executor Descriptor (ED), all information being part of the ED are obtained from the bytecode of the executors or from the metadata annotations. Only the mapped-names of executors are generated automatically by processors and the module-name is the simple name of the uploaded JAR file. An ED follows a certain XML Schema Definition (XSD), such that it is conceivable for a future version of the current implementation, that an Executor Provider might specify executors not only through metadata annotations but also through an ED. This is comparable to an EJB Deployment Descriptor and the related metadata annotations, e.g., to specify enterprise beans.
Listing 4: XML-based Executor Descriptor for an Example Executor Module

```xml
<reconf version="1.0" encoding="UTF-8"/>
<reconf-exec xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.lspi.wiai.uni-bamberg.de/reconfexec-jar-1-2"
xsi:schemaLocation="http://www.lspi.wiai.uni-bamberg.de/
reconfexec-jar-1-2
file:///home/vogel/reconfexec-jar_1_2.xsd">
<module-name="sampleExecModule.jar">
<executor-name>ModuleDeployer</executor-name>
<executor-class>sample.executors.DefaultModuleDeployer</executor-class>
<mapped-name>mKernel/reconfiguration/stepExecutor/
 f0f3c923d9e149838ba97d27da25b</mapped-name>
<description>Deploys an EJB module into the container.</description>
<step-type>MODULE DEPLOYMENT</step-type>
<input>
<description>Identifier of the EJB module at the Deployment Level that should be deployed.</description>
<field-name>ejbModuleId</field-name>
<field-type>java.lang.String</field-type>
<optional>false</optional>
</input>
<input>
<description>Enables the logging</description>
<field-name>verbose</field-name>
<field-type>boolean</field-type>
<optional>true</optional>
</input>
<output>
<description>The deployment state of the corresponding EJB module after trying to deploy it.</description>
<field-name>ejbModuleState</field-name>
<field-type>mKernel.ejb.DeploymentState</field-type>
<optional>false</optional>
</output>
</step-executor>
</plan-executor>
<executor-name>PlanExecutorSeq</executor-name>
<executor-class>sample.executors.DefaultPlanExecutor</executor-class>
<mapped-name>mKernel/reconfiguration/planExecutor/
 aclf0b36daf4f77b5432ee1fcd7408</mapped-name>
<description>Plan executor that executes all steps sequentially.</description>
</plan-executor>
</reconf-exec>
```

The information contained in an ED are represented as Java objects during the preprocessing of an uploaded executor module and these objects are used as an information base for the further steps in the preprocessing.

The next step in the preprocessing is the bytecode manipulation of step executor classes, while plan executors classes are not modified. As described in section 7.1.4 a step executor must only implement the interface IStepExecutor that defines the method runStep, but
a plan executor that invokes a step executor during a reconfiguration uses the method executePlanStep of the interface IPlanStepExecutor. Thus, using Javassist [Chi08] the bytecode of a step executor is modified in order to add the method executePlanStep and the interface IPlanStepExecutor to the executor class. Listing 5 shows the simplified source code of the modified step executor of listing 1 on page 55. This example is used to describe the applied bytecode modifications. Thus, line numbers occurring in the following refer to listing 5.

Listing 5: Adjusted Example Step Executor

```java
@StepExecutor (name = "ModuleDeployer",
    description = "Deploys an EJB module into the container.",
    stepType = StepType.MODULE_DEPLOYMENT)
public class DefaultModuleDeployer implements IStepExecutor,
    IPlanStepExecutor {

    @Input (description = "Identifier of the EJB module at the "+"Deployment Level that should be deployed.")
    private String ejbModuleId;

    @Input (description = "Enables the logging", optional = true)
    private boolean verbose = false;

    @Output (description = "The deployment state of the "+"corresponding EJB module after trying to deploy it",
        optional = false)
    private DeploymentState ejbModuleState;

    private boolean ejbModuleStateHasBeenSet;

    @Override // the only method from the IStepExecutor interface
    public void runStep(Container container) {
        // code for module deployment provided by an executor provider
    }

    @Override // method from IPlanStepExecutor interface
    public Map<String, byte[]> executePlanStep(Map<String, byte[]> input, Container container) {
        byte[] ejbModuleIdInBytes_mKernel = input.get("ejbModuleId");
        if (ejbModuleIdInBytes_mKernel == null) {
            if (!false) { // !optional
                throw new ExecutorException("Non-opt param ejbModuleId of step executor ModuleDeployer is not set.");
            }
        } else {
            Object ejbModuleIdInObject_mKernel = this.resolveInput_mKernel(ejbModuleIdInBytes_mKernel);
            this.ejbModuleId = (String) ejbModuleIdInObject_mKernel;
        }

        byte[] verboseInBytes_mKernel = input.get("verbose");
        if (verboseInBytes_mKernel == null) {
            if (!true) { // !optional
                throw new ExecutorException("Non-opt param verbose of step executor ModuleDeployer is not set.");
            }
        } else {
```

Distributed and Mobile Systems Group 63
As depicted in line 4, the adjusted step executor implements the interfaces `IStepExecutor` and `IPlanExecutor`. The `runStep` method of the interface `IStepExecutor` is implemented by an `Executor Provider` (lines 19-22). The `executePlanStep` method of the interface `IPlanExecutor` is generated and added automatically to the executor class and it begins in line 24. This method will be invoked by plan executors that have to provide values for the required input parameters of the step executor through the first method argument `input`. At first, the `executePlanStep` method tries to obtain values for the required input parameters of the step executor from the `input` map. These values are assigned to the corresponding instance fields that are annotated with @Input and that represent input parameters of the step executor. Keys of the `input` map are the field names and values of the map are the corresponding parameter values. E.g., in line 27, values for the parameter `ejbModuleId` are retrieved. Since this input parameter is non-optional, an exception will be thrown if no value has been provided for it (see lines 28-31). This can be seen as a violation of a precondition for the use of the step executor. Thus, if no values for at least one non-optional input parameter have been submitted, the actual task of the step executor, i.e., the `runStep` method, is not executed. In contrast, for optional input parameters, like, e.g., the second input `verbose`, no values have to be provided, such that no exception will be thrown if this really happens (see lines 38-41). If values for optional and non-optional input parameters are provided through the `input` map, these values are converted from bytes to the appropriate type of the parameter and they are assigned to the corresponding instance fields (see lines 32-35 and 42-45). The conversion of values from bytes to `Object` is done by the method `resolveInput_mKernel` that is added automatically to the step executor class. After all instance fields representing inputs or at least the non-optional inputs have been supplied with values, the `runStep` method is invoked (see line 47) and it processes input parameter values simply by using the corresponding instance fields. Execution of the `runStep` method performs the tasks of the step executor. In this example, the EJB module with the identifier `ejbModuleId` will be deployed. During execution of the `runStep` method, values for non-optional output parameters have to be and values for optional output parameters might be assigned to the corresponding instance fields being annotated with @Output. To check whether or
not values have been assigned to outputs, for each output an additional instance field is added to the executor class. E.g., for the output \texttt{ejbModuleState} (see lines 13-16) the field \texttt{ejbModuleState\_HasBeenSet\_mKernel} of type \texttt{boolean} has been added (see line 17). Its default value is \texttt{false} and it is switched to \texttt{true} if a value is assigned at least once to the field \texttt{ejbModuleState} within the \texttt{runStep} methods and its potential helper methods within the executor class. Thus, it is possible to distinguish between default values of output instance fields and values that have been assigned to the fields during reconfiguration. This information is used by the executePlanStep method after the \texttt{runStep} method has finished for returning the output values (see lines 49-58). E.g., the output \texttt{ejbModuleState} is a non-optional output parameter and if no value has been assigned to this field, an exception will be thrown (see lines 51-52). This corresponds to a violation of a postcondition because the a value for a mandatory output parameter is not provided by the step executor at runtime. Otherwise, the assigned value of the field \texttt{ejbModuleState} is returned to the invoking plan executor (see lines 53-58). For the case of optional outputs, assume that \texttt{ejbModuleState} is an optional one. Then, in line 51 the \texttt{false} in the condition is replaced with \texttt{true}. Thus, an exception will not be thrown if no value is assigned to the field \texttt{ejbModuleState}. Nevertheless, the value of this field will only be returned if a value has been assigned at least once to this field (lines 53-56), i.e., if no value assignments have occurred, the initial value at field declaration time will not be returned. For returning values of outputs, the values are converted to bytes, which is done by the method \texttt{toBytes\_mKernel} that is added automatically to the executor class. Finally, all output values are returned within the result map to the invoking plan executor.

To sum up, the task of an Executor Provider to implement a step executor is simplified through the bytecode modifications after development, because an Executor Provider must not write code for the executePlanStep method and its generic parameters. Nevertheless, the generic generatePlanStep method is required for plan executors. A plan executor must be able to invoke all different kinds of step executors that require or provide different kinds of parameters. Thus, all step executors must share a common interface, which can only be solved with generic methods. Furthermore, the task of an Executor Provider is simplified because he or she must not write code for handling the conversion of parameter values from byte arrays to objects of the correct type, and vice versa. Nevertheless, the use of byte arrays for the transmission of parameter values is necessary to avoid class loading errors because executors might be distributed among several EJB modules. E.g., if a plan executor is packaged in a different module than the step executors, the plan executor module need not to have all classes that are types of parameters in its classpath, because a plan executors forwards only the byte arrays among step executors without unpacking them.

Finally, the executePlanStep method runs within a single transaction, i.e., at the beginning of the method, a new transaction is started and just before the method returns, the transaction is committed. If an exception is thrown within the executePlanStep or within the runStep method a rollback of the transaction is performed. For purposes of clarity, this is not depicted in listing 5. Transaction support comes from the EJB container. Each executor, either plan or step executors, becomes a session bean, which is described in the following, such that a bean-managed transaction is integrated into a step executor class together with the executePlanStep method.

After the bytecode modifications of step executors have been finished, the executor module and the executor descriptor of this module are sent to an enterprise bean that is part of
the deployed reconfiguration infrastructure. This bean processes the executor descriptor and generates an EJB deployment descriptor. This deployment descriptor specifies that executors become stateless session beans. E.g., listing 6 shows the deployment descriptor that is generated from the executor descriptor depicted in listing 4 on page 62.

Listing 6: Example Executor Module Deployment Descriptor

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee">
  <enterprise-beans>
    <session>
      <ejb-name>Deploy an EJB module into the container.</ejb-name>
      <description>
        <ejb-name>ModuleDeployer</ejb-name>
        <mapped-name>mKernel/reconfiguration/stepExecutor/
          f0f3c923d9e149838ba97d28d27da25b</mapped-name>
        <business-remote>mKernel_reconfiguration_executor.
          IPlanStepExecutor</business-remote>
        <ejb-class>sample.executors.DefaultModuleDeployer</ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Bean</transaction-type>
        <resource-env-ref>
          <resource-env-ref-name>transaction</resource-env-ref-name>
          <resource-env-ref-type>javax.transaction.UserTransaction</resource-env-ref-type>
        </resource-env-ref>
        <injection-target>
          <injection-target-class>sample.executors.DefaultModuleDeployer</injection-target-class>
          <injection-target-name>ut_mKernel</injection-target-name>
        </injection-target>
      </session>
      <session>
        <description>Plan executor that executes all steps sequentially.</description>
        <ejb-name>PlanExecutorSeq</ejb-name>
        <mapped-name>mKernel/reconfiguration/planExecutor/7
          ac1f0b36daf4f77b5432ee1fccd7408</mapped-name>
        <business-remote>mKernel_reconfiguration_executor.IPlanExecutor</business-remote>
        <ejb-class>sample.executors.DefaultPlanExecutor</ejb-class>
        <session-type>Stateless</session-type>
      </session>
    </session>
  </enterprise-beans>
</ejb-jar>
```

A step executor provides the interface **IPlanStepExecutor** as a remote business interface, such that the **executePlanStep** method might be invoked location-transparently by any plan executor. Likewise, a plan executor provides the interface **IPlanExecutor** as a remote business interface for submitting plans to it which will be executed. Almost all information contained in the deployment descriptor is taken from the executor descriptor. Only the specification of a dependency injection for having bean-managed transaction support within a step executor has been added (see lines 11-19 of listing 6). The generated deployment descriptor is integrated into the binaries of the executor module, such that the executor module becomes an EJB module.
Finally, the last step of the executor module integration process uses the executor descriptor to store meta information about the executor module and its included executors together with the binaries of the module into the database. Thus, information about executor modules and executors can be retrieved through the API that is described in section 7.1.4 and shown in figure 12 on page 59. Furthermore, using the interface `ExecutorModule`, the life cycle of an executor module can be managed. This is the reason why the binaries of executor modules are also stored in the database. After deploying and starting an executor module, executors being part of this module are ready to be used during a concrete reconfiguration.

7.2.2 State Transfer and the Life Cycle of Stateful Session Bean Instances

A state transfer addresses the *Conversational States* of stateful session bean instances. The conversational state is defined as the field values of the stateful session bean instance, its associated interceptors and their instance field values, plus the transitive closure of the objects from all these fields reached by following Java object references [DK06a]. A state transfer is necessary for the replacement of stateful session bean instances and it comprehends the extraction of a state from a replaced instance, an optional conversion of a state and the injection of a state into a replacing instance. To enable a state transfer, the conversational state must be serializable. This is required because the EJB standard does not specify the `system principal`, i.e., it is not defined how many *Java Virtual Machines* (JVMs) an EJB container may use to run instances of enterprise beans. E.g., a container implementation can execute all instances of all enterprise beans in a single JVM or another implementation can use a separate JVM for each EJB module and its bean instances. Thus, it is likely that a state transfer crosses the boundaries of JVMs, which requires a serializable conversational state. Nevertheless, the requirement of a serializable state is fulfilled when a state transfer is performed at certain stages in the life cycle of the affected stateful session bean instances. In the following, this life cycle is presented briefly and it is shown how it can be used to enable a state transfer.

The life cycle of stateful session bean instances is defined by the EJB standard [DK06a]. The simplified life cycle of a stateful session bean instance is depicted in figure 13. It is simplified because it does not cover the influence of transactions on it. Therefore, for details of the life cycle, refer to the EJB standard [DK06a].

![Figure 13: Simplified Life Cycle of a Stateful Session Bean Instance](image-url)
For certain transitions among states of the life cycle, a Bean Provider may define callback methods for a stateful session bean, which are invoked on bean instances by the EJB container during state transitions. Thus, a bean instance is notified about changes in its life cycle if callback methods are provided. The life of a stateful session bean instance starts when an instance is created, e.g., because a client obtains a reference to it through a JNDI lookup or through dependency injection. The state of the instance switches from does not exist to method ready. During this state transition, the PostConstruct callback method, if available, is invoked on the instance. This method usually performs some initializing operations. In the method ready state, an instance is ready for servicing business calls invoked by its client and it actually services calls in this state. An instance can be destroyed because of three reasons. First, a client initiates the destruction, e.g., by calling a business method that is defined as a Remove method. The instance will be destroyed after the call has finished. An invocation of a Remove method signifies the end of a session. Second, a timeout elapsed during which the instance is not used and the container initiates the destruction. For both cases, before an instance is actually destroyed, the PreDestroy callback method, if existent, is invoked on the instance. The last reason for a destruction of an instance is a system exception that is thrown by the instance, which however does not lead to a call of the corresponding PreDestroy method. After destruction, the instance does not exist any more. While being in the method ready state, an idle stateful session bean instance might be evicted from the memory because of performance reasons that the container can efficiently manage the size of the working set of bean instances. The instances and their conversational states are transferred temporarily to secondary storage, which is called passivation. The transfer back from secondary storage to the working set is called activation. The container decides which instances should be passivated, e.g., by selecting the least recently used ones. Nevertheless, instances are not passivated when they are currently engaged in method invocations. Before an instance and its state is saved to secondary storage, the PrePassivate callback method is invoked, if existent. An instance is activated if, e.g., a business method is invoked on a passive instance. During activation, before the instance becomes method ready again, the PostActivate callback is invoked, if existent. Finally, a passive instance can be removed from secondary storage if a timeout elapsed such that it might be assumed that the session will not be continued. For the transition from passive to does not exist no callback method can be defined.

Though a stateful session bean need not to have any callback methods, the EJB standard requires that a Bean Provider ensures that the PrePassivate method leaves the conversational state of a stateful session bean instance ready to be serialized by the container. Thus, either the conversational state of an instance is already serializable and a PrePassivate method is not required, or the PrePassivate method is required and it changes the state such that it becomes serializable.

This requirement that a stateful session bean has to support the passivation of its instances can be used for enabling a state transfer. The idea is to transfer the state after the PrePassivate callback method has been executed. Nevertheless, the passive state of an instance is not helpful for two reasons. First, the management of the life cycle of stateful session bean instances is performed by the container and it should remain the responsibility of the container. Thus, the reconfiguration infrastructure does not take over the life cycle management and it does not force the container to passivate certain instances when a state transfer should be performed. Second, the reconfiguration infrastructure is realized, among others, as a set of standard-compliant enterprise beans and the container implementation should not be adapted. Thus, the infrastructure cannot...
access the secondary storage to retrieve the conversational states, because the secondary storage is usually the file system of the server running the container. The EJB standard forbids enterprise beans to access the file system. Furthermore, withdrawing states from secondary storage affects the container-managed life cycle of bean instances.

One solution is that the reconfiguration infrastructure or step executors make use of the life cycle callback methods of stateful session beans by directly invoking them, comparable to business calls, if necessary. Thus, stateful session bean instances stay in the method ready state, but virtually are in one of the sub-states of method ready, namely sim-predestroyed, active or sim-passive, as it is shown in figure 14. The prefix sim stands for simulated because these states simulate states of the original life cycle. Thus, the enhanced life cycle exists almost virtually. This figure shows the enhanced life cycle of a stateful session bean instance. However, this enhanced life cycle does not affect the original container-managed life cycle, i.e., the container is not prevented from instantiating, passivating, activating or destroying an instance if required by the container. The enhanced life cycle is only relevant for a reconfiguration when stateful session bean instances have to be replaced. Then, the sub-states of method ready are used for a state transfer, which is described in the following.

The states does not exist, passive and active of the enhanced life cycle and transitions among them correspond completely to the states does not exist, passive and method-ready of the original life cycle and their transitions, respectively. If no replacement of stateful session bean instances takes place, instances are only in one of these three states of the enhanced life cycle. In case of a replacement, an instance that is going to be replaced starts its life cycle and its replacing instance continues this life cycle. If a stateful session bean should be replaced, all of its instances have to be quiescent to ensure consistency of the reconfigured application. This is described in section 2.3. Thus, clients are not able to invoke any business method on any of those instances. Instances that should be replaced are in the active or passive state. Before their states are extracted, the PrePassivate method should be invoked on each instance through a step executor that performs the

![Figure 14: Enhanced Life Cycle of a Stateful Session Bean Instance](image-url)
state extraction. Since this invocation is a business call for the container, instances in the passive state are activated and the PostActivate callback method is invoked by the container. Thus, all instances are in the active state, and the call of the PrePassivate methods on them through a step executor, transfers them to the sim-passive state. If one instance invokes within its PrePassivate method a business method on another instance that is already in the sim-passive state, the reconfiguration infrastructure activates this instance through the PostActivate method, such that the call can be serviced. After the call has been finished, the infrastructure invokes the PrePassivate method in order place the instance in its original sim-passive state. Thus, all instances are able to reach the sim-passive state. After all affected instances are in this state, all conversational states are serializable, such that they might be extracted from the instances.

All replacing instances are in the state does not exist. Through the mKernel API new instances of the replacing stateful session bean can be created which results in active instances. Nevertheless, the PostConstruct callback method, if any, invoked by the container cannot be blocked because it cannot be distinguished whether an instantiation originates from mKernel or from a client requiring an instance. E.g., the Interrupt/Non-Interrupt strategy, being described in section 8, performs a transfer of stateful session bean instances during which other instances of the replacing bean can be created and used by clients. Thus, after the creation of replacing instances, the PreDestroy method, if any, is invoked to compensate the PostConstruct method call. Replacing instances are now in state sim-destroyed during which the extracted conversational states can be injected into them. After injection of states, the PostActivate method should be invoked on replacing instances through the step executor that performs the state injection to compensate the PrePassivate method invocation on replaced instances before the state extractions. Thus, replacing instances are now in the active state and continue the original life cycle of their replaced instances.

If the container destroys an instance in state sim-destroyed the PreDestroy callback invocation by the container is blocked by an specific interceptor, i.e., the call does not reach the bean instance, because the instance is already destroyed virtually by the previously invoked PreDestroy method. Likewise, during the passivation of instances in the sim-destroyed state, invocations of the PrePassivate callback by the container is blocked, because the conversational states that are injected into these instances are already in a passivated state. Comparably, for passivation of instances in the passive state, calls of the PrePassivate callback initiated by the container are blocked, because the instances have already been passivated virtually with the PrePassivate method invoked by a step executor or by the infrastructure. Likewise, during destruction of sim-passive instances by the container, calls of the PreDestroy callback are blocked, which corresponds to the destruction of passive instances for which no callbacks are invoked by the container.

To sum up, the life cycle and the life cycle callback methods of stateful session bean instances are used to enable a state transfer because during certain life cycle states it can be guaranteed that the conversational state is serializable. This is already required by the EJB standard, such that this approach does not impose further restrictions that have to be followed by the Bean Provider to develop enterprise beans.

In the following, the implementation of the enhanced life cycle is presented briefly, such that the reconfiguration infrastructure and step executors can make use of the life cycle callback methods of stateful session beans. Life cycle callback methods are intended to be invoked by the container. Therefore, they need not be offered as business methods to clients and they may have an arbitrary access modifier, even a private one. An interplay
of \textit{mKernel}, of an interceptor, called \textit{life cycle interceptor}, and of the bean instance enables the invocation of life cycle callback methods by the reconfiguration infrastructure or by step executors. The interceptor is attached to each stateful session bean. It intercepts invocations of life cycle callbacks that are invoked by the container on the corresponding bean instance and it manages the transitions among the \textit{sim-destroyed}, \textit{active} and \textit{sim-passive} states of its bean instance (see the part of the enhanced life cycle in figure 14 on page 69 that is shaded gray). Thus, only the \textit{PostActivate}, \textit{PreDestroy} and \textit{PrePassivate} have to be invoked by the infrastructure or by step executors. To enable this, the preprocessing tool \textit{managizer} of \textit{mKernel}, being described in section 5, is enhanced. Internally, the \textit{managizer} consists of several processing modules each performing a certain task within the preprocessing of an EJB-JAR file. Thus, one processing module has been added to the \textit{managizer} that performs the following steps:

- For each stateful session bean being part of the processed EJB module, the names of the \textit{PostActivate}, \textit{PreDestroy} and \textit{PrePassivate} callback methods, if existent, are obtained.

- Three methods being defined by a Java interface are added to each stateful session bean being part of the processed EJB module. This is done with the help of bytecode modification using \textit{Javassist} [Chi08]. Each of these three methods represent either the \textit{PostActivate}, \textit{PreDestroy} or \textit{PrePassivate} callback method. If the corresponding callback exists, which is checked by the previous step, each of the three methods only invokes the corresponding callback method internally. Otherwise, the method body is empty. Thus, these three methods are delegates for the callbacks, but they have a \textit{public} access modifier. Consequently, each stateful session bean implements the interface with these three methods, such that the methods might be invoked by an interceptor that is attached to the corresponding beans.

- The generic \textit{life cycle interceptor} that manages the enhanced life cycle of stateful session bean instances is attached to each stateful session bean. Thus, a generic parameterized method to address this interceptor is sufficient and the interceptor invokes the corresponding delegate method, being defined by the previous step, on the bean instance. Thus, the interface with the delegate methods need not to be provided as a business interface to clients.

Such a generic method, called \textit{performAdministrativeTask}, is provided by the \textit{mKernel API} for \textit{HoldingReferences}, i.e., for quiescent session bean instances which are part of a \textit{QuiescenceRegion}, like replaced stateful session bean instances, or which have been created newly, like replacing stateful session bean instances. Thus, during a state transfer, this generic method can be used for replaced and replacing stateful session bean instances to place them into one of the three states \textit{sim-destroyed}, \textit{active} or \textit{sim-passive}, which is appropriate for a state transfer during a stateful session bean replacement.

### 7.2.3 State Transfer and Persistence

The EJB standard defines what might be part of a conversational state, after the \textit{PrePassivate} method has been executed, such that a container is able to passivate the stateful session bean instance (see p.63f. of [DK06a]). According to this standard, a reference to a
container-managed `EntityManager` and a reference to an `EntityManagerFactory` might be part of the state and the container must be able to save and restore the references across a passivation and activation even if they are not serializable. Furthermore, an entity manager factory or an entity manager is associated to a certain persistence unit or persistence context, which is described in section 4.2.3. Additionally, the conversational state might contain references to `Entities` which might be in so called `managed` or `detached` states. Thus, a transfer of an `EntityManager`, of an `EntityManagerFactory` or of entities has to consider aspects that go beyond a simple serialization and deserialization. In the following, it is described how a transfer of an `EntityManager`, of an `EntityManagerFactory` and of entities is realized by this work. Most of the other potential parts of a conversational state, especially references to other bean instances, are already handled by `mKernel`.

**Entity Manager and Entity Manager Factory** The basic idea for transferring an entity manager or an entity manager factory is to wrap them with a serializable object. Therefore, two wrappers, one for the entity manager factory and one for the entity manager, are defined:

- The wrapper `ManagedEntityManagerFactory` holds internally a reference to an `EntityManagerFactory` provided by the container. The wrapper implements the interface `javax.persistence.EntityManagerFactory` and delegates calls to this reference. The wrapper also contains meta information about the factory, namely the name of the corresponding persistence unit and the name under which the factory is bound in the local JNDI context of the enterprise bean that uses the factory.

- Likewise, the wrapper `ManagedEntityManager` holds internally a reference to an `EntityManager` that is provided by the container. The wrapper implements the interface `javax.persistence.EntityManager` and delegates calls to this reference. Meta information being held by the wrapper are, among others, the name of the corresponding persistence unit, the name under which the entity manager is bound in the local JNDI context of the enterprise bean that uses the entity manager, a boolean indicating whether the entity manager is container- or application managed, and the type of the persistence context the entity manager is associated with.

The persistence context type defines the lifetime of a container-managed persistence context and its entity manager. The lifetime may be either scoped to a transaction (transaction-scoped persistence context) or it may have a lifetime that extends beyond the scope of the lifetime of a single transaction (extended persistence context). The type of a context is defined, when an instance of an `EntityManager` is created. The scope of application-managed persistence contexts and entity managers is always extended. For details, refer to the *Java Persistence API* specification [DK06b].

In order to influence the serialization and deserialization of wrapper instances, both wrappers implement methods as specified by the `java.io.Serializable` interface. During serialization of a wrapper instance only the meta information being held by the instance are serialized but not the reference to the `EntityManager` or `EntityManagerFactory` because the container implementations of both of them need not to be serializable. Indeed, e.g., an `EntityManager` instance being passed by value from one bean instance to another through a remote business interface, could not be utilized in the receiving instance. Thus, the `EntityManager` implementation of the Glassfish Application Server, version 2.1 [gla]
is not properly serializable. Nevertheless, this is not really a disadvantage of the EJB component model because entity managers and entity manager factories are intended to be used within the local environment of an enterprise bean (see chapter 16, *Enterprise Bean Environment*, of the EJB standard [DK06a]). Even, references to entity manager instances need not to passed from one bean to another through local interfaces because a persistence context is propagated across entity manager instances with a transaction. I.e., both bean instances may individually obtain an appropriate entity manager instance and both instances share the same persistence context (refer to *Persistence Context Propagation* in the EJB specification [DK06b]).

When a wrapper instance is deserialized the meta information that have been serialized before might be used to obtain a reference to an instance of the corresponding entity manager or entity manager factory from the container. For this purpose, the serialized meta information are sufficient.

Since a state transfer serializes the conversational state during state extraction from a replaced stateful session bean instance and deserializes the state during injection of the state into a replacing stateful session bean instance, the transfer of entity manager instances and entity manager factory instances are limited to the transfer of meta information. Nevertheless, the transferred meta information might not be properly usable during deserialization at the replacing session bean instance. E.g., the replacing and replaced modules have defined compatible persistence units, but the replacing one might have defined a different name for the unit than the replaced module. A unit name identifies a persistence unit within an EJB module (compare to paragraph *Persistence in the EJB Component Model* in section 4.2.3). Thus, the transferred unit name is not helpful at the replacing module. Therefore, mappings between persistence units might be specified during a state transfer to map the transferred unit name to the corresponding unit name defined by the replacing module. Currently, only 1-to-1 mappings from persistence units of the replaced module to units of the replacing module might be specified. If no mapping is specified for a transferred unit name, it is assumed that the unit name has not changed and there exists an appropriate unit with this name in the replacing module. These mappings and the transferred meta information are used during state injection, when a wrapper instance is deserialized, to obtain an instance of an entity manager or of an entity manager factory for the appropriate persistence unit from the container. Finally, the implementation currently supports only changes of persistence unit names and 1-to-1 mappings among units, but this can be seen as a proof of concept that simple mappings are feasible and more complex mappings are potentially feasible.

Nevertheless, a distinction has to be made whether a wrapper instance is deserialized in the context of a state injection or in any other context, like, e.g., a step executor for the step *Conversational State Conversion*, being described in section 6.3.1, unpacks an extracted conversational state that contains these wrappers. The first mentioned context is a state injection in the replacing EJB module where appropriate persistence units have been defined and an instance of an entity manager or entity manager factory can be obtained. Other contexts, like, e.g., an executor module, need not to have defined appropriate persistence units because they do not work on the same database as the reconfigured EJB modules. Thus, within other contexts, an attempt to obtain an appropriate instance of an entity manager or entity manager factory causes likely an error. Therefore, during deserialization, the default behavior of a wrapper instance is to make no attempts to obtain an entity manager or entity manager factory instance. However, an interceptor is attached to each stateful session bean of reconfigured applications. This interceptor is able
to recognize if a state injection is in progress and if this is the case, the default behavior of wrapper deserialization is changed. Thus, a wrapper instance actually attempts to obtain an instance of an entity manager or entity manager factory, which is successful because replacing and replaced module have defined equivalent persistence units and, if required, mappings among units may address differences in the unit names. If the replacing module does not specify an appropriate persistence unit, then it does not operate on exactly the same database as the replaced module. Thus, it makes no sense to transfer an entity manager or an entity manager factory for the persistence unit of the replaced module to the replacing one. Therefore, a step executor for the step Conversational State Conversion should remove entity managers or entity manager factories from the extracted conversational states before these states are injected. To cover such scenarios by adapting the extracted conversational states, such that the states can be injected without any problems into the replaced stateful session bean instances, is the task of the step Conversational State Conversion.

The use of the wrappers have to be transparent to the application such that the business logic is not affected. Furthermore, the wrappers have to be transparent to Bean Providers in order to maintain the idea of separation of concerns, such that the provider is not concerned with code, like, e.g., the wrappers, that is required for reconfiguration. Therefore, the wrappers are integrated into an EJB module through the mKernel preprocessing tool manager by analyzing and modifying the bytecode of enterprise beans and the deployment descriptor of the module using Javassist [Chi08] and the Java Architecture for XML Binding (JAXB) [Kaw06], respectively. This is described briefly in the following.

- Dependencies of a bean to EntityManagers and EntityManagerFactories might be declared through metadata annotation in the source code of the bean or through the deployment descriptor. To find these dependencies, the bytecode of beans and the deployment descriptor, both being part of the preprocessed module, are analyzed.

- To integrate the wrappers into the EJB module, the specifications of dependency injections of EntityManagers and EntityManagerFactories are removed from the bytecode of enterprise beans and from the deployment descriptor that are part of the module. Thus, these dependency injections are not performed at runtime. In exchange, the dependencies to EntityManagers and EntityManagerFactories are fulfilled within the PostConstruct callback methods of the enterprise beans. If a bean does not specify a PostConstruct method, a completely new method will be added to the bean. Otherwise, the existing PostConstruct method is modified. Within the PostConstruct method, the corresponding bean instance fields, that are either of type javax.persistence.EntityManagerFactory or of type javax.persistence.EntityManager, are initialized with the appropriate wrappers. During initialization of a wrapper, the wrapper obtains internally a reference to an EntityManager instance or to an EntityManagerFactory instance through a JNDI lookup. Thus, the dependencies of enterprise beans to entity managers and entity manager factories are fulfilled appropriately through JNDI lookups instead of dependency injections. The current implementation covers only the integration of wrappers for entity managers and entity manager factories that are injected into instance fields of enterprise beans. Dependency injections into the setter property methods of the corresponding instance fields and JNDI lookups performed by beans themselves are not addressed.
Finally, meta information about entity managers and entity manager factories are added as *Simple Environment Entries* to the context of enterprise beans. E.g., it is possible to obtain the name under which an entity manager or an entity manager factory is bound in the local JNDI context of a bean when the name of the persistence unit is known. Thus, these *Simple Environment Entries* are used when an entity manager or an entity manager factory wrapper is deserialized during state injection into a stateful session bean instance. During deserialization, the name of the persistence unit is known by the wrapper, because the name has been transferred and possibly mapped to a proper name (see above). But to get a reference to an `EntityManager` or to an `EntityManagerFactory` instance through a JNDI lookup, the wrapper must know the appropriate name under which an `EntityManager` or an `EntityManagerFactory` is bound in the local JNDI context of the replacing bean. This name is obtained from a *Simple Environment Entry* of the replacing bean. Though, a transfer of a wrapper includes a JNDI name, this name corresponds to the name under which an `EntityManager` or an `EntityManagerFactory` is bound in the local JNDI context of the replaced stateful session bean. Thus, regarding entity managers and entity manager factories, the names within the local JNDI contexts of replaced and replacing bean need not to be same and there is no requirement that they have to be the same.

To sum up, with this approach the transfer of entity manager instances or entity manager factory instances being part of a conversational state of a stateful session bean instance is facilitated without affecting the business logic of the application and without giving up the idea of separation of concerns.

**Entities**  As described in the paragraph *Persistence in the EJB Component Model* in section 4.2.3, an entity is a lightweight persistent object. In the following, the approach of this work is described to transfer entity instances being part of a conversational state of a stateful session bean instance is facilitated without affecting the business logic of the application and without giving up the idea of separation of concerns.

The *Java Persistence API* specification [DK06b] defines a life cycle for an entity instance. Regarding a state transfer, the relevant states of an entity instance are the managed and the detached states. A managed instance has been persisted and it is currently associated with a persistence context. A detached entity instance has also been persisted, but it is no longer associated with a persistence context. Being associated with a persistence context ensures that the entity instance is or can be synchronized to the underlying database. In contrast, detached entity instances live outside of a persistence context and their state is no longer guaranteed to be synchronized to the database. Synchronization to the database happens at transaction commit or when explicitly invoking the *flush* operation of an `EntityManager`. A *flush* invoked on a `EntityManager` synchronizes all entity instances of the persistence context that is managed by this `EntityManager` to the database. However, a synchronization does not refresh any managed entity instances with the current database state unless the *refresh* operation of an `EntityManager` is invoked explicitly for each entity instance. Since a state transfer is performed during a quiescent state, all transactions have committed. Thus, for a state transfer, the entity instances are synchronized to the database. Regardless of a state transfer, ensuring that entity instances have always the current state as the database state is not possible. The reason is that immediately after a *refresh* of an entity instance, the refreshed data is stale, because any other piece of software, even the same application, might change the data of the entity through modifying an corresponding entity instance being part of a
different persistent context. Thus, a state transfer that involves entity instances need not to consider any aspects regarding the current synchronization among entity instances and databases. Nevertheless, a state transfer should consider whether entity instances being part of a conversational state are associated to a persistence context or not. Thus, it should be distinguished between managed and detached instances. Consequently, managed entity instances being part of the conversational state of a replaced stateful session bean instance should be in the managed state after they are injected, together with the rest of conversational state, into a replacing bean instance. Likewise, detached entity instances should remain detached throughout a state transfer. Whether or not a conversational state contains managed entity instances depends on the type of the persistence context that is used by a bean instance. If a transaction-scoped container-managed persistence context is used, managed entity instances being part of the conversational state become detached at transaction commit because the lifetime scope of the transaction determines the lifetime scope of the persistence context. Thus, after transaction commit, only detached entity instances are part of the conversational state. In contrast, if an extended persistence context is used, which is either an application-managed or an extended container-managed context, managed entity instances remain managed after transaction commit, unless, e.g., the persistence context has not been cleared or the EntityManager of an application-managed persistence context has not been closed before transaction commit. Furthermore, entity instances may have become part of a conversational state, because the instances might have been passed by value to a stateful session bean instance. Serialization of entity instances detaches these instances. Thus, a conversational state of a bean instance that uses an extended persistence context may contain managed and detached entity instances. Consequently, the type of persistence context is not helpful for identifying whether all entity instances are managed or detached. This has to be checked for each entity instance individually. Nevertheless, only if an extended persistence context is used, managed entity instances might be part of a conversational state. Thus, for a state transfer, when an entity instance as part of the conversational state should be extracted from a stateful session bean instance, it should be diagnosed whether the entity instance is managed or detached. This information, i.e., a boolean value indicating if the instance is managed or not, should be extracted together with the entity instance, such that it can be used for state injection. A state extraction serializes the state, such that all managed entity instances being part of the state become detached. When an entity instance should be injected, the transferred boolean value can be used to decide whether or not the entity instance should be attached to an appropriate extended persistence context in a replacing stateful session bean instance, i.e., whether the instance should be in the detached or managed state. An appropriate persistence context refers to the persistence unit that contains the class of the transferred entity instance. Such an extended persistence context, to which transferred entity instances might be attached, will be shared by all entity manager instances which have also been transferred or which will be created by the replacing bean instance if the entity manager instances refer to the corresponding persistence unit. The same holds for entity manager instances being obtained from entity manager factory instances that have been transferred or that are created by the replacing bean instance if the factory instances refer to the corresponding persistence unit. Thus, after a state injection, a replacing bean instance is able to start its work with managed entity instances as part of its conversational state like the replaced instance has stopped its work.
To realize the transfer of managed entity instances, all entities of an EJB module are adjusted. This is done transparently to the Bean Provider through the mKernel preprocessing tool managizer. In the following, the adjustments of entities and how they are implemented within the managizer are discussed briefly. The realization of the entity adjustments involve several steps:

- First, the META-INF/persistence.xml file is parsed using the Java Architecture for XML Binding (JAXB) [Kaw06]. Within this file, persistence units are defined for the EJB module. If no persistence unit is specified in this file, the preprocessed EJB module does not make any use of entities. Thus, no entity classes are part of the EJB module, which requires no further actions. Otherwise, there exist at least one persistence unit and the module contains at least one entity class. An entity class is either a class that is annotated with @Entity or that is specified as an entity in a XML-based mapping file. An EJB module might have several mapping files which are referenced by the META-INF/persistence.xml file and which specify the object-relational mappings of entities. A mapping file is not mandatory, because object-relational mappings of entities can also be expressed through metadata annotations applied to the source code of entity classes. Nevertheless, information about the object-relational mapping of entities are not relevant here. Metadata annotations and mapping files can be used to identify which classes of the EJB module are entities. Therefore, the current implementation only relies on metadata annotations and analyzes the bytecode of classes with the help of Javassist [Chi08] to find @Entity annotations. Potential mapping files are neglected. Furthermore, information from the META-INF/persistence.xml file is used to map entity classes to persistence units. Currently, the implementation requires that an entity class must be part of exactly one persistence unit. Finally, after this step, all entity classes being part of the EJB module and the persistence unit of each entity class are known.

- Each entity class being part of the preprocessed module is adjusted through bytecode modification using Javassist [Chi08] and Velocity [vel]. Methods as specified by the java.io.Serializable interface are added to the each entity class to influence the serialization and deserialization of entity instances. Furthermore, an additional class is integrated into the EJB module that holds a map to obtain the name of the persistence unit for a given fully qualified name of an entity class that is part of the same EJB module. This map might be accessed statically from code within the added entity methods. Thus, within the added methods, an entity instance is able to obtain an entity manager instance for its persistence unit in order to check whether it is managed or not, or to attach itself to a persistence context. When an entity instance is serialized during a state extraction, the instance is completely written to the stream and afterwards, additional information is written also to the stream. Writing data additional to the actual object instance to the stream is possible as specified by the Java Object Serialization Specification [Sun05]. This additional data is only a boolean value that indicates whether the entity instance is managed or detached. To get this information during serialization, an entity instance obtains an EntityManager, referring to the appropriate persistence unit, that manages the same extended persistence context as used by the application. The contains operation of this EntityManager is used to check whether or not the entity instance is part of this extended context, i.e., whether the instance is managed or detached.
During state injection, when an entity instance is deserialized, the instance is re-
constructed automatically by the Java Object Serialization technology. Then, the
additional data, indicating whether the instance has been in the managed or de-
tached state within the replaced stateful session bean instance, is read from the
stream. If the entity instance has been in the managed state, an EntityManager
for the extended persistence context of the replacing stateful session bean is obtained
and the entity instances is attached to this context. Thus, within the replaced bean
instance, the injected entity instance is also in the managed state. If the entity in-
stance has been in the detached state within the replaced bean instance, it remains
detached within the replacing bean instance.

Thus, a transfer of entity instances that considers the state of entity instances is feasi-
ble. Moreover, the adjustments of entities are transparent to the application, because
the changes of entity classes comprehend only the addition of methods as specified by
the interface java.io.Serializable. Additionally, these added methods do not modify
the default (de)serialization of entity instances. Thus, a changed entity class remains
compatible to the original, i.e., unmodified, class (compare to the Java Object Serializa-
tion Specification [Sun05]). Consequently, an instance of a changed entity class might be
passed by value, e.g., to a client application outside the EJB container that is not aware
of the changed entity class and only knows about the original class. Thus, at the client
application side, the entity instance can be deserialized properly, and the additional data,
i.e., whether the entity instance has been in a managed or detached state, is ignored.

Thus, the additional data might always be written to the stream when an entity instance is
serialized within the preprocessed EJB module. Nevertheless, this data is only processed
during deserialization of an entity instance if the instance is injected into a bean instance
as part of a state transfer. Otherwise, e.g., if an application passes entity instances by
value from one bean instance to another, the default behavior is to ignore the additional
data. Consequently, the default behavior for the deserialization of an entity instance is
equivalent to the default Java deserialization, i.e., as if no adjustments have been done to
the entity classes to influence the (de)serialization of corresponding instances. Thus, the
normal execution of an application is not affected and the changes of entity classes are
transparent to the application. To change the default behavior, an interceptor is attached
to each stateful session bean that recognizes if a state is currently injected into a bean
instance and that directs entity instances to process the additional data during deserial-
ization.

Finally, within a state transfer, i.e., between a state extraction and state injection, a
step executor realizing the step Conversational State Conversion might convert the state.
Therefore, a state must be unpacked, which involves a deserialization of the state, and
after converting the state, it must be packed, which involves a serialization of it. Thus, en-
tity instances as part of a conversational state might be serialized and deserialized within
an executor module, i.e., within the module that contains the step executor. During
state conversion, the additional information that indicate whether entity instances have
been in the managed or detached state and that have been added during state extraction
should not get lost. Therefore, the step executor must work with the modified entity
classes and not with the original ones. After the implementation of the step executor,
it is packaged together with all related classes, including classes of business objects that
are part of the conversational state that should be converted, into a JAR that has to
be integrated into the reconfiguration infrastructure. This integration is described in the
section 7.2.1 and it performs, among others, the same adjustments of entity classes as it is
done during the preprocessing of EJB modules by the manager. Thus, the adjustments of entity classes during executor integration is described here. The only difference to the preprocessing through the manager is that the entity classes that should be modified during the integration of an executor module cannot be identified automatically, because step executors might use the entity classes as simple Java classes without any persistence support. Thus, among others, an executor module need not to have a persistence.xml file that addresses the relevant entity classes and the entity classes need not to have any metadata annotations. Consequently, when an executor module is integrated into the infrastructure, the classes that are entities of the reconfigured application and that are part of the executor module, if any, have to be indicated to identify the set of classes that should be adjusted. Nevertheless, the adjustments of entity classes are transparent to the Executor Provider. Within the methods that are added to application entity classes, as described above, the case that entity instances as part of conversational states might be deserialized and serialized by a step executor is considered. An entity instance recognizes this case because step executors have to signalize it. Thus, an entity instance is aware of its environment, i.e., an executor module, and it changes its (de)serialization behavior as follows. When it is deserialized, the instance and the additional information is stored in a map that might be statically accessed. Thus, a step executor might query the map to find out whether an entity instance has been in the managed or in the detached state. Finally, when an entity instance is serialized, the additional information of an entity instance is obtained from the map and it is written to the stream together with the entity instance. Thus, the map administrates the additional information obtained trough state extraction throughout the state conversion, such that these information are available for state injection.
8 Design and Realization of Provided Strategies

This section presents a proof of concept for the reconfiguration approach of this work. Thus, it is shown that the reconfiguration model and the reconfiguration infrastructure, which are described in section 6 or 7, respectively, can be utilized to reconfigure an EJB-based application. As a reconfiguration scenario, the replacement of an EJB module is chosen. At first, the design of four reconfiguration strategies is discussed. Each of these four strategies might be used for a module replacement, but there is no constraint that a module replacement is limited to these strategies. Afterwards, the realization of these strategies and the corresponding executors are presented briefly together with their restrictions and assumptions.

8.1 Design of the Provided Strategies

This section describes four reconfiguration strategies that are designed and realized as a proof of concept for this approach. Three of these provided strategies are the ones presented by Rosa, Rodrigues and Lopes [RRL07] and they are discussed in section 2.4, namely Flash (F), Non-Interrupt (NI), and Interrupt (I). The fourth strategy has been identified as a mixture of the strategies I and NI for the replacement of an EJB module. Therefore, it is called Interrupt/Non-Interrupt (I/NI).

The idea of the I/NI strategy is that the replaced and replacing module are running concurrently, but newly created sessions are forwarded to the replacing module and start processing immediately. Already running sessions on the old module will not run until they finish, like it is done with the NI strategy. In contrast, they are driven into a quiescent state and their instances of the stateful session beans are transferred to the replacing module, where finally the sessions continue their processing. Here, a session can be seen as a conversation between a client and a server, and the server-side application contains stateful session beans. The advantage of I/NI is that system disruption is minimized because newly created sessions are not blocked from servicing requests. Therefore, the underlying database must be usable by both modules concurrently. Furthermore, the replaced module is removed from the system consistently, when no clients are using instances of this module.

In the following, the design of the four strategies F, NI, I, and I/NI are presented, i.e., how steps can be combined into procedures that realize these strategies. Table 2 on page 81 shows four procedures, each of which realizes one of the four strategies. The first column contains identifiers for the steps, and the second column contains the names of the steps as they are presented in section 6.3.1. The third column contains the transitive dependencies of each step, while a ‘/’ depicts that the particular step does not depend on any other step. E.g., starting an EJB module requires that the module has been deployed before, therefore, step o depends on step b if both steps address the same module. As some steps might be optional, some dependencies might be dropped if one or more optional steps are not applied within a procedure. Then, all of the remaining dependencies have to be fulfilled by a procedure. E.g., if a state transfer is not necessary because only stateless session beans have to be reconfigured, the steps j, n, p, and q can be omitted, and the step of stopping the replaced module (v) does not depend on step q, but only on the steps k, r, and s. Thus, the dependency from step v to step q does not need to be fulfilled by a procedure. Consequently, dependencies are influenced by a concrete arrangement of steps.
### Table 2: Reconfiguration procedures and their steps

<table>
<thead>
<tr>
<th>ID</th>
<th>Step</th>
<th>dep.</th>
<th>F</th>
<th>NI</th>
<th>I</th>
<th>I/NI</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Module Creation</td>
<td>/</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>Module Deployment</td>
<td>a</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>EJB References Connection</td>
<td>b</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>d</td>
<td>Simple Environment Entry Modification</td>
<td>b</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>e</td>
<td>Quiescence Region Declaration</td>
<td>j</td>
<td>-</td>
<td>-</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>f</td>
<td>Quiescence Region Tracking</td>
<td>e</td>
<td>-</td>
<td>-</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>g</td>
<td>Delay</td>
<td>f</td>
<td>-</td>
<td>-</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>h</td>
<td>Quiescence Region Blocking</td>
<td>g</td>
<td>-</td>
<td>-</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>i</td>
<td>Waiting for Quiescence</td>
<td>h</td>
<td>-</td>
<td>-</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>j</td>
<td>Conversational State Extraction</td>
<td>i</td>
<td>-</td>
<td>-</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>k</td>
<td>Database Extraction</td>
<td>j</td>
<td>-</td>
<td>-</td>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>l</td>
<td>Database Conversion</td>
<td>k</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>-</td>
</tr>
<tr>
<td>m</td>
<td>Database Injection</td>
<td>l</td>
<td>-</td>
<td>-</td>
<td>13</td>
<td>-</td>
</tr>
<tr>
<td>n</td>
<td>Conversational State Conversion</td>
<td>m</td>
<td>-</td>
<td>-</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>o</td>
<td>Module Starting</td>
<td>c, d</td>
<td>5</td>
<td>5</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>p</td>
<td>Conversational State Injection</td>
<td>n, o</td>
<td>-</td>
<td>-</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>q</td>
<td>Reference Publication</td>
<td>p</td>
<td>-</td>
<td>-</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>r</td>
<td>Newly Established Connection Rerouting</td>
<td>m, o</td>
<td>6</td>
<td>6</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td>s</td>
<td>Existing Connection Rerouting</td>
<td>m, o</td>
<td>-</td>
<td>7</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>t</td>
<td>Quiescence Region Release</td>
<td>q, r, s</td>
<td>-</td>
<td>-</td>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>u</td>
<td>Quiescence Region Destruction</td>
<td>t</td>
<td>-</td>
<td>-</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>v</td>
<td>Module Stopping</td>
<td>k, q, r, s</td>
<td>7</td>
<td>8</td>
<td>22</td>
<td>19</td>
</tr>
<tr>
<td>w</td>
<td>Module Undeployment</td>
<td>v</td>
<td>8</td>
<td>9</td>
<td>23</td>
<td>20</td>
</tr>
<tr>
<td>x</td>
<td>Module Destruction</td>
<td>w</td>
<td>9</td>
<td>10</td>
<td>24</td>
<td>21</td>
</tr>
</tbody>
</table>

that may skip optional steps and by the concrete modules and enterprise beans each step is addressing. For the specification of a concrete strategy, the dependencies, which actually have to be fulfilled, have to be considered either through parameter mappings or through explicit dependencies between steps (see section 6.3.3). Furthermore, it is conceivable, that the utilized step executors require parameter mappings among them that introduce additional dependencies to those listed in the third column of table 2.

Each of the last four columns of table 2 shows a realizing procedure for the corresponding strategy that fulfills the dependencies between its steps. The entries of these columns are to be read as follows. A step that is not applicable or available within a strategy is denoted with a "-". Otherwise, the number indicates the position of this step within the procedure, as if the procedure is a linear sequence of steps. Nevertheless, these four procedures are exemplary realizations of the strategies, because the order of steps might change as long as the dependencies between the steps are fulfilled. Thus, it is conceivable that a strategy is realized by several procedures, i.e., different orders of steps. In the following each of these procedures and how their steps cooperate are described in detail. The goal of each procedure is the replacement of an EJB module with an alternative implementation for this module.
The first four steps and the last three ones are the same for each procedure. Thus, they are presented here for all procedures, while the following subsections focus on the specifics of each procedure and strategy.

The steps *Module Creation* \((a)*, *Module Deployment* \((b)*, *EJB References Connection* \((c)*, and *Simple Environment Entry Modification* \((d)* perform together the integration of an EJB module into an EJB-based application. In case of a module replacement, these steps address the replacing module. Therefore, the preprocessed JAR file of the replacing module has to be integrated into the \(mKernel\) system by loading it up. At this point, the reconfiguration performed by the procedures starts. The step *Module Creation* \((a)* retrieves the replacing module type at the *Type Level* from the \(mKernel\) repository and creates a deployable module at the *Deployment Level*. This module is deployed in the container by the second step *Module Deployment* \((b)*. Thus step \(b\) depends on step \(a\) (see third column, second row of table 2). The following two steps, namely *EJB References Connection* \((c)* and *Simple Environment Entry Modification* \((d)*, perform the configuration of the enterprise beans being part of the replacing module. Both steps are only required, if the replacing module contains enterprise beans whose receptacles have to be connected to business interfaces or whose *Simple Environment Entries* have to be assigned values, respectively. Therefore, the receptacles of those beans have to be connected to provided interfaces of session beans that are part of any deployed module, except the one which is going to be replaced. Furthermore, appropriate values for the *Simple Environment Entries* provided by the beans being part of the replacing module have to be set. This configuration is performed after the deployment of the module, such that both steps, \(c\) and \(d\), depend on step \(a\). Nevertheless, configuration has to be done before the module has been started (step \(o\)). Otherwise, the enterprise beans of the module might not be well-configured and might not work appropriately if clients would start using instances of them. Hints for the configuration of the beans of the replacing module might be obtained from the bean configuration of the replaced module on the condition that the beans of the replaced module have been configured well.

The last three steps of each procedure address the replaced module that is running in the container and available to clients. At first, the module is stopped by the step *Module Stopping* \((v)*, such that it becomes unavailable to clients and no operations can be performed on this module. Since the replaced module is not needed any more, it can be undeployed (Step \(w\)) and finally destroyed (Step \(x\)) in order to remove it from the container and to remove the *Deployment Level* representation of it from the \(mKernel\) system, respectively. This representation is an *EjbModule*, which is presented in section 5. Since only stopped modules can be undeployed, step \(w\) depends on step \(v\). Furthermore, an *EjbModule* can only be removed from the \(mKernel\) information model, if the module is not deployed. Consequently, step \(x\) depends on step \(w\).

Regarding the other steps of the procedures, there are differences among the four strategies. E.g., some steps are not applied within a strategy or they are applied in different orders. Therefore, each procedure together with its realized strategy is presented separately in the following subsections.

### 8.1.1 Flash

In comparison to the other procedures, the procedure realizing the *Flash* \((F)* strategy (see the fourth column of table 2 on page 81) is the simplest one. It only addresses the *Deployment Level* and disregards the *Instance Level* of the reconfigured application. Ad-
tionally, the objective of preserving the consistency of the application is neglected. Consequently, existing bean instances and connections among them are not handled. Thus, a state transfer of Conversational States of stateful session bean instances is not performed, which skips the steps j, n, p, and q. Likewise, a database reconfiguration is not considered, which leads to the omission of the steps k, l, and m. Not addressing existing connections among bean instances, the procedure does not make use of step s. Performing no state transfer or database reconfiguration, the replaced module is not placed in a quiescent state, such that the steps e, f, g, h, i, t, and u, which realize the concept of quiescence, are not applied. Thus the switch to the replacing module is only done by the steps o and r.

The replacing module is started with the step Module Starting (o) after the deployment and configuration of its enterprise beans. The subsequent step Newly Established Connection Rerouting (r) directs the clients of the replaced module to use the replacing module when connections are newly established. Consequently, the replacing module takes part in interactions with such connections, while request on already existing connections are still serviced by bean instances of the replaced module. Thus, existing sessions run on the replaced module, but these sessions are destroyed when the replaced module is stopped through step v. All clients of destroyed sessions will experience errors because they hold references to bean instances that do not exist any more. Thus, inconsistencies in the reconfigured application are conceivable. Finally, the replaced module will be undeployed and removed from the mKernel system (steps w and x).

After the replacing module has been started (step o) and newly established connections have been rerouted to the replacing module (step r), both modules, replaced and replacing one, are running concurrently. New sessions run on the replacing, and existing sessions on the replaced module. If it is not desired or not possible to run both modules concurrently, e.g., the steps r and v might be exchanged in the order of the procedure while the rest of the procedure remains unchanged. Thus, after the replacing module has been started (step o), the replaced one will be stopped (step v). Thus, existing sessions running on the replaced module are destroyed and existing connections to the replaced module become invalid. Furthermore, clients requesting a new session or a new connection experience errors because they are still directed to use the replaced module that is not available any more. After being directed to use the replacing module for new sessions and new connections (step r), clients are able to start working with the replacing module. This arrangement of steps guarantees that replaced and replacing module are not running concurrently.

8.1.2 Non-Interrupt

In contrast to Flash, the Non-Interrupt (NI) strategy addresses the Instance Level of the reconfigured application in addition to the Deployment Level and it preserves the consistency of the application. After the replacing module has been deployed and its beans have been configured, it will be started through step o. With the step Newly Established Connection Rerouting (r) clients of the replaced module are directed to use the replacing one for newly created connections. Optionally, existing connections can be rerouted consistently to bean instances of the replacing module with the step Existing Connection Rerouting (s) if the targets of the connections are instances of stateless session beans. In contrast, rerouting a connection whose target is a stateful session bean instance breaks the consistency if the Conversational State of the replaced instance is not transferred to the new and corresponding target instance, because the client-specific state would get lost.
Since no state transfer is performed by the Non-Interrupt strategy, step s should only address existing connections whose targets are stateless session bean instances. Thus, at least existing connections whose targets are instances of stateful session beans still use the replaced module. To ensure application consistency, these existing sessions should finish instead of being destroyed by stopping the replaced module like it is done in the F strategy. As a consequence, the step Module Stopping (v) should be performed when there exists no session any more that is still running on the replaced module. Thus, clients that started their sessions with the replaced module are able to finish them using the replaced module, while clients starting new sessions use the replacing module immediately. After stopping the replaced module, it can be undeployed and removed from the mKernel system (steps w and x).

As long as existing sessions on the replaced module are still running, both, the replaced and replacing module, are concurrently active. At every point in time, either the replaced module, both modules, or the replacing module is operational. Thus, the underlying database might be continuously in use, such that it cannot be reconfigured. Performing no database reconfiguration and no state transfer, a quiescence of the affected part of the application is not required. Consequently, the steps for state transfer (steps j, n, p, and q), database transfer (steps k, l, and m), and quiescence (steps e, f, g, h, i, t, and u) are not necessary and not applied.

8.1.3 Interrupt

The Interrupt (I) strategy is the only one among the four provided strategies that includes a state transfer and the opportunity to reconfigure the database on which the replaced module operates. Therefore, after the replacing module has been deployed and its beans have been configured, the procedures places the replaced module into a quiescent state. Thus, the QuiescenceRegion defined by the step Quiescence Region Declaration (e) consists only of the replaced module together with the beans and corresponding bean instances of the replaced module. The subsequent step Quiescence Region Tracking (f) initializes the tracking of references to bean instances which are part of the QuiescenceRegion and the following step Delay (g) pauses the reconfiguration in order to track the instances. These instances are later on accessible, e.g., in case of stateful session bean instances to extract the Conversational States. Afterwards, the quiescent state is initialized by the step Quiescence Region Blocking (h) and the step Waiting for Quiescence (i) delays the reconfiguration until the quiescent state has been reached. At this point, the replaced module is not actively engaged in servicing requests and the replacing module has not been started yet. Thus, this quiescent state enables the consistent reconfiguration of the modules and the underlying database. Existing instances of enterprise beans and connections among them are handled. Therefore, stateful session bean instances of the replaced module are replaced by corresponding instances of the replacing module, which requires state transfers. The step Conversational State Extraction (j) extracts the states of stateful session bean instances that have been tracked by the steps f and g. Thus, the application state being held by bean instances has been captured, such that the underlying database can be reconfigured (steps k, l, and m). Afterwards, the extracted Conversational States are adapted by step n to match the state type of the replacing stateful session beans. This step may use the reconfigured database for the conversion of the states, which explains the dependency of step n on m. Then, the replacing module is started (step o), such that replacing beans can be instantiated. To replace stateful session bean instances, replacing
instances have to be created and the extracted and converted states have to be injected into them (step \( p \)). After the replacement of stateful session bean instances, clients that hold references to replaced instances have to be provided with references to corresponding replacing instances, such that no invalid references exist within the application. This exchange of references is performed by step \( q \) that addresses existing connections among bean instances whose targets are transferred instances of stateful session beans. Other connections are covered by the subsequent steps \( r \) and \( s \). The first of them reroutes newly established connections, such that clients that depend on the replaced module will use the replacing one if connections are created. The second of them, reroutes existing connections among bean instances whose targets are instances that have not been transferred. These are connections whose targets are stateless session bean instances of the replaced module, and the step \( s \) reroutes them to stateless session bean instances that provide the same functionality and that are part of the replacing module. After the execution of the steps \( q \), \( r \) and \( s \), which provides references to bean instances of the replacing module for clients, no client has or gets any reference to a bean instance of the replaced module. Thus, the changes have been applied completely and the quiescent state can be resolved by step \( t \), such that blocked calls continue execution using references which are provided through steps \( q \), \( r \), or \( s \) and which point to bean instances of the replacing module. After freeing all blocked calls, the quiescence region can be destroyed through step \( u \), and the replaced module that is not engaged in any operation can be stopped and undeployed from the container (steps \( v \) and \( w \)). Finally, the representation of the replaced module can be removed from the \( mKernel \) system (steps \( x \)).

### 8.1.4 Interrupt/Non-Interrupt

The strategy \( Interrupt/Non-Interrupt (I/NI) \) is a mixture of the \( I \) and \( NI \) strategies, because while transferring the replaced module into a quiescent state, like it is done by the \( I \) strategy, the replacing module becomes immediately operational and starts running new sessions comparable to the \( NI \) strategy. Thus, after the replacing module has been deployed and its beans have been configured, the \( QuiescenceRegion \) is defined and the tracking is initialized comparably to the \( I \) strategy (steps \( e \), \( f \) and \( g \)). Nevertheless, before any operations are affected because of blocking method invocations in order to reach a quiescent state, the replacing module is started (step \( o \)) and newly created connections are rerouted to the replacing module (step \( r \)). Thus, clients creating new connections will start sessions and interactions with bean instances of the replacing module. These newly created sessions are not affected by the subsequent steps and they continue execution normally. Thus, the subsequent steps only address already existing bean instances of the replaced module and already existing connections to those instances. Therefore, bean instances of the replaced module should be quiescent, which is initialized by step \( h \). Step \( i \) waits until quiescence has been reached, in order to transfer tracked stateful session bean instances of the replaced module to instances of the replacing module. Likewise to the \( I \) strategy, this transfer is performed by the steps \( j \), \( n \), \( p \), and \( q \). Afterwards, all existing connections to stateful session bean instances have been handled, and connections to existing stateless session bean instances of the replaced module are rerouted by step \( s \). Thus, all connections point to bean instances of the replacing module and the replaced one is not used any more. Therefore, the quiescent part of the application can be released (step \( t \)) and the quiescence region can be destroyed afterwards (step \( u \)). Consequently, clients whose sessions have been transferred resume their operation using instances of the
replacing module. Finally, the replaced module can be stopped, undeployed from the container and removed from the mKernel system through the last three steps.

Comparable to the NI strategy, there is no point in time when it can be guaranteed that both modules, replaced and replacing one, are inactive. Consequently, at least one of them is operational such that the underlying database might be continuously in use, which prevents a database reconfiguration. Thus, the steps $k$, $l$, and $m$ cannot be applied.

Two steps, namely the Adapter and the Analyzer, of the provided set of steps that has been presented in section 6.3.1 are not used by the four procedures that have been discussed in this section. The need for an Adapter can only be identified during the specification of a strategy, i.e., when step executors are assigned to the steps and when the Strategy Creator is occupied with specifying parameter mappings. Then, incompatibilities of step executors become apparent and the integration of them requires an Adapter between the corresponding steps. Thus, during the design of a strategy, it is not known whether or not an Adapter is required. The same holds for the Analyzer step. Only when the requirements of the chosen step executors are known, which is the case during the specification of a concrete strategy, a decision can be made whether or not any analysis of the reconfigured application is required in order to fulfill the requirements of some executors. Thus, these two steps are usually incorporated into a strategy during the specification phase.

8.2 Realization of the Provided Strategies

The designed strategies Flash, Non-Interrupt, Interrupt and Interrupt/Non-Interrupt that are described in section 8.1 have been realized. Thus, step executors for all applied steps and an appropriate plan executor have been implemented, and each strategy has been fully specified. Therefore, the API of the reconfiguration infrastructure has been used, which is described in section 7.1. Specification of a strategy includes the definition of all of its steps, the corresponding step executors, its input and output parameters, mappings among parameters, explicit dependencies among its steps, and of a plan executor. This is described in detail in section 6.3.3. The realization of the strategies focuses on the reconfiguration that is applied within an EJB container and does not address the reconfiguration of databases on which EJB modules are operating. Thus, the steps Database Extraction, Conversion, and Injection are not implemented. Nevertheless, if no database reconfiguration is required, the realized strategies might be used to replace an EJB module with an alternative implementation for this module. The implementations of the executors are generic, i.e., they do not depend on concrete EJB modules that should be replaced. Consequently, any module might be replaced through applying one of these strategies. Furthermore, the implementation of each step executor is not influenced by one of the strategies. Consequently, a step is only implemented by exactly one executor. If a certain step is part of several strategies, the executor for this step is used within a reconfiguration regardless of which of these strategies is actually applied. However, an executor for the step Conversational State Conversion that is capable of adapting all kind of conversational states appropriately cannot be implemented generically, because state adaptations require application-specific knowledge and usually involve application-specific business objects. Therefore, the realized strategies do not consider conversions of conversational states, but such a conversion is not prevented if an Executor Provider provides an appropriate executor for the step Conversational State Conversion that fulfills the needs of concrete state conversions for a concrete application. Thus, from the technological
perspective, state adaptations are feasible, and if it is known, how concrete extracted states should be converted such that they fit into replacing stateful session beans, state conversions might be performed.

Nevertheless, the provided implementation of the executors that realize the strategies imposes restrictions on the EJB module that should be replaced, called the replaced module, and on the replacing EJB module. These restrictions are the following ones:

1. The replacing module must provide implementations for at least those interfaces that are provided by the replaced module and referenced by clients. If an interface is not referenced by any clients, then the interface is not used and it might not be needed any more, such that the replacing module might not be required to provide such an interface. However, providing the same interfaces as the replaced module implies that the replacing module must fulfill the same contracts specified by these interfaces as the replaced module because clients rely on these contracts. Thus, syntactic and semantic aspects of interfaces have to remain unchanged.

2. Each interface identified through restriction 1 must be implemented and provided by exactly one session bean inside both, replaced and replacing modules. However, a bean may provide more than one interface.

3. For all required EJB References of each of the replacing session beans providing at least one of the interfaces identified through restriction 1, there must exist appropriate providers. An appropriate provider is a session bean which is part of any deployed module in the container except of the replaced module. If the provider is part of the replacing module, this restriction must hold recursively. All EJB References of providers not being part of the replacing module must be connected to interfaces, recursively. This restriction ensures that the dependencies of replacing session beans to other beans can be fulfilled.

4. For each session bean of the replaced module, there exists one session bean in the replacing module that provides at least the same interfaces w.r.t. restriction 1.

5. For stateful session beans, the state transfer at instance level is only performed for those fields of the replaced bean for which there exists a matching counterpart in the replacing bean. In this context, two fields are matching if they have the same name and type in both, the replacing and the replaced beans. The state transfer for matching fields is performed regardless of the access modifiers of the fields in the replaced and replacing bean. Because of this restriction, a state transfer corresponds to copying values among matching fields, such that a conversion of extracted states is not necessary.

6. For the replacement of stateful session beans, one instance of the replaced bean is replaced by exactly one instance of the replacing bean, and one replacing instance replaces exactly one replaced instance. Thus, a merging or splitting of conversational states is not necessary. However, the actual problem would be, e.g., to identify the concrete instances of replaced beans whose conversational state should be merged and injected into one instance of the replacing bean. Such problems do not occur with a 1-to-1 relation at the instance level, because for each instance of a replaced bean a completely new instance of the replacing bean can be created.
The implementation of some step executors rely on these restrictions that are imposed on the replaced and replacing EJB module. Furthermore, these restrictions reduce the information that have to be provided by an administrator to perform a concrete reconfiguration.

The first restriction ensures that the replacing module provides at least the same interfaces as the replaced one, and the second restriction ensures that each provided interface of the replaced module can be mapped to exactly one provided identical interface of the replacing module. Thus, it is possible to autonomically identify the client components that depend on a certain interface provided by the replaced module and to reroute their connections to the mapped interface that is provided by the replacing module. Otherwise, if, e.g., the replacing module provides a certain interface more than once, a decision or further information is required which of the provided interfaces should be used by clients that originally have used the interface of the replaced module. Consequently, because of these two restrictions, an executor for rerouting connections among enterprise beans has been implemented, that does not require any decision. It only requires the information provided by the following Analyzer step. This Analyzer step makes use of the restrictions and it identifies for each session bean of the replaced module exactly one session bean of the replacing module. This identified bean replaces one or more corresponding beans of the replaced module. Thus, at the deployment level, there exist $x$-to-1 relations among replaced and replacing beans. This issue is covered implicitly by the second and fourth restrictions that are helpful to identify the replacing bean for a replaced bean. The provided interfaces of session beans are used to find matching replaced and replacing session beans and there is no need that an administrator has to specify matching session beans. Furthermore, the third restriction guarantees that all receptacles of replacing session beans might be connected to appropriate provided interfaces. Thus, it is possible to fulfill all dependencies of replacing session beans and there is no need for further reconfiguration actions, like, e.g., deploying and starting some modules that provide appropriate interfaces to which receptacles of replaced session beans might be connected. Consequently, these restrictions simplify the replacement of an EJB module and reduce the information that have to be provided by an administrator, because several information might be obtained autonomically from the reconfigured application itself.

Though these restrictions are imposed on the reconfigured modules, several scenarios are conceivable for which the implementation of the provided strategies is helpful. E.g., the alternative implementation of the replacing module may eliminate failures in the behavior of the replaced one in order to correct the reconfigured application. Furthermore, the new implementation might be a more efficient one to perfect the quality of service of the application. Additionally, with the replacing module new functionality might be integrated into the application. Therefore, new or enhanced business interfaces might be adopted by the replacing module. Changes, like, e.g., fixing a failure in the behavior of a stateful session bean, do not necessarily require that the conversational state of the bean has to be changed, too. Modifications may only involve the code within a method, but not the instance fields of a stateful session bean. Thus, regarding the structure and the semantics of states, the state of the replacing stateful session bean might correspond completely to the state of the replaced bean. Consequently, for a replacement of such bean instances, state adaptations are not required, such that the step Conversational State Conversion is omitted and an implementation for this step does not have to be provided. Instead, the state transfer is performed as described by the fifth restriction.
To sum up, for certain reconfigurations of applications that fulfill the restrictions, being discussed above, an administrator need not to provide executors and he or she can fully rely on the executors that are implemented by this approach. Furthermore, each of the four provided strategies is fully specified, such that they might be used immediately without requiring further specifications. For a concrete reconfiguration, if one of the four provided strategies should be used, the reconfiguration plan for the chosen strategy requires only the identifiers of the replaced module and of the replacing module type as input. Consequently, with these strategies, the information that have to be provided for a reconfiguration by an administrator is minimized. After indicating both identifiers, the replacement of the module can be performed autonomically, i.e., there is no need for further interaction with an administrator at runtime.

In the appendix on page 94, figure 15 shows a simplified example for a specification of the *Flash* strategy. This example is simplified because all optional parameters of steps and of the strategy are omitted in order to show only the least required specification of the strategy. Arcs between parameters represent parameter mappings, while arcs between steps represent explicit dependencies among steps. E.g., the step *Module Undeployment* depends explicitly on the step *Module Stopping*, because there exist no parameter mapping among these two steps and the replaced module has to be stopped before it might be undeployed. In contrast to the procedure for the *Flash* strategy presented in table 2 on page 81, assuming, that the replaced module contains no enterprise beans that provide any *Simple Environment Entries*, the procedure of the figure does not contain the step *Simple Environment Entry Modification*. Nevertheless, it contains an *Adapter* step, that receives a pair of identifiers and that returns the second part of the pair such that subsequent steps might use this part because they are not able to process pairs of identifiers as inputs. The first part of the pair is the identifier of the replacing module type and the second part is the identifier of an instance of this module type, i.e., the replacing module at the deployment level. Furthermore, it contains an *Analyzer* step, as described above, that identifies mappings of corresponding replaced and replacing beans which are used by the *Newly Established Connection Rerouting* step, such that clients that depend on the replaced bean are directed to use the corresponding replacing bean for newly created connections. The strategy requires two inputs, namely the identifiers of the replacing module type and of the replaced module, and it provides two outputs, namely, the deployment state of the replaced module and information about which EJB references of replacing beans are connected to which provided interfaces.

Comparably to this strategy specification, the strategies *Non-Interrupt*, *Interrupt* and *Interrupt/Non-Interrupt* are also specified completely in this way. The decision for presenting the specification of the *Flash* strategy originates from its simplicity while the other ones are more complex and therefore harder to visualize clearly. Nevertheless, regarding the implementation and its tests most attention has been paid to the *Interrupt* strategy as being the most complex one.

Finally, the realization of the four provided strategies is a proof of concept for the reconfiguration infrastructure, because the realization is based on the API provided by the infrastructure (see section 7.1). It shows, that the concept of strategies, steps, plans, and executors is feasible to reconfigure EJB-based application. Moreover, it demonstrated that a reconfiguration need not to be limited to a certain reconfiguration strategy and that an administrator might be supported in a concrete reconfiguration situation by leveraging reusability of strategies and executors, and by executing a reconfiguration autonomically.
9 Related Work

The approach to seamless reconfiguration, being presented by this work, is inspired by the work of Rosa, Rodrigues and Lopes [RRL07] who present a framework for message-oriented systems that supports a fixed set of reconfiguration strategies. In contrast to their work, this approach is extensible w.r.t. the integration of new strategies. Moreover, the replacement of strategy elements, like, e.g., step executors, is supported which provides additional flexibility. The advantage of a reconfiguration strategy is that strategies abstract from fine-grained reconfiguration tasks, such that for a concrete reconfiguration an administrator is freed from handling these tasks. Moreover, reusability of strategies does not require the creation of a strategy for each reconfiguration need. Thus the work of an administrator is simplified. Finally, the approach of this work addresses a different application area, namely EJB-based EAs, than the approach of Rosa et al.

Since this work focuses on the reconfiguration of EJB-based applications, this section discusses some selected approaches in the EJB and JEE domain and compares them with the approach of this work.

Göbel and Nestler [GN04] extend the EJB specification by adding one more bean type, namely a composite bean. This composite encapsulates runtime adaptation by selecting different sub-components of the composite, e.g., depending on the current system load. Thus, a developer must consider this extension to the EJB standard when implementing a composite component. He or she must provide an enhanced deployment descriptor and code for each composite bean that selects an appropriate sub-component of the composite and forwards invocations to the chosen sub-component. Consequently, only anticipated reconstructions and adaptations are possible that depend on the internals of the composite. These internals are defined at development time. Furthermore, Göbel and Nestler enhanced the implementation of an application server in order to enable the support of composite beans. In contrast, neither the reconfiguration approach presented by this work nor mKernel adjust the EJB container implementation or extend the EJB specification which might affect the work of enterprise bean developers or assemblers. Moreover, a state transfer among stateful session bean instances is addressed by this approach, while nothing is mentioned by Göbel and Nestler about the states of sub-components. Thus, they do not address the issue, how instances of a sub-component that has been selected to service requests might get the relevant state from instances of the previously selected sub-component, which might be required if stateful sub-components are used.

Jarir, David, and Ledoux [JDL02] enhance the EJB container to provide limited reconfiguration. They focus on adapting associations between application components and middleware services. Therefore, calls are intercepted and forwarded to these services before they reach the actual application component instances. Jarir, David, and Ledoux do not consider container-provided services being defined by the EJB standard as middleware services. Instead, middleware services are user-defined and they consider only non-functional aspects that might be imposed on business calls. Thus, the application components and their functional behavior remain unchanged. Consequently, Jarir, David, and Ledoux do not address the reconfiguration of application components, like, e.g., a replacement of such a component.

More possibilities for a reconfiguration are provided by the approach of Rutherford et al. [RAC+02]. Nevertheless, their work is restricted to reconfigurations at the deployment
level. They consider the management of the deployment life cycle of modules and the modification of properties and of connections among enterprise beans. For certain reconfiguration tasks, like, e.g., to reload the properties or bindings of a component, their approach requires support from the affected component that has to provide certain interfaces. These interfaces have to be implemented by the component developer, such that the developer is concerned with reconfiguration aspects during development time. This contradicts the idea of separation of concerns. Additionally, the instance level is not addressed by their approach. Nothing is said about the handling of bean instances, i.e., replacing bean instances together with their possible conversational states is not considered. It seems that instances of components might only be managed and reconfigured through those interfaces which components are required to provide, but which are not described in detail in their paper. Finally, Rutherford et al. recognize that reconfiguration of databases might be part of an application reconfiguration. In contrast, the approach presented by this work maintains the idea of separation of concerns, i.e., developers of applications do not have to consider reconfigurations during development, and it covers the deployment and instance level of EJB-based applications.

In contrast to Rutherford et al., the problem of the state transfer is recognized by Matevska-Meyer, Olliges, and Hasselbring [MOH04]. Their approach to reconfiguration of EJB-based applications is confined to the redeployment of modules. They conclude that stateful session beans are not safe to structural changes because of conversational states that would have to be transferred in order to maintain running sessions throughout a redeployment. Nevertheless, they provide no solution to the state transfer problem and they had no implementation of the redeployment functionality at the time their paper has been written. Since no implementation is provided, it is less helpful to compare their approach to the approach of this work that provides an implementation for replacing stateful session bean instances including transfers of conversational states.

The last approach, which is presented here, originates from a research group at the Peking University [WCMY02, WHS+03]. This group has implemented an own application server, called Peking University Application Server or PKUAS for short, that includes an EJB container. This container enables the reconfiguration of components, connections among components, and of interceptors. Furthermore, it incorporates the necessary technological facilities for updating modules including bean instances and state transfer. During a state transfer, when the attributes of replaced and replacing bean remain unchanged, the attribute values of replaced bean instances are copied to the attributes of replacing instances. If attributes have changed between replaced and replacing bean, either only values for unchanged attributes are transferred or no state transfer is performed, but existing instances of the replaced bean are kept until they are removed by their clients, i.e., until sessions are still running. Thus, in contrast to the approach of this work, an opportunity to convert extracted states before they are injected into replacing instances is not considered. Nevertheless, this research group addresses the deployment and instance level of reconfigured applications such that existing sessions running on existing instances need not to become invalid because of an update. This corresponds to the approach of this work, when an EJB module is replaced using a strategy that maintains application consistency, like, e.g., the Non-Interrupt strategy. However, they do not support any higher-level facilities, like reconfiguration strategies, that may simplify the role of administrators when preparing and performing a reconfiguration.
10 Conclusion and Future Work

With this work, one approach to seamless reconfiguration of EJB-based enterprise applications has been presented. Using this approach, anticipated and unanticipated changes might be applied to running applications while avoiding a shutdown of reconfigured systems. For executing a reconfiguration, changes are performed automatically, such that the degree of automation in software maintenance increases. Changes comprehend parameter and compositional adaptations and this approach considers the objectives of a reconfiguration, namely, that system disruption should be minimized and that consistency of the reconfigured application should be preserved throughout a reconfiguration. Therefore, especially when applying structural changes, it is possible to place an application or only the affected part of an application into a quiescent state. Moreover, if modules should be replaced and stateful session beans are involved, a state transfer from the replaced to the replacing module is feasible, which might be necessary to maintain application consistency. Thus, existing and running instances of enterprise beans are considered, such that this approach addresses the deployment level and the instance level of EJB-based applications. Furthermore, this approach does not prescribe a certain reconfiguration strategy. Several strategies are conceivable. Four strategies are provided by this approach and they are generic and reusable, such that they might be used to replace any EJB module with an alternative implementation for this module. For a concrete reconfiguration, a strategy must be chosen, instantiated and configured. Afterwards, the reconfiguration is executed autonomically. Thus, an administrator is freed from handling fine-grained reconfiguration tasks because they are hidden by strategies. Moreover, the work of an administrator is simplified by the reusability of strategies. For each concrete reconfiguration existing strategies might be applied instead of having always to create a new strategy. Nevertheless, an administrator might create and integrate new strategies, or he or she might adapt existing ones, e.g., by providing executor implementations that replace the already existing executors within a strategy. Thus, the development of a reconfiguration is limited for an administrator to the implementation of custom executors and specification of custom strategies. In this way, specific requirements of reconfiguring certain applications might be fulfilled.

The implementation of this approach is a reconfiguration infrastructure that is used through an API. This API assists an administrator in creating, modifying and using strategies, in preparing and executing a reconfiguration, and in implementing and integrating executors into the infrastructure. The strategies that are provided by this approach might be used through this API, too. Moreover, these strategies have been realized using the reconfiguration infrastructure or precisely the API of the infrastructure in the same way as an administrator would realize his or her strategies. Thus, the realization of the provided strategies and the application of them for replacing EJB modules at runtime can be seen as a proof of concept for the reconfiguration model and infrastructure of this approach. Finally, neither this approach nor the infrastructure as the implementation of this approach requires that developers of applications have to consider reconfiguration aspects during bean development in order enable the reconfiguration of modules and beans. Furthermore, no restrictions beyond those of the EJB specification are imposed on developers. Thus, reconfiguration is transparent to developers who can concentrate completely on the realization of the business logic, which maintains the idea of separation of concerns. Likewise to developers whose beans must conform to the EJB specification, the reconfiguration infrastructure does not violate the EJB standard or performs any operations that are forbidden by the standard.
As future work, it would be desirable that the approach addresses the EJB *Timer Service* during a reconfiguration for two reasons. First, to reach or to maintain a quiescent state of reconfigured parts of an application, method calls invoked by the container on stateless session bean instances because of a *Timer Service* associated to the corresponding beans, which belongs to the reconfigured parts of an application, should be managed. Such calls should be either blocked or the corresponding *Timer Service* should be cancelled, to avoid such calls, during affected parts of an application are becoming or already are quiescent. Second, during a replacement of an EJB module, *Timer Services* associated to replaced beans should be cancelled and an equivalent *Timer Service* might have to be associated to corresponding replacing beans. Thus, when replacing modules, *Timer Services* might also have to be replaced.

Furthermore, first considerations are made to weaken the restrictions of the executors that have been implemented by this approach. These restrictions are described in section 8.2. Instead of replacing only one module with another module, it should be possible to replace \( n \) modules with \( m \) modules. Moreover, for the replacements of beans, it is desirable that \( x \) beans might be replaced by \( y \) beans instead of only \( x\)-to-1 relations. If the restrictions will be changed, investigations need to be done which of the current executors might still be utilized or not. Probably, new executors have to be implemented for certain steps. Additionally to weakening the restrictions, more subtle reconfigurations at the instance level are conceivable, like, e.g., replacing only a subset of stateful session bean instances being part of the replaced module. A decision whether an instance should be replaced or not may rely on the status of its conversational state at the time reconfiguration takes place. Currently, all instances of a stateful session bean, being part of the replaced module, are replaced. More fine-grained reconfigurations at the instance level might require reconfiguration strategies that might be completely new ones or that are mixtures of existing strategies.

Finally, this approach has shown that comprehensive reconfigurations could be applied successfully to EJB-based applications. Regarding the control loop of autonomic systems that is discussed in section 3, the *Execute* stage is covered by this approach. Therefore, the other stages of the control loop might be addressed by future work to make progress towards applications that are able to configure and reconfigure themselves. Such self-(re)configuring systems conform to the vision of *Autonomic Computing* (AC). This approach contributes to the AC vision by reducing the complexity of performing a reconfiguration for any kind of administrator, either a human or an artificial one.
Appendix A: Specification of the *Flash* Strategy

Figure 15: Specification of the *Flash* Strategy
Appendix B: Attached CD

A CD is attached to this work that contains the source code of \textit{mKernel} and of the reconfiguration infrastructure. For details, like, e.g., running an example for a reconfiguration, refer to the instructions on the CD. The following shows the contents of the CD:

\begin{verbatim}
dev
  Folder containing the source code
  |__ mKernel
     Source code of mKernel (this folder also contains the enrichments of the manager done by this work)
  |__ Reconfiguration
     Source code of the Reconfiguration Infrastructure
  |__ api
     |__ src
        Source code of the infrastructure API
        - exec.xml is an sample Executor Descriptor
        - reconfexec-jar_1_2.xsd is the XSD for Executor Descriptors
  |__ common/src
     Source code of classes shared by the infrastructure
  |__ executor/src
     Source code of the provided executor implementations
  |__ lib
     Folder containing the utilized libraries
  |__ server
     Source code of the server side infrastructure
     |__ comp
        Implementation of the server side infrastructure
     |__ res/META-INF
        Folder containing ejb-jar.xml & persistence.xml
     |__ src
        Source code of the server side implementation
  |__ contract/src
     Remote interfaces provided by the server side infrastructure
  |__ test/src
     Contains some test cases for the infrastructure
  |__ tool
     Implementation of the tool that processes Executor Modules
     |__ res/templates/*
        Contains Velocity templates used for bytecode manipulations
     |__ src
        Source Code for the tool
        - build.properties Properties for the Ant script
        - build.xml is the Ant script for the project

glassfish
  Folder containing Glassfish v2.1, b31
sample
  Folder with a sample application for tests of the Interrupt Strategy
\end{verbatim}
References


[Che02b] CHEN, XUEJUN: Extending RMI to Support Dynamic Reconfiguration of Distributed Systems. icdecs, 00:401, 2002.

References


[GHS+04] Ganek, Alan G., Cristiane P. Hilkner, John W. Sweitzer, Brent Miller and Joseph L. Hellerstein: The Response to IT Complexity:
References


References


References


References


Erklärung

“Ich erkläre hiermit gemäß § 27 Abs. 2 APO, dass ich die vorstehende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.”

Bamberg, 07.07.2008