Formal Analysis of Component Adaptation Techniques

by

Kavita Kanetkar

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2002

APPROVED:

Professor George T. Heineman, Thesis Advisor

Professor Elke Rundensteiner, Department Reader

Professor Micha Hofri, Head of Department

Abstract

Increasing demand for commercial software components has led to a development and deployment issue of overcoming differences between the customer requirements and developer specifications for the component. Component Adaptation is one solution to the issue. This thesis focuses on modeling the adaptations to an Enterprise JavaBean™ component using the Z notations and carrying out the adaptations using Active Interfaces adaptation technique. We also formally model the Active Interfaces adaptation technique.

Acknowledgements

This material is based upon work supported by The National Science Foundation under grant number 9733660.

I want to thank Dr. George T. Heineman, my thesis advisor, for his continual support and guidance. I also want to thank Professor Elke Rundensteiner for being the reader for this thesis. I want to thank Professor Kathi Fisler for providing numerous comments on the early draft of this thesis.

I thank my friends Subramanian Raju and Anuja Gokhale for their assistance in LaTeX and their help in presenting this manuscript. I thank all my friends and colleagues for being there whenever I needed.

Last but not the least I dedicate this piece of work to my parents who made all this possible for me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation to adapt software components

1.1.1 Introduction to software components

A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard [1].

Components can be deployed independently, are made up of different classes and can be given run-time resources. Also they can independently interact with other components to form sub-systems of a completely functional system. In other words, components are black box implementations of functionality subject to third-party composition that is conformant with a component model [2]. Treating components as black boxes [3] are as essential as abstraction and information hiding are for better OO programs. The component model defines how the components interact with each other. These aspects of the component technology lead to a strong commercial market for software component technology. Component technology is increasingly gaining importance because the components can be deployed, reused and assembled to form larger systems. Commercial

component technologies such as Microsoft's COM+ [4] and Sun Microsystems' Enterprise JavaBean™ [5] are component models [2]. The component model defines design rules that every software component has to follow.

We have focused our attention on Sun Microsystems' Enterprise JavaBean™ (EJBs), which support a framework for the deployment of heterogeneous components. In addition to interaction between the components to form a larger system, it is also important to support third-party compositions. The components are developed in a developer environment and deployed in the customer environment. Often, however, the requirements specified by the customer to assembling a system using third-party components may not exactly match the requirements known by the component developer. To overcome any differences, one must adapt the component, modify its source code or wrap it.

1.2 Component Adaptation

If an application builder decides to adapt the component, he must understand the complex behavior and functionality of its classes, so any change in either of them will not to break the structure of the system specified by component designer. Commercial components such as EJBs always need integration. While component adaptation occurs during integration, it is distinct from the component customization where the application builder selects from certain pre-defined options. It may not be possible to satisfy the additional requirements, thus the adaptation is difficult. Prior research has identified a list of requirements for component adaptation [6]:

- The adapted component may behave differently but should be used in the same way as the original component would have been used.
- There should not be any additional effort in making the adapted component integrate with the target system and the client-side view is maintained. This property

of adaptation is called *transparency*.

- The original component should not have any knowledge of the adaptation. The component must be open to future adaptations and should not lose its identity as an original component.

While adapting the components all the software engineering phases [7] have to be taken into consideration:

- **The design phase**: The component designer has specified the interfaces, interoperability and relationships between components.
- **The implementation phase**: The component is constructed from the design specifications. This phase includes writing the code, obtaining the source code for all the dependable components, implementations of interfaces and database tables, compilation and linking of the source files.
- **The deployment phase**: The component is deployed into a component infrastructure.

The composition of the components must also be considered during adaptation. In COM [4] for example, the outer aggregate of two components manages the semantic logic of the inner composite component. To adapt the composite [8], the outer component may need to be modified or adapted too. The adaptation should not affect connectors [9] in any way. When adapting as a composite, original components' out interface [1] should not change in terms of declaration or the prototype. Similarly the original components' in-interface should remain unchanged. There are various ways in which we can adapt software components. In this thesis, we have focused on Binary Component Adaptation [10], Active Interfaces [11] and Wrapper [12].

[1] An interface exposed by a component to provide a particular service to its interacting component.

1.3 Available component adaptation techniques

1.3.1 Binary Component Adaptation

Binary component adaptation (BCA) [10] allows the components to be evolved or adapted by the programmer when they are loaded into memory for the first time. As the name suggests, this technique manipulates components in their compiled form and thus are loaded. BCA offers following benefits:

1. BCA requires no source code and operates on the binary, compiled code.
2. The original component and the adapted component are compatible with the original system as before the adaptation.
3. It offers a flexible range of adaptations including renaming a method in a particular class, adding a method to a class, changing or adding parameters to a method, replacing a method, changing the interfaces or changing the sub-typing hierarchies.
4. The processing of the adaptation occurs at load-time.

However there are some downsides of BCA:

1. It introduces load-time overhead because the adaptation is deferred until load-time.
2. The BCA tools must be contextualized for a specific language and operating system (The current implementation only supports JDK version 1.1.3 on Solaris platform).

Figure 1.1 shows the BCA mechanism with a single Java class and the JVM. Only the class file is needed for adaptation and is loaded by the Java Virtual Machine (JVM) class loader. BCA is able to defer the adaptation until load-time because all byte code references are resolved at load-time by the class loader. BCA modifies the JVM to provide a

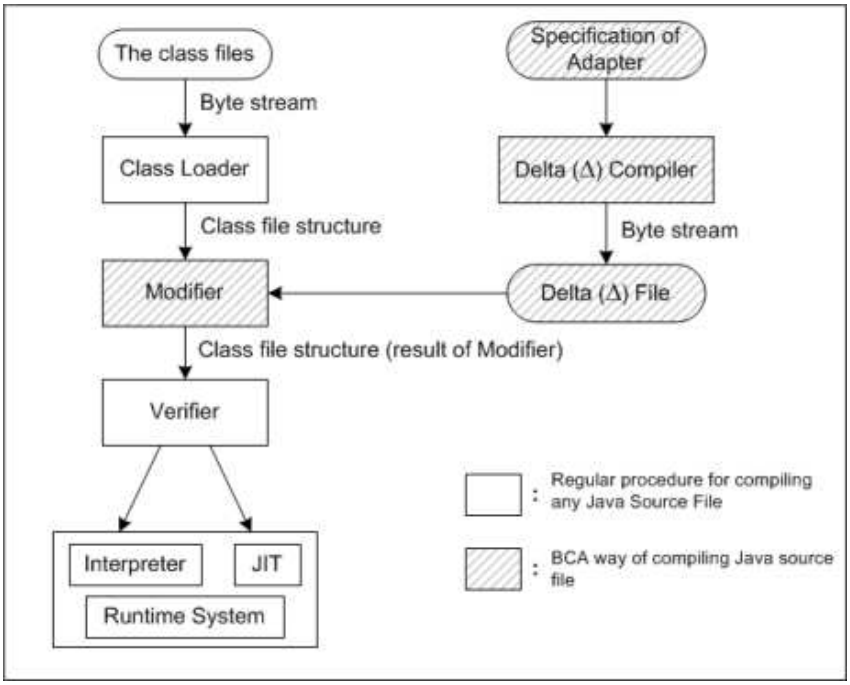

Figure 1.1: Adaptation of Java classes using BCA

customized class loader that loads the bytecode for a component and then invokes a *modifier* to alter the class structure according to the adaptation specifications provided in the *Delta* (Δ) file. The *Delta* file specifies the difference between the existing class structure and functionality and the desired result after the adaptation. The output of the modifier is verified to conform to JVM standards and the class is executed as per the adaptations. This whole procedure creates new classes that form modified application, similar to the adapted application in Active Interfaces [11]. Also the authors of [10] say that BCA has load-time overhead due to online adaptation.

1.3.2 Wrappers

The wrapper technique can be explained by combining design patterns such as decorator, adapter and strategy [12].

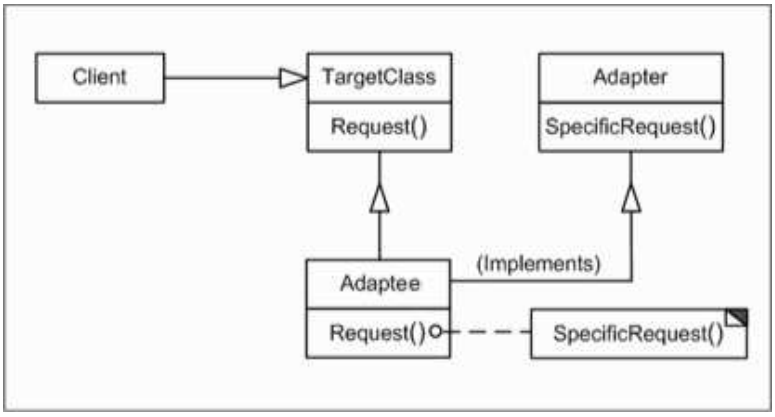

Figure 1.2: Adapter Design Pattern

1.3.2.1 Adapter design pattern

Study of Adapter (or the Wrapper) design pattern [12] shows that the wrapper pattern is used when the interfaces of a class are converted into the interfaces of another class for compatibility. It shows how the interfaces *adapt* to each other. Another important application of this pattern is creating re-usable classes to co-operate with unforeseen classes. The structure of Adapter (or the Wrapper) pattern is shown in Figure 1.2.

The `TargetClass` specifies the domain specifications that client needs to run successfully in a particular domain. For an enterprise application, the interfaces may be the remote interfaces. The `Adapter` defines the interfaces that need adaptations. The `Adaptee` inherits from the `TargetClass` and implements those methods needed by the original `Adapter`. All client requests for the `Request()` are adapted to `SpecificRequest()`. The result of such an adaptation is that the `Adaptee` overrides some of the `TargetClass'` behavior due to inheritance. This feature motivates us to discuss the *override* aspect of adaptation, in Chapter 3. The main drawback of adapter pattern is that they are not transparent to all the clients. There are solutions that support a two-way adapter to solve this problem [12].

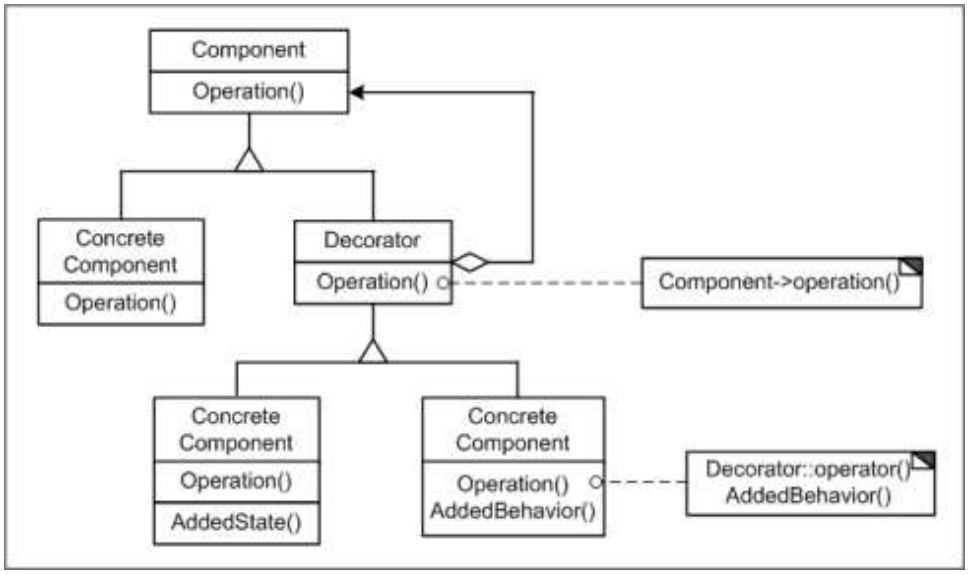

Figure 1.3: Decorator Design Pattern

1.3.2.2 Decorator design pattern

The decorator design patter [12] provides an alternative to inheritance for extending the functionality. The structure of the decorator pattern is shown in Figure 1.3.

The decorator pattern adds responsibilities to the individual objects dynamically. These responsibilities are not rigid and can be added and withdrawn at any time. Also in case of the complex unforeseen classes, the decorator pattern helps in attaching responsibilities to a simple basic class. There is a drawback of applying the decorator pattern: The classes that are to be adapted must support this class hierarchy.

1.3.3 Wrapping Entity Beans with Session Beans

Another pattern of interest applies to the EJB component technology. Chapter 3 covers this component technology in detail. According to this design pattern, the session beans provide a wrapper for underlying entity beans. [13] gives a detailed description of adding a layer of session bean, thus enhancing the performance of the EJB system. Authors of [13] have developed an ATM enterprise application showing the extraction of business

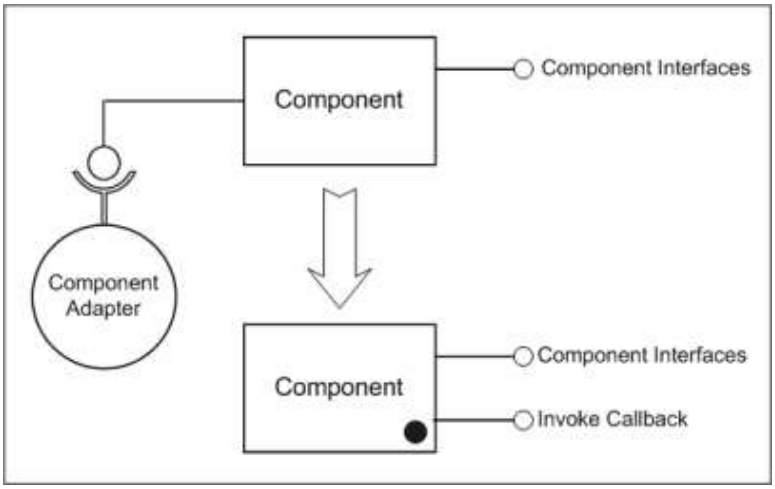

Figure 1.4: Component Adaptation using Active Interfaces

logic from the entity beans. Using such a wrap around an entity bean enhances reuse of entity beans since the session bean represents a façade design pattern [12]. Wrapping the entity beans require certain value objects that are serialiazable Java Beans that contain the client data. The object factories [13] build value objects since they keep track of all the data sources. The underlying entity beans do not contain any application-specific logic. The factories can decide whether to replace entity beans with the data sources. This pattern uses the action objects that correspond to the session logic.

1.3.4 Introduction to Active Interfaces

Active Interfaces [11] can be used to adapt software components to change their behavioral specifications or functionality. Active Interfaces insert additional interfaces that are invoked in the components in addition to the already exposed interfaces of the original component. Callbacks can be inserted for the selected methods in the form of hooks. There are two kinds of hooks: *before* hooks and *after* hooks that map to the *pre* and *post* conditions discussed in Chapter 3. Active Interfaces are language-independent; the working prototype uses Java. Static adaptation is possible since all the classes (those being

adapted and the glue classes) are known prior to loading the system. The methods that are adapted have an associated component adapter that controls the insertion of hooks into those methods. The callback specifies the method name, the phase (*before* or *after*), the glue object, and the new method name along with the parameters. The glue object is created to enable the implementation of the new method specified by the callback. Callbacks are specifically useful while monitoring large systems and to evaluate the *pre* and *post* conditions on the methods. The monitor can be sent all the information, which would get evaluated in the pre-phase of the callback, and depending on the information, actions such as denial, augmentation and overriding of functionality can take place. When a service provided by the method is refused it is called *denial* of functionality. When the values of method parameters are altered, it is known as *augmentation* of functionality, and when the method is overridden, it is known as *overriding* the functionality.

There is a special environment for developing applications with Active Interfaces. It is called Active Interface Development Environment (AIDE) [14]. Currently the environment works for Java programs, but it can be extended to be applicable to any object-oriented programs. There is a front-end tool, which enables the user to select the Java source file and the methods in that class that need to be instrumented. The Figure 3.7 in Chapter 3 shows the window used to instrument the class. After instrumentation, the class is compiled through an AIDE compiler (JavaCC) that generates the new source file with additional *Adaptable* interface and the callbacks for the source methods. This instrumented new source file can be compiled using a normal Java compiler and is JDK version independent unlike the BCA.

The Active Interfaces adaptation technique offers the following advantages:

1. The adapted glue code contains the logic of adaptation, which is separate from the original component code.

2. It can be applied to the applications written in any programming language.

3. All the methods can be associated with callbacks, even those which cannot be externally seen because of the access permissions.

4. The insertions of callbacks take place inside the component. All the other interacting components of the system view it as an original unaltered component.

5. Active Interfaces provide a runtime support for dynamically adding or removing the hooks.

However there are a few drawbacks for this technique:

1. Currently the working environment (AIDE) supports only the components written in Java language.

2. Unlike BCA, Active Interfaces mechanism requires source code for instrumenting the callbacks into the component being adapted.

Due to the advantages of Active Interfaces, and the technology chosen by us (EJB) has all its components written in Java, we choose Active Interface as our adaptation technique.

1.4 Applying adaptation techniques to industrial-scale component models

Enterprise JavaBean™ (EJB) component technology represents one of the most popular component models chosen by many enterprise application developers due to its extensive distributed environment support. We propose a design that allows us to adapt EJB applications. Furthermore we intend to study the benefits of third party run-time adaptations in greater depth by formally specifying the behavior before and after adaptation of the

application, using an appropriate tool. To represent any change in behavior there are various formal notations available such as Unified Modeling Language (UML) [15], IDL [16] and CSL [16]. We are using Z-notations for modeling the adaptations and and the delta (difference in behavior) required to manually change the component behavior. This thesis will focus on building an application involving an entity bean, formally specifying it using Object-Z notations (Z file), packaging and deploying it with an Enterprise Achieve (EAR) file into the EJB container. We then introduce Active Interfaces adaptation mechanism to insert the callbacks. Using the Z' file created, we would infer the methods, which have to be instrumented, compiled, packaged and deployed with bean. Our further aim would be to automate mapping the additions in the Z' file to instrument the callbacks, which is beyond the scope of this thesis and is open for the future work. Chapter 3 describes each step in detail and presents the overall picture of the thesis.

1.5 Conclusion

In this Chapter we have discussed the importance of adapting of commercial components. There are various ways of adapting the components and we argue that Active Interfaces are best suited for our chosen EJB technology. Thus the *adapter* of an EJB application would ideally be a third-party developer who modifies the Z specifications and deploys the Z' into the EJB container that corresponds to the changed bean which must be adapted manually.

Chapter 2

Background

2.1 Active Interfaces

An advantage of Active Interfaces [6] technique is that it does affect information hiding in the component. Using Active Interfaces callbacks does not affect the inter-component dependencies. Active Interfaces either subtract, add or override the component behavior. In Chapter 3, we discuss these adaptations in detail. Due to the advantages of Active Interfaces discussed in Chapter 1, we choose Active Interfaces for adapting our EJB component.

2.2 Specification Formalism

There are many modeling languages such as UML [15], CSL [16], ADL [16] that formally specify the behavior of a component. A previous study shows that the concept used in CSL [16] focused on the interoperability of sub-components of a system. Study of CSL stresses on describing the component adaptation mechanisms. CSL is useful in capturing the inherent complexity of interfaces of the components. Authors of [17] state that formal

notations are increasingly gaining importance in software industry.

2.2.1 Z Notations to specify a component formally

The main purpose of specifying a software component in Z notation is that the specification can be decomposed into schemas to model different scenarios. Z is used to specify the *pre* and *post* conditions for various methods defined by the component. Z notations are also useful to model the static as well as the dynamic aspects of the component. The static aspects are the *pre* and *post* conditions and the system invariants. They also include state information. Dynamic aspects include behavior modeling and component interoperability [18]. Z is a strongly typed specification language, hence every expression or every declaration expressed is associated with a particular type. There are various tools available that allow us to validate the schemas by type matching technique. Z provides a wide range of specification glossary starting from atomic objects to composite objects[19]. It also provides useful set notations such as Cartesian product and ordered pair mapping. We will be using Cartesian product for the modeling of Active Interfaces, where we define an ordered pair (Callback $\times$ SourceMethod $\times$ Phase) for specifying that a particular method is a *before* or an *after* callback to a particular source method. Its ability to check the ordering and to check the type helps us in precise specification of the component interfaces with strongly typed arguments. Schema specifications and bindings are more useful. Bindings are used in behavior specification of a component using operations, where the sub-components of the schema map to the sub-components of a binding. For example, the schema `Container` describes all the container configurations that form the elements of the schema. When `Container` is contained in the schema `EJBServer`, the container configurations can be accessed as the elements of the binding `Container`. Object Z provides logical syntax and quantifiers such as existential and universal. These are very useful while stating the predicates in the schemas. Z notations are also very use-

ful in specifying conditional behavior of the component. We are using the keywords `if, then` and `else` to model the conditional behavior. Z defines relations and functions, which can directly map to the methods and interfaces of the component. In Section 2.2.2, we introduce another effective language to formally specify a software system.

We chose Z notations due to the above stated advantages and formal specifications of the pre/post conditions that map to the *before* and *after* phase of the Active Interfaces callbacks. We can easily specify the Z' file which relates to the adapted behavior of the component. Also due to availability of GUI tools, Z is an ideal choice for specifying the enterprise component.

2.2.2 CadiZ as a tool for specifying a component in Z

We initially selected the CadiZ [20] tool to be our Z editor. CadiZ provides a user-interface to develop Z specifications. It checks the type and correctness of a Z specification, and report type-errors. We used this tool for specifying the software components using troff [20] and LaTeX markup languages. The reference manual specifies the usage of all the schema boxes and the available toolkits. The source directory hierarchy and its usage is shown in the manual. CadiZ is an open-source application that provides all the header files in a particular directory. Apart from being open-source, CadiZ code is portable. These are the advantages of using CadiZ as a tool to model our software. But there are certain features of CadiZ that go against our choice of using this tool for specifying the components for this thesis, they are:

1. The layout printed by "print" command is not always consistent with the actual output.

2. Generic global constraints can be written but not used in proofs.

3. This tool was not robust and its interface was difficult to use.

4. All the symbols for schema and theorems needed to be encoded either in LaTeX or troff and compiled for getting the visual specifications.

5. Typing Z-Specifications in CadiZ becomes tedious because all the toolkits have to be referred for an appropriate symbol and its equivalent markup for compiling the file. It lacks the screen-based interactive inspections of Z predicates or schemas while developing the specifications.

Due to these disadvantages we decided to choose Z/Eves [19] as the tool for specifying components in Z notations.

2.2.3 Z/Eves as the tool for specifying a component in Z

The Z/Eves 2.1 tool allows us to specify any software component in Z notations, analyze it and edit its specifications. It uses the state-of-the-art formal methods [19] techniques with powerful deduction capability. It helps analysis of specifications by checking the syntax, checking the type, allowing schema expansion, aiding the pre/post conditions verification, checking the domain and supporting theorem proving capabilities. The specifications of software component are written in plain text and then mapped to the Z-domain. Z/Eves's easy-to-use GUI helps users inserting the mathematical and logical symbols used in specifications and checking the schema. It has wide range of toolkits (Mathematical and Logical). We are using Mathematical toolkit for certain mathematical operations such as incrementing the account number after creating a new account and for configuration options to limit the maximum server instances to 10. Z/Eves's theorem prover window facilitates heuristics and conditional rewriting of specifications [19]. Z/Eves also extends the prover commands that enable attaching labels to predicate in an axiomatic box. Snapshots of the Z/Eves paragraph window used for mapping the component specifications for an Account EJB component into Z-domain are as shown in Figure 2.1

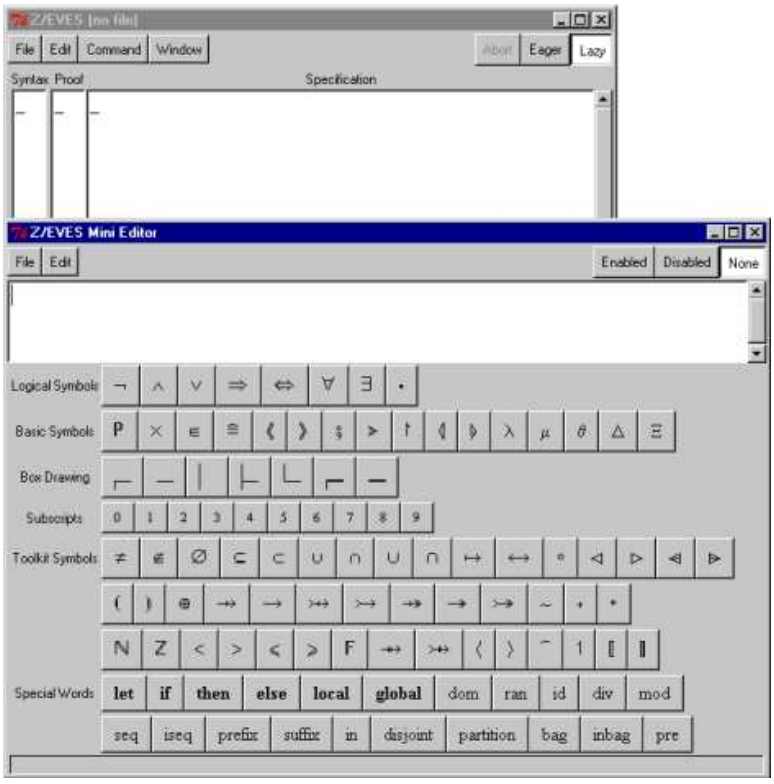

Figure 2.1: Z/Eves paragraph window and toolkit window.

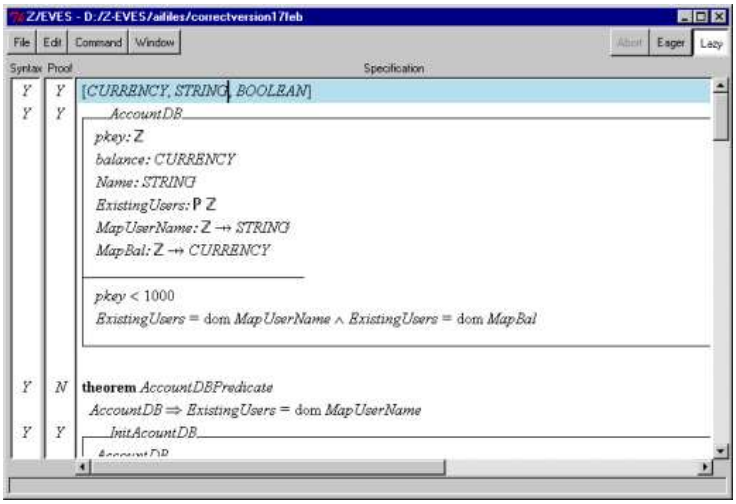

Figure 2.2: Type-checking predicates and theorems in Z

The specifications are written in Z notations with all the paragraphs, schemas and proofs type-checked. The "Y" value shown in Figure 2.2 indicates that the corresponding predicate is logically valid. Chapter 3 gives a complete set of the Z syntax that we use to model the EJB system formally. Both these files are run through the Z/Eves verifier and the type checker to verify that they are valid Z specifications. The output of the type-checker is a result of domain checking and type checking on the Z predicates.

2.3 Industry Scale Component model

2.3.1 Introduction to J2EE

J2EE [21] supports connector architecture [9] where components developed by different vendors provide standard vendor-unspecific connectivity between application servers and enterprise information service. This connector architecture is proposed to be part of version 1.3 of J2EE platform. EJBs are supported by J2EE architecture. J2EE provides multiple resource adapters that are pluggable into an application server. J2EE commands also supports system-level pluggability that includes collection of system level contracts

between server and EIS [21]. J2EE provides various mechanisms such as transaction, security and pooling. Various EJB components are deployed and the servers extract information from them in the form of XML. XML is the most favored mode of data exchange over HTTP or HTTPS protocol [22]. The `application-client.xml`, deployment descriptor and the container configuration files in our example are written in XML [23]. These files are listed in Appendix D.

2.3.2 Motivation for an EJB application

EJB applications are scalable, portable, robust and transactional [21]. EJB is defined by Sun Microsystems, Inc. and is supported by J2EE platform. EJB applications support three tiers: Presentation tier, Business logic tier and Enterprise information system tier. The EJB container forms major part of business logic tier. The difference between the tiers is explained in section 2.3.3. As shown in [5], EJB defines the business logic that operates on enterprise data. An enterprise bean's instances are created and managed at run-time by the container. EJBs can be customized at deploy-time by editing environment entries such as the type of EJBs, the data sources they references etc. Chapter 3 shows a detailed example of an entity bean developed for adaptation.

2.3.3 EJB Three-tier architecture

The EJB three-tier architecture is shown in Figure 2.3. The EJB model contains of the following tiers [24]:

- The Presentation tier : The client applications contain this tier. The clients can access the server-side components using HTML pages, applets, or a standalone application. This tier provides dynamic information from EJB components to the clients.

- Business Logic tier : This tier drives the enterprise application. An EJB server container hosts enterprise beans providing all the lower-level system services. Figure 2.4 shows three types of EJBs. They are:

  - Session beans : The session beans have the same life-cycle as each other [21] The session beans last until the client is connected. Session beans are further divided into following types:

    * Stateless session beans : These beans have no class-level variable declarations [21]. Every method operates on its local parameters. The state of a session bean is not preserved when client re-connects.

    * Stateful session beans: These beans need to have database connection for storing the session information. These beans have instance variables declared in its class definition. The client can manipulate the values of these variables and use them in other method calls.

  - Entity beans: These beans have an underlying data source to store the state of each entity bean. An entity bean corresponds to one row in a database table that is used for persistent storage by the EJB server. By default an entity bean is always stateful. Not only does it uses the bean instance variables across all methods, but also synchronizes these variable-value pairs to the database. This can be done either by the container or the bean itself.

    An entity bean has two ways of managing persistence:

    * Bean Managed Persistence: The EJB developer includes the JDBC and SQL calls [25]. It is very important that the bean is portable across application servers. [21] and hence correct implementation is needed. In our sample implementation we bind the bean to the *Cloudscape* [26] datasource named "JDBC/Accounts". This is a relational database storing

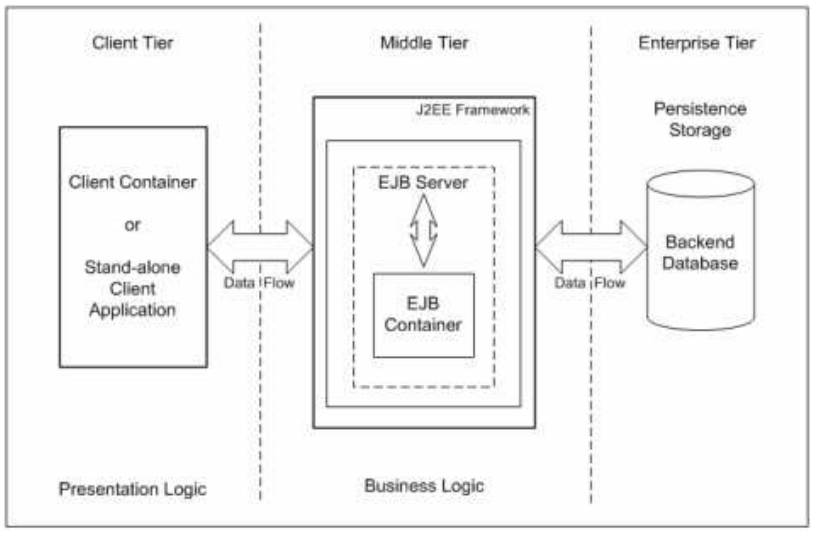

Figure 2.3: EJB Three-tier Architecture.

information per customer for the banking system developed as our example.

* Component Managed Persistence: The entity bean delegates the persistent management responsibilities to the container. The bean implementation does not include any code for dealing with databases. At deploy time, the container generates code required to bind the bean to data sources and implements persistent storage of entity object. At deploy-time, the container maps bean fields to database fields.

– Message-driven beans: These beans are composed of asynchronous messages and synchronous method invocations. Message-driven beans are included in EJB specifications 2.0 [5]. The beans are activated or created when the server receives asynchronous messages.

• The enterprise information system tier: The enterprise information tier deals with different kinds of resources such as databases.

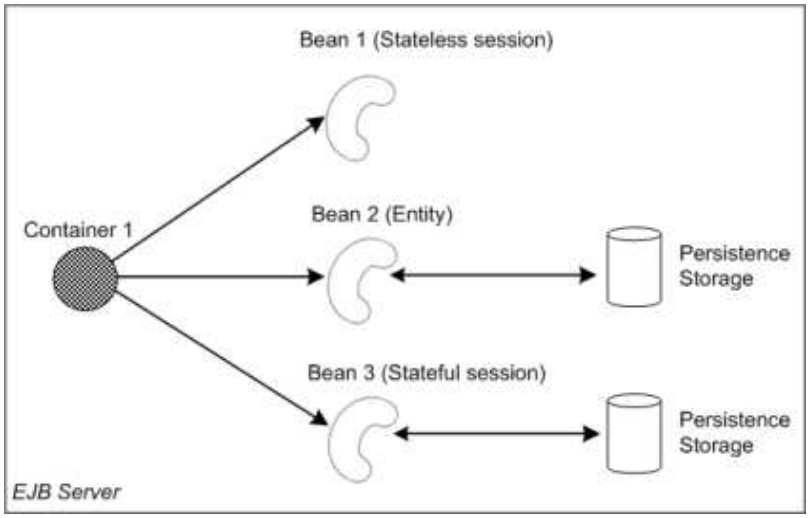

Figure 2.4: Different types of Enterprise beans.

EJB conforms to a component model that supports distributed transactions. The Java Transaction Service (JTS) [21] assists the EJB server to integrate with other EJB servers. The example we discuss in Chapter 3 involves distributed architecture that supports implicit transactions where the entity bean does not specify the transaction demarcation. We will not be using client-demarcated transactions. The *deny* aspect of our adapted EJB maps to the rolling back of transactions. We chose the EJB model because it is based on industry standard protocols and the model is suitable for small applications to large-scale business systems. As discussed in Chapter 4, behavior modeling largely focuses in interoperability of components or sub-components in a large scale software system. The EJB model supports high level of integration and interoperability. An entity bean implements `javax.ejb.EntityBean` interface (As shown in Appendix A).

2.4 Developing Account EJB Application

We choose Iona's iPortal Application Server (ipas) for developing our application. Iona's ipas enables us to develop, deploy, and run component-based enterprise applications. It

provides graphical development tools in which we can assemble Java 2 Platform, Enterprise Edition (J2EE) applications and a standard runtime environment to deploy run applications. We chose the ipas EJB server for following reasons:

1. Iona's iportal application server (ipas 3.0) [24] supports multi-platform applications over Windows NT and Sun Solaris 2.0 onwards.

2. It provides an easy to use GUI to customize the container configurations and make the JDBC connections.

3. It provides a standard run-time environment for developing these applications.

4. It provides a scalable backbone for developing J2EE [21] applications with the in-built Cloudscape JDBC driver loaded.

5. We will focus on EJB server although ipas 3.0 supports various other services such as Java Transaction Architecture (JTA), Java Mail 1.1 [21] etc.

The ipas 3.0 EJB server can contain one or more beans. It can support one or more containers. An EJB container assists an EJB server in managing security, transactions and distributed object communications. In our example we will deal with an entity enterprise bean. In case of an entity bean, it is responsible for persistence of the bean. Figure 2.5 shows working of IONA's ipas3.0 EJB server. As mentioned in Section 3.8, ipas provides a complete set of tools that enable an enterprise application developer to create, deploy and manage applications. The standard format of packaging an enterprise application is an EAR (an Enterprise Archive) file. It contains compiled classes of the bean, interfaces, and deployment descriptor written in XML describing each bean in the container.

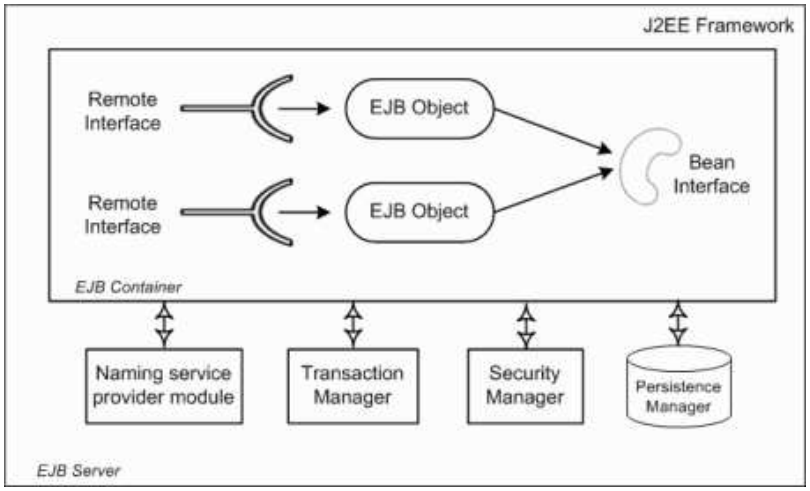

Figure 2.5: EJB Server

2.5 Conclusion

In this Chapter we discussed the purpose of specifying a software component in Z-notations. We compared the tools available to model a system in Z and selected Z/Eves tool to formally specify our EJB component. We focused on the J2EE technology and chose the Iona's ipas 3.0 EJB Server for developing the example. We enumerated some advantages of IONA's ipas application server to host an EJB application.

Chapter 3

Methodology and Design

3.1 Design of thesis in phases

Figure 3.1 shows all the phases in which this thesis is composed. The basic step is to define interfaces, configurations for J2EE environment and those specific to Iona's ipas 3.0 application server. The initial phase is termed as creating/designing of the entity bean. This is when the Z file is first created. The next phase shows the construction of the application by implementing the business methods required by the client. The third stage is called building the application. Iona provides various tools and GUIs to build entity beans. This stage produces an additional EAR file. The fourth phase appends the configuration details with the data source registry. The fifth phase verifies the deployment process of the bean. The sixth and the final phase is developing and testing the client application. Now, phases seven and eight show the Object-Z specifications and the delta applied to the system respectively. This is done manually. The Z' file is packaged and deployed without the server shutting down. Phase nine shows instrumentation of the bean class. Phase ten shows the mapping between the additions or deletions in the Z file to form the Z' file, and the glue code written to change the behavior of the bean according

to the Z' specifications.

3.2 Adaptation choices for adapting an entity EJB

We develop the EJB using the Z specifications which declares the functions, relations, and mappings that design the business logic of the EJB. During adaptation, we deploy the Z' file that describes the behavior additions and the behavior subtractions desired in the EJB. This file contains additional mappings, functions, theorems and relations that describes the *adapted* EJB. To map these changes we manually write the glue code to adapt the EJB. We invesigated the following three choices for adapting the entity bean:

1. As shown in Figure 3.2, there is a centralized adaptation manager bean called `Adapt` bean that is responsible for inserting and deleting the callbacks in all the methods of all the beans. The beans in the container reference this `Adapt` using the bean cross reference tag in their XML bean descriptor. Figure 3.2 shows a bean named `Account Bean`, which is the bean developed according to the Z file. There may be any number of beans in the container. All the beans that are to be *adapted* specify the bean-to-bean cross-reference in the tag called `<EJB-REF>` in the `ejb-jar` XML file (listed in Appendix D). The `Adapt` bean is associated with a component adapter that inserts the callbacks to instrument the specified source methods. The `Adapt` bean calls the `insertCallback()` and `invokeCallback()` methods in the component adapter stating the bean's name, the source method's name, callback method's name and the phase. The idea here is to be able to execute the *before* and *after* callbacks for any method and for any bean in the container.

   There is an advantage to this adaptation decision:

   - The critical `Adapt` bean has to be deployed only once for planned adaptations.

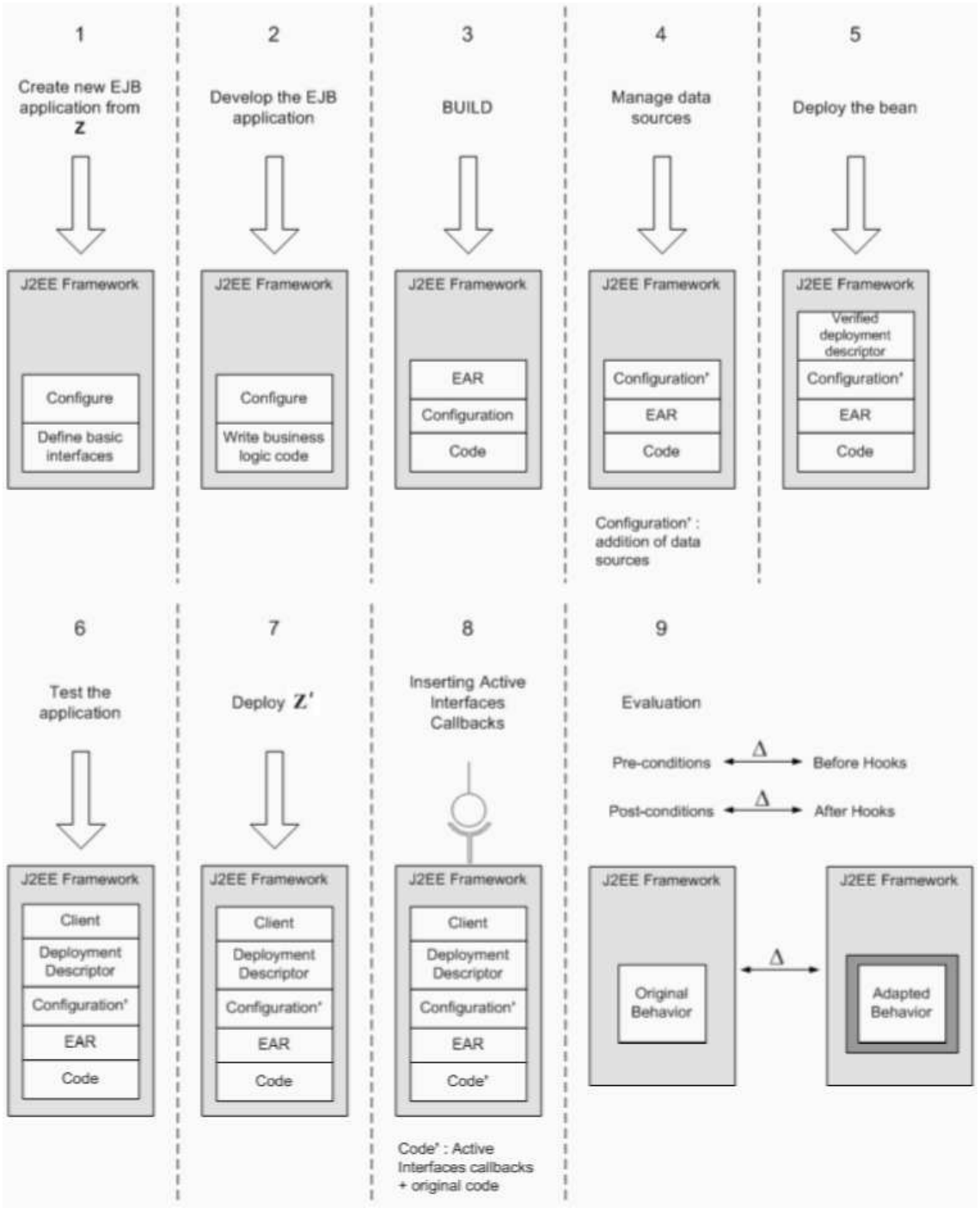

Figure 3.1: Phases for Adapting an EJB

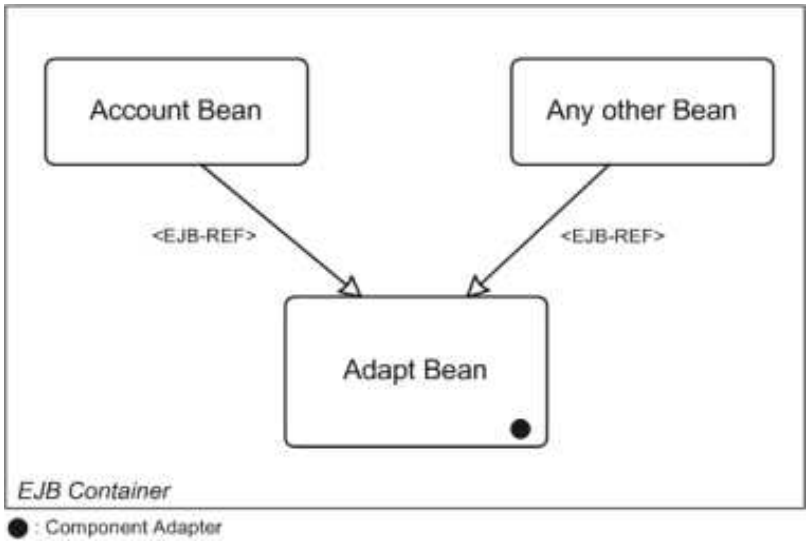

Figure 3.2: Cross referencing the beans for adaptation.

- Specifying the cross-reference between the beans is simple.

Disadvantages of this design are as follows:

- For every change, we have to re-deploy the `Adapt`. Thus, simple additions or deletions of the callbacks forces redeployment.
- It adversely affects the performance of the other beans that depend on `Adapt` bean. There may be several active beans when the *adapter* decides to insert or delete a callback for a specific bean.
- For cross-referencing the beans, container must become involved in the process of adaptation. Specifically, adaptations that change cross-reference values will require the server, supporting the container, to be restarted.

2. As shown in Figure 3.3, there are individual adaptations.

   We extract the centralized adaptation manager out of the container and choose an individual adaptation manager for all the beans in the container. Figure 3.3 shows that each bean references its own adapter. The `Account Bean` shows the domain

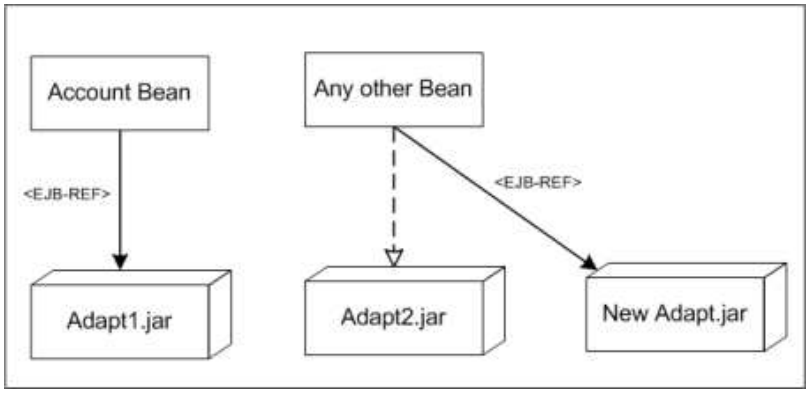

Figure 3.3: Using one adapter bean for each bean in the container.

specific bean that we developed as a working example of an entity EJB. It uses the `<EJB-REF>` tag in the container's `ejb-jar` XML file to state the individual cross-reference between each bean and its own adapter bean. Whenever there is an addition or deletion of the callback for the methods of one bean, its individual adapter bean is undeployed and the new changed adapter is deployed in its place. Each adapter bean (`Adapt1.jar, Adapt2.jar,` and `New Adapt.jar`) is associated with a component adapter that manages the insertions and deletions of the callbacks into the source methods for the bean the adapter is associated with.

An advantage of this scheme is that adaptations for one bean do not affect the other beans in the container. However, there are some disadvantages:

- There has to be a way of automatically undeploying the old bean and deploying the new one.
- The EJB server we have selected to implement our example (ipas 3.0) needs the server to be shut down completely before undeploying the beans. This may affect the active beans in the container.
- It involves the container in the adaptation mechanism.

3. As shown in Figure 3.4 and Figure 3.5 each bean in the container is associated

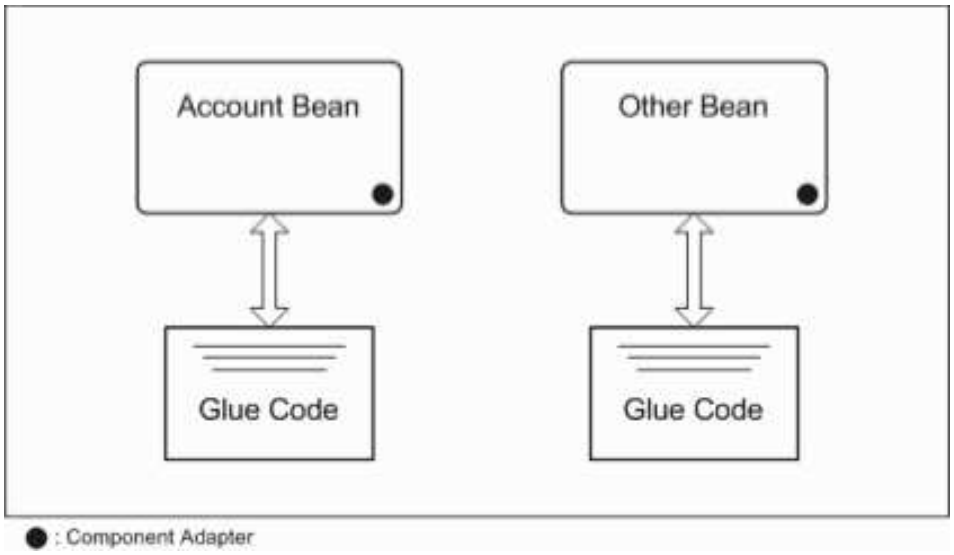

Figure 3.4: Individual glue-code for every bean.

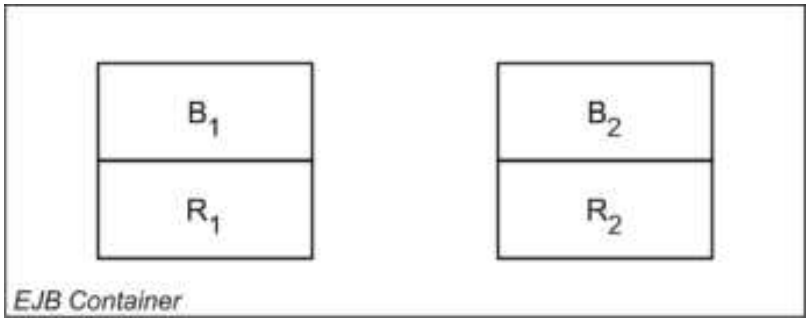

Figure 3.5: Individual registry for every bean.

with a component adapter. Each bean is responsible for registering itself to be adaptable. The meaning of registering is associating a component adapter with itself. The registry code not only associates the component adapter with the bean but also specifies the glue code that goes as the *before* and *after* callback for the source method.

There are following advantages for this adaptation choice:

- The container is not involved in the adaptation mechanism because there are no cross-references between the beans for adaptation purpose.
- The adaptation of the bean `B1` in Figure 3.5 is completely independent of the adaptation of the bean `B2`.

Due to these advantages we chose to implement this adaptation choice.

For our example, the entity bean implements an interface called `Registry` that instantiates an object of class `BeanNameRegistry` in the `glue` package that contains the code for registering the bean be adaptable.

3.3 The semantics for Z

The domain chosen for our application is a banking account system, where the server hosts an entity EJB to serve its clients. The clients perform transactional methods such as creating, deleting a user, accessing the balance, account holder and the overdraft limit. We have modeled the system in parts. The Z file that shows the normal behavior of the bean and the Z' file that shows the *adapted* behavior of the bean are described in Section 3.4. In this Section we will discuss the common terms in Z-notations that we have used to model our system.

3.3.1 Schema Box

The basic block in the Z specifications is the schema box. The schema box contains two parts: a declaration part and a predicate part. The declaration part contains all the type definitions to be used in the predicate part. All the variables have to be declared before they can be used in any predicate. A schema can include another schema. We use the notations such as Δ and Ξ to specify whether the included schema's state is changed by the current schema. The Δ specifies that the state of the schema has changed and the Ξ specifies that the schema is referenced and its state is unchanged.

The schema calculus includes the following operations:

1. S1 $\widehat{=}$ S2 $\wedge$ S3. We will use this for one of the Z-specifications. It is known as the horizontal schema. Thus the schema S1 is a composition of S2 and S3. For example the Z file has a `AddThenDelete` schema defined as composition of `Add` and `Delete` schemas.

2. The schema conjunction is specified by the formula S1 $\wedge$ S2. It is read as the schema S1 "and" schema S2. The schema disjunction is written as S1 $\vee$ S2 and is read as schema S1 "or" schema S2. We use schema conjunctions to strengthen the post conditions after adaptation and we use schema disjunctions to weaken the pre conditions.

3. The concept of component selection is used for binding a part of schema in a predicate. It is denoted by S1.c.

3.3.2 Defining sequences in Z-notations

We define sequences for chaining the Active Interfaces callbacks. Table 3.1 shows the sequence operations and how they are denoted using Z-notations. The `seq` keyword is used to indicate a finite sequence.

Sequence operation	Explanation
s1 ⌢ s2	Sequence concatenation
head s1	First element of the sequence s1
tail s1	All but the head element of the sequence s1
last s1	Last element of the sequence s1
front s1	All but the last element of the sequence s1
s1 in s2	Sequence inclusion known as sequence segment relation

Table 3.1: Sequences in Z

3.3.3 Generic declarations of names

The predefined types are included in declaration box. They represent the basic type definitions. The variables in the schemas, function, relations and predicates that follow the declarations can use them as their types. All the Z files we use include these declarations. For example the Z file specifies a declaration `STRING`. The username that is a variable in the schemas is defined to be of type `STRING`.

3.3.4 Definitions

Table 3.2 gives the Z definitions and their explanation.

Definitions	Explanation
a1 == a2	Abbreviation definition
a ::= b1 | b2|...	Free type definition
a _	Prefix operator
_ a	Postfix operator
_ a _	Infix operator

Table 3.2: Definitions in Z

The EJB specification file shows some of the free type definitions. The word `Boolean` is defined to take either the value "t" for true and "f" for false. It is denoted as: `Boolean ::= t|f.` Similarly the phases in the Active interfaces are denoted as: `Phase ::= before|after.`

3.3.5 Logical operators for the predicate calculus

We use certain logical symbols in the predicates. Table 3.3 shows some of the logical symbols used in our Z specifications and their meaning

Logical Symbol	Explanation
true	Logical true constant
false	Logical false constant
$\neg$ p	Logical negation
p $\wedge$ q	Logical conjunction
p $\vee$ q	Logical disjunction
p $\Rightarrow$ q	Logical implication
p $\Leftrightarrow$ q	Logical equivalence
$\forall$X $\bullet$ q	Universal quantification
$\exists$X $\bullet$ q	Existential quantification

Table 3.3: Logical Symbols in Z

Here p and q denote predicates or the schema variables. There are two types of quantifiers: universal (read as *for all*) and the existential (read as *there exists*).

3.3.6 Defining sets in Z

We use set expressions in all the Z files. Sets are useful notations and they form a part of Mathematical toolkit. Table 3.4 shows the set notations and their explanation.

We use conditional statements for specifying the conditional behavior of an EJB component. The power set symbol states that the set contains all the elements of a particular type. The Cartesian products are used for mapping the tuples. We use set union to include tuples into a relation or a function. The union determines the domain and the range of the function for including the tuple in the function. For example, when we add a user to the existing database, we use set union to update the domain of existing user function (in the Z file).

Set Notations	Explanation
x = y	Equality
$x \neq y$	Inequality
$x \in A$	Set membership
$x \notin A$	Non-membership
$\emptyset$	Empty set
$A \subseteq B$	Set inclusion. Set A is a sub-set of set B
$A \subset B$	Strict set inclusion.
if p **then** x **else** y	Conditional expression
(x,y, ...)	Tuple
A x B x ...	Cartesian product
$\mathbb{P}$A	Power set
$A \cap B$	Set intersection
$A \cup B$	Set union
first x	First element of an ordered pair
second x	Second element of an ordered pair
# A	Number of elements in a set

Table 3.4: Sets in Z

3.3.7 Relations in Z-notations

The symbols we used to define relations in Z are shown in Table 3.5. The mappings are used to define relations (one-to-one, many-to-one, etc.). The keywords `dom` and `ran` specify the domain and the range; used to retrieve a specific datum. For example, an account number can be used to get the balance or the username. The domain restrictions are used to delete a member of a particular set, for example, to remove a particular user from the database. We use compositions to apply more than one relation.

3.3.8 Functions in Z-notations

Table 3.6 shows explanation of the functions we used in the Z specifications. The functions are used to carry out operations such as adding a user to database or deleting a user from the database. Mappings show if the functions are one-to-one, many-to-one or one-to-many. For example, the mapping of an account number to the username is one-to-

Relation	Explanation
$A \leftrightarrow B$	Binary relation
$a \mapsto b$	Maplet
dom R	Domain of a relation
ran R	Range of a relation
$Q \fatsemi R$	Forward relational composition
$Q \circ R$	Backward relational composition (R Q)
$A \lhd R$	Domain restriction
$A \rhd R$	Domain anti-restriction
$A \lhd R$	Range restriction
$A \rhd R$	Range anti-restriction

Table 3.5: Relations in Z

one whereas the mapping of component adapter in AIDE to a component is one-to-many. When we use functions, the input variables are specified by a "?" and output variable by an "!" symbol. For example, in the function for deleting a user from the database, given an account number specifies the input variable is `keydel?`. The schema is shown in Section 3.4. Similarly the schema `credit` shows that the variable `result!` is the output of executing the `credit` method.

Fucntion	Explanation
$A \pfun B$	Partial functions
$A \fun B$	Total functions
$A \pinj B$	Partial injections
$A \inj B$	Total injections
$A \bij B$	Bijections

Table 3.6: Functions in Z

3.4 Designing Z specifications

3.4.1 The Z file

The Z file describes the normal working of the EJB component. It is divided into subsections that model a generic EJB component and our domain knowledge (banking account system), which can be used to implement the bean's implementation class (or the bean's business logic).

- Modeling EJB component in Z: The Z-file for describing the EJB component contains following Schemas: The declaration boxes state the basic definitions such as Java Transaction Architecture, basic data type String, definition of method and Bean instances and their pooling. The bean's state can be one of the following: loaded, passivated or activated. The type of the bean's context is normal for *non-adapted* EJB. The `ExceptionType` is local or remote. The result of any transaction can commit or rollback. The `OperationSet` schema declares the operations of the types: error, rolling back a transaction and switching the bean context. The data-sources, the types of beans, the types of exception and the types of context are predefined. The `ServerInstance` schema declares the server instance to be of type integer and the configuration option for maximum allowed server instances as 10. The `Context` schema declares the type of the context and two exceptions that are thrown. One of the exceptions is thrown if the user does not exist in the database and the other exception is thrown if the amount specified for the transaction is invalid (negative or non-number). The `Configuration` schema and `EJBServer` state the low-level details required to host an EJB application over a distributed network. The schema `ContextLookup` deals with the database sources and their pooling. The `BusinessLogic` schema declares the `HomeInterface, RemoteInterface` and the business methods. It speci-

fies the domain name for the EJB application. Based on the domain the Bean's implementation class is modeled as described in the next Section. We have used an example of banking application. Depending on the domain for which the bean is designed, the model for the bean's implementation class changes accordingly.

$[Operation, NewOperations, String, Currency, int, JTA, CosNaming, Pooling, BeanInstance, CloudscapeRows, Method]$

$BeanState ::= load \mid store \mid activate \mid passivate$

$ContextType ::= Normal$

$BeanResult ::= Exception \mid Success$

$ExceptionType ::= Local \mid Remote$

$Data ::= CloudScape \mid MerrantSQL$

$Res ::= RMIException \mid SuccessfulCompletion$

$Transaction ::= RollBack \mid Commit$

$EJBean ::= StatelessSession \mid StatefulSession \mid Entity$

$Boolean ::= t \mid f$

$ResourceRef ::= JDBCAccount \mid Other$

ServerInstance

$instanceno : \mathbb{Z}$
$MaxInstances : \mathbb{Z}$

$MaxInstances = 10$

OperationSet

$ErrorReport : Operation$
$Adapt : Operation$
$Rollback : Operation$
$SwitchContext : Operation$

Context

$type : ContextType$
$amountEx : Boolean$
$nameEx : Boolean$

Container

$jndi : CosNaming$
$type : EJBean$
$ports : \mathbb{Z}$
$\Delta ServerInstance$
$instanceno' : \mathbb{Z}$
$datastorage : Boolean$

$instanceno' = instanceno + 1$
if $type = Entity$ **then** $datastorage = t$ **else** $datastorage = f$
$ports \geqslant 9000$

Configuration

$Beantype : EJBean$
$BeanRef : ResourceRef$
$DatasourceRef : Data$
$bmp : Boolean$

EJBServer

$Container$
$jndi : CosNaming$
$jta : JTA$
$cc : Configuration$

$cc.bmp = t$
$cc.Beantype = Entity$
$cc.DatasourceRef = Cloudscape$
$cc.ResourceRef = JDBCAccounts$

ContextLookup

$Map : Pooling \longrightarrow ResourceRef$
$result! : Res$
$p? : Pooling$
$d? : ResourceRef$

$d? : JDBCAccounts \Rightarrow Map = Map \cup \{p? \mapsto d?)\} \land result! = SuccessfulCompletion$

$Class ::= LocalException \mid HelperClass \mid AccountBean$

$Domain ::= OtherDomain \mid BankingAccount$

$Interfaces == \{Method\}$

BusinessLogic

$HomeInterfaces : \{Method\}$
$RemoteInterfaces : \{Method\}$
$BusinessMethods : \{Method\}$
$domain : Domain$
$BeanImplementation : Class$
$entityDBMapping : BeanInstances \longleftrightarrow CloudscapeRows$
$create, delete, storeBean, loadBean, activate, passivate : Method$
$credit, debit, getHolder, setHolder, getOverdraftLimit, setOverDraftLimit : Method$

$domain = BankingAccount$
$HomeInterfaces = HomeInterfaces \cup \{create\}$
$\land\ HomeInterfaces = HomeInterfaces \cup \{delete\}$
$\land\ HomeInterfaces = HomeInterfaces \cup \{passivate\}$
$\land\ HomeInterfaces = HomeInterfaces \cup \{storeBean\}$
$\land\ HomeInterfaces = HomeInterfaces \cup \{loadBean\}$
$BusinessMethods = BusinessMethods \cup \{credit\}$
$\land\ BusinessMethods = BusinessMethods \cup \{debit\}$
$\land\ BusinessMethods = BusinessMethods \cup \{setOverDraftLimit\}$
$\land\ BusinessMethods = BusinessMethods \cup \{getOverDraftLimit\}$
$\land\ BusinessMethods = BusinessMethods \cup \{getHolder\}$
$\land\ BusinessMethods = BusinessMethods \cup \{setHolder\}$
$RemoteInterfaces = HomeInterfaces \cup BusinessMethods$
$BeanImplementation = AccountBean$

- Modeling the Account Bean in Z-notations: This is included in the Z file as a model for the banking application. The types such as boolean, string and context type are pre-defined for their use later. They do not form part of any predicates. The persistent storage maps to the `AccountDB` schema. The fields in this schema map to the fields for a database used for the banking system. Every customer should have an account number that is the primary key for the database. This mapping is one-to-one. The primary key is defined to be in the range of 1 to 1000 and of the type integer. There is a predicate called `ExistingUsers` that specifies all the users having a valid account. There are two functions namely `MapUserName` and `MapBalance`, which map name and balance to a specific account number respectively. The schema `InitAccountDB` initializes the system and starting with no users in the database and global primary key to be 0, so that the first user gets an account number 1. The Δ symbol in the schema `AddUser` states that `AccountDB` schema is changed by the `AddUser`. `AddUser` adds the user details to the database if the user did not already exist. The domain and the range of

the relation MapBal and MapUserName is manipulated accordingly. The predicate section of the schema AddUser checks whether or not the user belongs to ExistingUsers. The schema DeleteUser performs reverse operations as those described by the schema AddUser. There are two accessor schemas declared, they are: GetUserName and GetUserBalance. These functions return the username and balance corresponding to a particular account number. The composition schema AddThenDelete restores the state of database, if a user is added or deleted accidentally. The schema Hash increments the global primary key and assigns a new primary key to the new user. It is contained in the schema AddUser. The business methods or the transactions carried out by the client are modeled using the following schemas: debit shows the Δ symbol for the AccountDB schema which states that the information in the schema is changed and the Ξ symbol for the GetUserBalance and GetOverdraftLimit states that these schemas are referenced but not changed. The debit schema deducts the debit amount, checks for the overdraft limit and commits the transaction. If the overdraft limit is exceeded then it assigns *false* to the result of the transaction. Same is the case with credit schema with the difference that the overdraft limit is not checked for the transaction. The predicate sections of the schemas debit and credit check if the user requesting the transaction is a valid user and the transaction amount is positive. The schema debitcommitorrollback throws a remote exception if the conditions for debit fail to satisfy. It commits the transaction to the database if the conditions are satisfied. The schemas setOverdraftLimit and getOverdraftLimit map to the accessor methods for setting and getting the overdraft limit, which is the same for all users. The bean's implementation class can be constructed using the specifications mentioned in this Section. For example, the debit method can be implemented according to the debit schema. The Z file comprises the specifications

for developing the EJB application and the file that describes the bean's implementation class. This Z file is deployed with the bean and describes the application's domain and the business logic for normal functioning of the EJB component.

$$BOOLEAN ::= t \mid f$$

$$ContextType ::= Normal$$

$$Exception ::= BadAmount \mid OverdraftException$$

$$[STRING]$$

Context

$$type : ContextType$$
$$amountEx : BOOLEAN$$
$$nameEx : BOOLEAN$$
$$RemoteException : Exception$$
$$setRollBack : BOOLEAN$$
$$commit : BOOLEAN$$

AccountDB

$$pkey : \mathbb{Z}$$
$$OverdraftLimit : \mathbb{N}$$
$$ExistingUsers : \mathbb{P}\ \mathbb{Z}$$
$$MapUserName : \mathbb{Z} \rightarrow STRING$$
$$MapBal : \mathbb{Z} \rightarrow \mathbb{N}$$

$$pkey < 1000$$
$$ExistingUsers = \mathrm{dom}\ MapUserName \land ExistingUsers = \mathrm{dom}\ MapBal$$

Hash

$$\Delta AccountDB$$
$$newkey! : \mathbb{Z}$$

$$newkey! = pkey + 1$$
$$pkey' = pkey + 1$$

InitAccountDB

$$AccountDB$$

$$ExistingUsers = \varnothing \land pkey = 0$$

AddUser

$\Delta AccountDB$
$\Delta Hash$
$applicant? : STRING$
$startBalance? : \mathbb{N}$

$startBalance? = 0$
$applicant? \notin \operatorname{ran} MapUserName \land newkey \notin ExistingUsers$
$MapUserName' = MapUserName \cup \{(newkey! \mapsto applicant?)\}$
$MapBal' = MapBal \cup \{(newkey! \mapsto startBalance?)\}$

DeleteUser

$\Delta AccountDB$
$keydel? : \mathbb{Z}$

$keydel? \in ExistingUsers$
$MapUserName' = \{keydel?\} \lhd MapUserName$
$MapBal' = \{keydel?\} \lhd MapBal$

GetOverdraftLimit

$\Xi AccountDB$
$limit! : \mathbb{N}$

$limit! = OverdraftLimit$

GetUserName

$\Xi AccountDB$
$pk? : \mathbb{Z}$
$name! : STRING$

$pk? \in ExistingUsers$
$name! = (MapUserName(pk?))$

GetUserBalance

$\Xi AccountDB$
$pk? : \mathbb{Z}$
$bal! : \mathbb{N}$

$pk? \in ExistingUsers$
$bal! = (MapBal(pk?))$

$AddThenDelete \mathrel{\widehat=} AddUser \fatsemi DeleteUser[keydel! \mid keydel?]$

theorem *AddAndDelete*
$AddThenDelete \Rightarrow \Xi AccountDB$

debit

$\Delta AccountDB$
$\Xi GetUserBalance$
$\Xi GetOverdraftLimit$
$pk? : \mathbb{Z}$
$debitAmount? : \mathbb{N}$
$result! : BOOLEAN$
$currentBal, tempBal : \mathbb{N}$

$pk? \in ExistingUsers \land debitAmount? \geqslant 0$
$currentBal = (MapBal(pk?))$
$tempBal = currentBal - debitAmount?$
if $tempBal < -limit!$
then $result! = f$
else $result! = t \land MapBal' = MapBal \cup \{(pk? \mapsto tempBal)\}$

credit

$\Delta AccountDB$
$\Xi GetUserBalance$
$pk? : \mathbb{Z}$
$creditAmount? : \mathbb{N}$
$result! : BOOLEAN$
$currentBal, tempBal : \mathbb{N}$

$pk? \in ExistingUsers \land creditAmount? \geqslant 0$
$currentBal = (MapBal(pk?))$
$tempBal = currentBal + creditAmount?$
$result! = t \land MapBal' = MapBal \cup \{(pk? \mapsto tempBal)\}$

SetOverdraftLimit

$\Delta AccountDB$
$limit? : \mathbb{N}$

$limit? \geqslant 0$
$OverdraftLimit' = limit?$

debitcommitorrollback

$\Delta Context$
$\Xi debit$

$result! = f$
$\Rightarrow (RemoteException = BadAmount$
$\lor RemoteException = OverdraftException) \land setRollBack = t$
$result! = t \Rightarrow commit = t$

creditcommitorrollback

$\Delta Context$
$\Xi credit$

$result! = f \Rightarrow RemoteException = BadAmount \land setRollBack = t$
$result! = t \Rightarrow commit = t$

3.4.2 The Z' file

To change the behavior of the system, we deploy the Z' file into the EJB server and manually adapt the underlying bean implementation. The change in the specifications is the subtraction of the Z file from this Z' file. The sub-sections in the Z' file are only the additions to the original Z file. Z/Eves does not allow these sub-sections to be type-checked without including the schemas and the type declarations in the Z file. To avoid duplicates in the file, we only state the additional schemas that form the Z' file. The Z' file consists of four aspects:

- Modeling the Active Interfaces callback mechanism in Z-notations: This model registers a component to be *adaptable*. The schemas describe the insertion and deletion of the callbacks to the chosen source methods of the component to be adapted. The type declarations such as *BOOLEAN, Actions, ReturnType Phase* are defined for their use in the schemas that follow. The schema `Signature` states a method's signature, that is every method has a name, return-type and input parameters. `Interfaces` is defined to be set of methods. The schema `Component` identifies external, internal and the additional interfaces that a component has. When the component is adapted or associated with a component adapter, the additional interfaces is not a null set. It additionally has the *Adaptable* interface. The `ComponentAdapter` schema inserts the callbacks into the source methods. The relation `insertCallback` states the glue method or the callback, the source method and the phase. If the phase is *before* it adds the callback to the head of the chain of callbacks for that particular method and if the phase is *after*, it adds the callback at the tail. The chain is defined to be sequence of methods. The `ExecuteCallback` executes the chain for a source method starting from its head. The schema `Registry` is responsible for registering a com-

ponent to be *adaptable*. It associates a component adapter with the component. `AdaptSourceMethod` extracts the head of the chain for execution.

$[Interface, Order, String, Classes]$

$BOOLEAN ::= t \mid f$

$Actions ::= augment \mid deny \mid override$

$Values == \{Classes\}$

$ReturnType ::== NormalResult \mid Exception$

$Phase ::= before \mid after$

Signature

$returnType : ReturnType$
$methodName : String$
$parameters : \{Classes\}$
$input : Values$

Method

$MSignature : Signature$

$Interfaces == \{Method\}$

ChainSchema

$Chain : \operatorname{seq} Method$

$\#Chain \geqslant 1$

$chains : \operatorname{seq} Method$

MappingMethod

$MethodNameOf : Method \longleftrightarrow String$

Component

$interface : \mathbb{F}\, Interfaces$
$external : \mathbb{F}\, Interfaces$
$internal : \mathbb{F}\, Interfaces$
$methods : \mathbb{F}\, Method$

$\forall x : external \bullet x \in interface$
$\forall y : internal \bullet y \in interface$
$external \cap internal = \varnothing$
$\exists additionalInterfaces : \mathbb{F}\, Interfaces$
$\bullet\ additionalInterfaces = interface \setminus (external \cup internal)$

ComponentAdapter

$\Delta Component$
$insertCallback : \mathbb{P}\{Method \times Method \times Phase\} \rightarrow chains$
$chainForSrcMethod : chains$
$Adaptable : \mathbb{F}\, Interfaces$

$\forall src : Method;\ callback : Method;\ p : Phase;\ c : chains \bullet$ **if** $\{(src \times callback \times p) \mapsto c\} \in insertCallback$
then
  if $p = before$ **then** $callback$ prefix $chainForSrcMethod \land interface' = interface \cup \{Adaptable\}$ **else**
  $chainForSrcMethod$ suffix $callback \land interface' = interface \cup \{Adaptable\}$
else $interface' = interface \cup \varnothing$

Registry

$\Delta ComponentAdapter$
$associate : Component \pfun ComponentAdapter$
$selectSrcMethods : Method \pfun Phase$
$component? : Component$

$\exists ca : ComponentAdapter \bullet associate = associate \cup \{(component?, ca)\}$

AdaptSourceMethod

$srcMethod? : Method$
$\Xi ComponentAdapter$
$m : Method$

$m = head\ chainForSrcMethod$

ExecuteCallback

$\Delta Signature$
$m? : Method$
$returntype : ReturnType$
$action : Actions$
$newValue : Values$
$newParameters : \{Classes\}$

---

if $returntype = Exception$
then $action = deny$
else if $returntype = NormalResult$
  then $action = augment \land input' = newValue$
  else $parameters' = newParameters$

- Adding the specifications to the Z file that models the Account Bean: The schema `ChangedAccountDB` state that there is an addition needed to the mappings available in the original `AccountDB` and the additional mapping is defined to be `AddedMapOverdraft`. It maps a primary key to the user's overdraft limit. The schema `ChangeOverdraftLimitforUser` includes Δ`ChangedAccountDB` and changes the overdraft limit for a particular user by mapping his account number to the

changed limit value. The result is set to *t* so that there is no remote exception thrown. The schema `AdaptedebitCase1` models the behavior change in the EJB component when we augment the functionality. It changes the transaction amount to the new value. Same is the case with the `AdaptedcreditCase1` schema. The `AdapteddebitCase2` models the *override* feature of adaptation. If the user belongs to the blocked-user list, it returns a local exception. It sets the boolean `result` to be *false*. It uses the schema `BlockUser` to bock a particular user. In our example we have blocked user *Chris* from debiting any amount. This is a special case of weakening the preconditions and it maps to the *override* feature of Active Interfaces that has not been implemented. The schema `AdaptedcreditCase2` rolls back the transaction if it the credit amount exceeds $1000. The schema `Adapteddebitcommitorrollback` changes the context from *Normal* to *Adapted*.

$$ContextType ::= Normal \mid Adapted$$

ChangedAccountDB

$$AccountDB$$
$$AddedMapOverdraft : \mathbb{Z} \mapsto \mathbb{N}$$

ChangeOverdraftLimitforUser

$$\Delta ChangedAccountDB$$
$$pk? : \mathbb{Z}$$
$$newValueForOverdraft? : \mathbb{N}$$
$$result! : BOOLEAN$$

$$AddedMapOverdraft' = AddedMapOverdraft \cup \{(pk? \mapsto newValueForOverdraft?)\}$$
$$result! = t$$

*AdapteddebitCase*1

$$\Delta debit$$
$$newdebitvalue? : \mathbb{N}$$
$$result! : BOOLEAN$$

$$debitAmount?' = newdebitvalue?$$
$$result! = t$$

*AdaptedcreditCase*1

$\Delta credit$
$newcreditvalue? : \mathbb{N}$
$result! : BOOLEAN$

$creditAmount?' = newcreditvalue?$
$result! = t$

BlockAUser

$toBeBlocked? : STRING$
$BlockUserList : \{STRING\}$

$BlockUserList = BlockedUserList \cup \{toBeBlocked?\}$

*AdapteddebitCase*2

$\Xi debit$
$\Delta BlockAUser$
$result! : BOOLEAN$
$\Xi GetUserName$
$pk? : \mathbb{Z}$
$tempUserName : STRING$
$Chris : STRING$

$BlockedUserList' = BlockedUserList \cup \{Chris\}$
$tempUserName = (MapUserName(pk?))$
if $tempUserName \in BlockedUserList$ **then** $result! = f$ **else** $result! = t$

*AdaptedcreditCase*2

$\Delta Context$
$\Xi credit$
$result! : BOOLEAN$

$creditAmount? \geqslant 1000 \Rightarrow result! = t \wedge setRollBack = t$

Adapteddebitcommitorrollback

$\Delta Context$
$\Xi AdapteddebitCase1$
$\Xi AdapteddebitCase2$
$\Xi AdaptedcreditCase1$
$\Xi AdaptedcreditCase2$

$result! = f \Rightarrow type' = Adapted$

- Modeling the changes in the behavior of the component when adapted using the Active Interfaces: There are three schemas, one for each kind of behavioral change in the adapted EJB component. The schema `DenyingFunctionality` relates the Active Interfaces to the bean's implementation specifications. It demonstrates the association of callbacks to the business methods of the EJB component. It states that

it changes the `Registry`, `ComponentAdapter` and `ExecuteCallback`. This schema demonstrates that the `AdapteddebitCase2` denies the functionality and the glue code for the *before* callback is the method corresponding to the `AdapteddebitCase2` schema. It modifies the domain and the range of the relation `insertCallback` and returns an instance of local exception. The schema `AugmentingFunctionality` is similar to the schema `DenyingFunctionality` with the difference that it assigns new values to the method parameters. It states that the method corresponding to the `AdapteddebitCase1` schema is the before callback to the method `debit` in the bean's implementation. The schema `OverridingFunctionality` assigns completely new set of parameters to the method, thus overriding the original source method. The new method that overrides the `debit` method is named as `predebitToOverrideMethod`

DenyingFunctionality

$AccountBean : Component$
$\Delta Registry$
$AdaptSourceMethod'$
$debitMethod : Method$
$\Delta ComponentAdapter$
$AdapteddebitCase2 : Method$
$before, after : Phase$
$\Delta ExecuteCallback$

$\mathrm{dom}\ associate = \mathrm{dom}\ associate \cup \{AccountBean\}$
$srcMethod = debitMethod$
$\mathrm{dom}\ insertCallback = \mathrm{dom}\ insertCallback \cup \{(debitMethod \times AdapteddebitCase2)\}$
$\mathrm{ran}\ insertCallback = \mathrm{ran}\ insertCallback \cup \{before\}$
$action' = deny$
$returntype = Exception$

AugmentingFunctionality

$AccountBean : Component$
$\Delta Registry$
$AdaptSourceMethod'$
$debitMethod : Method$
$\Delta ComponentAdapter$
$AdapteddebitCase1 : Method$
$before, after : Phase$
$\Delta ExecuteCallback$
$changedValue : Values$

$\mathrm{dom}\ associate = \mathrm{dom}\ associate \cup \{AccountBean\}$
$srcMethod = debitMethod$
$\mathrm{dom}\ insertCallback = \mathrm{dom}\ insertCallback \cup \{(debitMethod \times AdapteddebitCase1)\}$
$\mathrm{ran}\ insertCallback = \mathrm{ran}\ insertCallback \cup \{before\}$
$action' = augment$
$returntype = NormalResult$
$newValue' = changedValue$

OverridingFunctionality

$AccountBean : Component$
$\Delta Registry$
$AdaptSourceMethod'$
$predebitToOverrideMethod : Method$
$\Delta ComponentAdapter$
$debitMethod : Method$
$before, after : Phase$
$\Delta ExecuteCallback$
$changedParameters : \{Classes\}$

$\mathrm{dom}\ associate = \mathrm{dom}\ associate \cup \{AccountBean\}$
$srcMethod = debitMethod$
$\mathrm{dom}\ insertCallback = \mathrm{dom}\ insertCallback \cup \{(debit \times predebitToOverrideMethod)\}$
$\mathrm{ran}\ insertCallback = \mathrm{ran}\ insertCallback \cup \{before\}$
$action' = override$
$returntype = NormalResult$
$newParameters' = changedParameters$

- Changes in the EJB model: There is a context type that is added and a schema `SwitchContext` that switches the context from normal to adapted. There is an exception that is added to the set of local exceptions for implementing the *denial* of functionality feature.

$BOOLEAN ::= t \mid f$

$ContextType ::= Normal \mid Adapted$

$Exception ::= BadAmount$
$\quad \mid UserNonExistant$
$\quad \mid LocalAdaptException$
$\quad \mid OverdraftException$

$[STRING]$

SwitchContext

$\Delta Context$
$type? : ContextType$
$type' : ContextType$

$type? = Adapted \Rightarrow type' = Normal$
$type? = Normal \Rightarrow type' = Adapted$

3.5 Choosing an EJB application domain

We develop a relevant EJB application from the Z file that describes the domain specific details (Section 3.4). We develop an entity bean for banking system. Basic requirements of an entity bean application are:

- A jar file containing the interfaces, classes and a deployment descriptor (`cc.xml`) in the Appendix D. The jar file for such an application includes a manifest file describing the beans contained in the `jar` file. The Z file can be included here.

- Home and Remote interfaces: In case of a banking application, the home interface defines standard methods for creating, loading, storing, activating and deleting server-side bean instances. The Remote interface declares the methods accessible to EJB clients. The clients do not get direct access to the bean instances. They get an handle to a remote EJB object known as `EJBObject` that is managed by the EJB container. In the example these methods are `create(), getHolder()` and transactional methods like `debit()` and `credit()`.

- Defining the deployment descriptor: The deployment descriptor (`cc.xml`) is shown in Appendix D. It contains all the information about the EJB modules in the EAR file. For the example, we have only one account entity bean in the container. The EAR contains a jar file that contains all the classes for the bean.

- The Bean implementation: This is the class, which defines all the business methods.

- The client interface and definition of client responsibilities: The client connects to the EJB server, requests transactions and prints out the results of those transactions. Thus client can execute all the methods shown in the file `Account.java` in Appendix C.

The application for our example requires persistent data storage. We are using Cloudscape's JDBC connection wizard [24] to connect the entity bean with database. Application clients access and manipulate the data in this database. The entity bean can be mapped to a row in a database table. Multiple clients can access the same entity bean simultaneously. The transaction manager component provided by EJB server handles this mechanism. The mapping of an entity bean to a database row is called *persistent management*. The implementation of the bean class (`AccountBean.java` listed in Appendix C) specifies that the persistence management be by the bean itself. The source code access the database rows, modifies the data if necessary and commits to the database. The bean managed persistence is specified by the field `<persistence-type>` in the `ejb-jar.xml`, shown in Appendix D.

3.6 Developing server-side elements

- `RemoteInterface`: It enumerates the business methods available to the client. `Account.java` in Appendix C, shows the remote interface for the example EJB. The EJB developer defines it. The `EJBObject` class object is a remote object that implements this remote interface. We define the following methods: [21]

    1. `getEJBHome()`: This method gets the Bean's Home Interface.
    2. `getPrimaryKey()`: This method returns the unique identifier of the bean.

3. `getHandle()`: This method returns an abstract handle to an `EJBObject`'s instance that can be used to (re)establish a reference to an `EJBObject`.
4. `isIdentical()`: This method is used for testing two `EJBObjects`.
5. `remove()`: This method deletes an EJB instance and an associate `EJBObject`.

- `HomeInterface`: `AccountHome.java` in Appendix C defines all the methods needed for the creating, deleting, loading, storing, activating and passivating a bean's instance. Section 3.7 gives detailed description of these methods because they are invoked by EJB clients.
- The bean's business logic implementation : `AccountBean.java` in Appendix C shows the bean's implementation class. It implements the business methods. It also shows a method `instrumentBean()` used to register the bean for adaptation. The adaptation is discussed in Section 3.9.

The account bean is developed according to the Z specifications for EJB application and the domain specific schemas map to the business logic implementation in the bean's implementation class. For example, the schema `credit` maps to the method `credit()` in the class `AccountBean`.

3.7 Developing Application Client

The stand-alone application we develop for our application defines the methods in the `RemoteInterface` of the EJB. For requesting any transaction on the bean, client gets a reference to `HomeInterface` object. The EJB server handles the JNDI [27] that helps publishing this reference. The client then gets a reference to the bean instance by calling appropriate `Home Interface` methods like `create()` method or

`findbyPrimarykey()` method. The client uses the bean instances to invoke the business methods implemented in the server-side bean class. The application client uses following methods:

1. `ejbLoad()`: This method is called by the container to instruct the bean instance to load its state. We will be implementing this, since we are dealing with Bean Managed Persistence (BMP).

2. `ejbStore()`: The example uses this method to synchronize the bean instance with the data source.

3. `ejbActivate()`: This method is invoked by the container to activate the bean instance or to (re)aquire the resources.

4. `ejbPassivate()`: This method is invoked by the container to notify the bean instance that it is about to be removed from memory. The bean returns all the resources in this method.

5. `ejbRemove()`: The container invokes this method to permanently remove bean instance and to free all the held resources.

6. `setEntityContext()`: This method passes the reference of entity context to the bean. In Chapter 3 we will discuss the need for passing the context to callback for enabling the 'deny' feature of the Active Interfaces.

7. `unsetEntityContext()`: The container invokes this method before it destroys the bean and removes the entity context from the bean.

The application client is described by a file called `application-client.jar` shown in Appendix D. It states that the client is a java file known as `AccountClient.java` and it connects to the EJB server for accessing the `Account` entity bean. We develop the client and call transactional methods on the account EJB.

3.8 Setting up an EJB environment

For setting up an environment for the example on Sun Solaris 5.8 using IONA's ipas 3.0 EJB server, an updated version of Java Development Kit (JDK) 1.3 or above is needed. The ipas 3.0 server needs to be installed in an appropriate directory (This is referred to as the `INSTALLDIR`). All the EJB options are to be configured. They include:

- Configuring client/server environment entries: We configure the EJB server to access the `cc.xml` as its container configuration file that describes the beans that are contained.

- Configuring for persistence: We specify that we are using bean managed persistence and the name for the data source is `JDBC/Accounts`.

- Configuring for deployment of the bean. Ipas 3.0 supports a deployment wizard that lets us configure the deployment descriptor and validate it.

  Also to log any errors (relating container, JDBC, server or client) Ipas 3.0 provides an event log.

The example fails under any JVM version below 1.3 because there are some standard jar files required by an EJB application that are not included in JDK versions below JDK 1.3. The `javax.ejb` interface included in `javax` package is shown in Appendix A. The JVM versions below JDK 1.3 do not support it. Appropriate jar files must be included in classpath. Our focus is an entity bean requiring JDBC connection that is a persistence storage. All the configuration variables are set appropriately so that the container and the bean are aware of the transaction demarcations and the persistence storage responsibilities. A platform is chosen for an entity bean server to run. This can be either Solaris 5.8 or Windows NT. We choose Solaris 5.8. Iona ipas 3.0 provides certain set of default configurations like the installation directory, the configuration for embedded

Figure 3.6: iportal.central tool to administer the EJB application.

services, the container configurations and the configurations for iportal administrators. There are certain configurations specific to application; such as options stating whether the host is local stand-alone application or remote (on the network). Apart from these server-side details, we need to develop an application client. Our example defines the methods for the transactions for a banking account system. Also, the client is completely unaware of the adaptation mechanisms. To set up the client-side environment we need to configure the host names and the ports on the EJB server it should connect to.

Figure 3.6 shows the tool to administer the EJB container; and Figure 4.6 in Chapter 4 shows the deployment wizard for choosing the container configuration file and the EAR file.

3.9 Developing EJB Application in AIDE

This Section describes in detail the process of running an EJB application in an AIDE. The bean registers itself to be adaptable.

The class `AccountBeanRegistry` (listed in Appendix C)implements the interface `AdaptationRegistry` in the `glue` package. The bean developer (or the third party "bean adapter") needs to define the methods in the glue code prior to instrumenting the bean. We have defined a function `init()` that takes in an Object as its argument. We need check if that object is adaptable and that a component adapter has been associated with the object. We perform this check by using `instance of Adaptable`. `Adaptable` is an interface defined in `$ADAPT_HOME` directory. It declares a particu-

lar class to have the active interface hooks inserted. If an object passed to the `init()` function of the registry class is found not to be adaptable, we return false with the consequence that adaptation cannot proceed. On the other hand, if the object is found to be adaptable, we insert callbacks. We typecast the generic object registered to be adaptable to the interface `Adaptable` and assign it to a new instance of `Adaptable`. The other thing we ensure is the component adapter that is associated with the bean class is not null. We create a new instance of `ComponentAdapter` class defined in the package adapt in the directory `$ADAPT_HOME`. We have the registered bean typecast to an `Adaptable` interface that has accessor methods such as `getAdapter()` and `setAdapter()`. The next task is to a associate component adapter with the bean. We do this using the accessor method `setAdapter()` because the adaptable object is in fact the bean object typecast to `Adaptable` interface. We have associated a component adapter with the bean. Next we decide on the methods to instrument. At this point we refer to the Z' file. The schemas that are changed map to the glue code to be written. If the system were automated, the changes to Z would be directly mapped to the glue code and the callbacks. The scope of this thesis limits this procedure to be manual. We refer to the methods that need instrumentation. For example, the `AdapteddebitCase2` schema is added in the Z' file that takes user-specific actions. This checking is done in the glue code or the *before* callback to the `debit` method.

We have implemented two business methods in our EJB application, they are: `credit()` and `debit()`. We instrument these methods as per the Z' specifications. We create an array of the arguments and the argument type needed by the source method. The parameter that gets passed to the `credit()` and the `debit()` method is a primitive data type double. Hence we create an array of a single element of a class `Double`. We assign the value to that object depending on the Z' specifications. The `ComponentAdapter` (shown in Appendix D) class defined in the package *adapt* has the following methods that

we use for inserting and invoking the callbacks:

1. `paramList()` : This method returns an array of objects of class `Class`, corresponding to the list of parameters of the original source method. We use this method to build an array of a single element corresponding to the data type double.

2. `getCallbackSet()`: This method accepts a String specifying the source method. This has to be unique. It contains the complete signature of the method. If any of the methods are overloaded, it distinguishes the correct one to instrument by the exact signature. It returns a callback set.

3. `insertHook()`: This method inserts the specified method as a callback. The location where the callback has to be inserted is either `BEFORE` or `AFTER` for the callback before the source method is invoked and after the callback method is invoked. It accepts the name of the source method, its exact signature that directly corresponds to instrumented method's parameters separated by comma and the source object. In our implementation we have a `glue` package. Every bean to be instrumented has a `Glue` class and a `Registry` class. Thus the class named `AccountBeanGlue` has an implementation of pre/post conditions of credit/debit methods of `AccountBean` class. The next parameter to this method is the callback method's name. There may be a chain of callbacks. The method that must be invoked is passed in with its exact signature to the method `insertHook()`.This method creates an array of classes corresponding to the arguments of the callback method passed in as a `String`. It then invokes `insertCallback()` method that does the actual plugging of callbacks at the specific locations.

4. `insertCallbacks()`: This is an overloaded method. We are using the version of this method which takes in the location of callback, source method's name, signature, the glue object, the callback method's name and the array of `Class` objects

corresponding to the parameters of the callback method. This method converts the parameters of the original method of the bean implementation into a `Class` array. It then checks if any previous hooks have been inserted. If they are, it appends the callback to the chain of hooks. This corresponds to the predicate rule of Z to strengthen the post conditions and weaken the preconditions. We logically "OR" the preconditions and logically "AND" the post conditions.

5. `removeCallback()`: This method removes the specified method as callback. It accepts the location (`BEFORE` or `AFTER`), the name of the source method, the glue object and the callback method's name. It checks if the chain of hooks is empty. If the chain is empty, it returns, or goes through the entire chain of hooks until it finds the callback method's name and removes the hook.

6. `invokeCallback()`: This method accepts the location of the callback, the source method's name, the glue object and the list of arguments for the callback method. The return value of this method is of interest. We have designed some cases to deny, augment, or override the functionality of the source method. To deny the functionality, we return an `Exception` class object. We do not throw this exception since there is no corresponding catch in the context. If we return an `Exception` object in the glue code, the instrumented code makes a check of the return value being an instance of an `Exception` class. If it finds to be an instance of `Exception`, it returns a value depending on the return value of the source method. To augment, we return the glue object.

We proceed with inserting the *before* and *after* hooks for `credit()` and `debit()` methods for our account bean EJB. When we specify the glue object with the callback method's name with a parameter as an object of class `AbstractContext` defined in package *adapt* in the `$ADAPT_HOME`, as a callback hook, the component adapter notifies

the bean to have a hook or chain of hooks associated with `debit`/`credit` method(s). Whenever the client invokes `credit()` or `debit()` method(s), the `preDebit`/`preCredit` method(s) are executed before the actual `debit`/`credit` methods are executed. The `postDebit`/`postCredit` methods are executed after the debit/credit methods. The client has an access to `EJBObject` and all the methods defined in the `RemoteInterface`. The client is unaware of any kind of third party adaptations (insertions or removal of callbacks). The interface provided to the client will never show the adaptation methods. Hence the third-party adaptations on the server side of account bean using active interfaces are absolutely *transparent* to the client. If we want to strengthen the post conditions or weaken the pre conditions, we go on chaining the callbacks on the required source method(s).

Use of `AbstractContext` in our adaptation of the business methods: This class forms the base class for call context. Appendix D shows the implementation of this class. It is defined in package `adapt`. It is an abstract class that has a method to return a target object for the context. To be completely generic, the method `getTarget()` returns object of class `Object`. There are other methods like `hasReturnValue()` to determine if the source method has a return value. The `Chain` class in the package `adapt` constructs a chain of callbacks for a particular source method. It defines a method named `getMethod()` that returns a particular method to be invoked. If the component adapter's callbacks are not inserted in correct manner, this method will throw an exception. The method `invokeCallback()` accepts an `AbstractContext` object parameter and returns one of the following:

1. An object of `Override` class (or its subclass) defined in package *adapt*. If this object is returned further calls in the chain are disregarded. Such a behavior of adapted component is termed as *overriding*. The original source method is not executed by active interfaces.

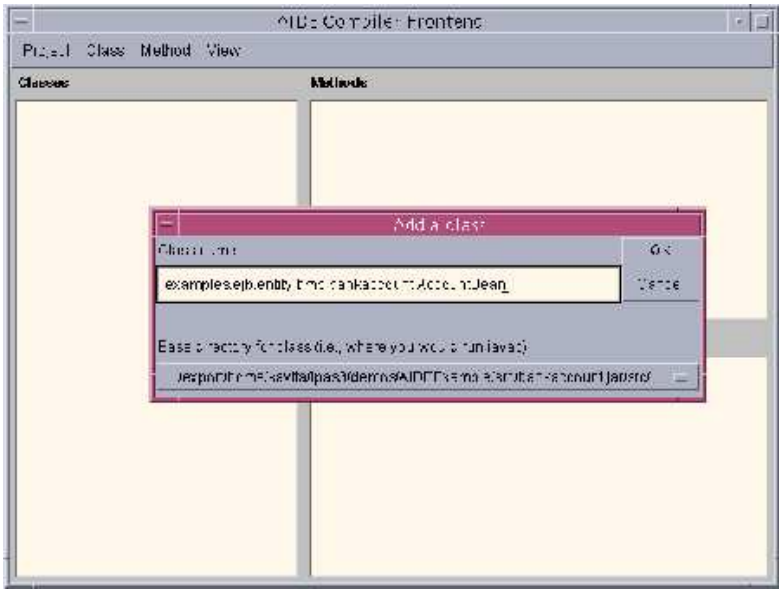

Figure 3.7: AIDE front-end tool.

2. An object of class `Exception` is returned and not thrown. Whenever such object is returned the behavior is called *denial* of functionality behavior or behavior subtraction. Even for this behavior the original method is not invoked and no other callback is invoked.

3. Any other return value other than `Exception` class object or `adapt.Override` class object. The processing continues with the remaining callbacks and the original method is executed by active interfaces. This behavior is called *augment* behavior or behavior addition.

3.10 Running the AIDE front-end tool

Figure 3.7 shows the tool [14] used to instrument the selected methods in an EJB. This tool automatically generates the instrumented Java code in a directory called `probe`. The tool loads up the specified class, lists all the methods and we select the methods that need instrumentation referring to the Z' file. For example, we instrument the methods

`debit()` and `credit()`. The method `preDebit()` (corresponding to the schema `AdapteddebitCase2`) checks the username. checks the username and if the username is found to be *Chris*, it returns an instance of `Exception`. The exception is not thrown because we do not want the adaptations to be *visible* to the client (discussed in Section 3.9). The instrumented code for the bean checks the return type of the executed *before* callback. If it finds the return object to be an instance of `Exception`, it returns and the debit method is never executed. This is an example of *denial* of functionality. If the functionality is to be *augmented*, the value of the method parameters can be changed in the *before* callbacks. The instrumented code assigns the value specified by the callback to the method parameter array. There are two scripts called `adaptClass` and `updateJar` that carry out the entire adaptation process. These files are listed in Appendix B. The script `adaptClass` compiles the instrumented bean's implementation class. The script `updateJar` takes the compiled code from the probe directory and updates the bean's `jar` file along with the `EAR` file that gets deployed. The Z' file is packaged in the `EAR` file using the deployment wizard. Thus the `EAR` file before the adaptation contains the bean's `HomeInterface, RemoteInterface` and bean's implementation class along with all the helper classes and local exceptions the bean needs. The Z file is also packaged with the EJB. After running the scripts, we have a modified `EAR'` file that additionally contains the `adapt.jar` that has the `ComponentAdapter` class, the glue class files and the class files needed for registering the bean to be adaptable. Instead of Z we package the Z' file. The *adapted* EJB is deployed using the deployment wizard of ipas 3.0.

3.11 Conclusion

In this Chapter we used a Z file to contain the description of a bean and the modeling of its bean's implementation class (class that implements the business methods). The Z' file that

is deployed with the *adapted* EJB contains all the original details with the additional descriptions of the desired adaptations. The Z' file thus contains all the schemas in the Z file with additional schemas such as `AdapteddebitCase1, AdapteddebitCase2, AdaptedcreditCase1`, and `ChangeOverdraftLimitForUser`. The Z' file also includes the modeling of Active Interfaces for better understanding of the insertions and deletions of callbacks. The Z file that describes the functionality and the kind of adaptations that Active Interfaces support also gets packaged along with the *adapted* bean. The glue code is written according to the changes in the Z' file and using the bean's implementation class and the `ComponentAdapter` class in the package `adapt`, the callbacks are written and inserted. The front-end tool is run for instrumenting the bean.

Chapter 4

Case Study

Studying the behavior of a software component before and after adaptation is a complex task. We formally model the behavior of the component.

4.1 Behavior Modeling

The requirements of a system are specified in a higher-level (English-like) language. To avoid ambiguity in the specifications, we use formal methods [17] or formal languages to specify the system. We model various behavioral features of the component such as the inter-component interaction and component dependencies [28]. We use formal methods to model the behavior because they are combined with formal reasoning. In case of the Z-notations, we use various tools, such as a theorem prover to check the type and domain of the specifications. The system built according to the Z specifications should be correct and consistent with the requirements. There are some properties of formal specification that determine the behavior of the system. They are:

1. The specifications must be unambiguous: The Z schemas for all the aspects of a component are unique. For example the Z file that specifies the container configu-

ration and the EJB server is unique.

2. The specifications must be consistent: The conditions specified for the behavior of the system must be fulfilled by some specificands [29]. In the example, we define a schema to describe the component interfaces. There must be a component to match those interfaces.

3. The specifications must be complete: All the behavioral aspects about the components must be specified. We have a Z-file that specifies the initial working EJB component. When there are any changes to be incorporated, we deploy the Z' file and make manual changes in the bean implementation. The completeness of Z specifications can be achieved incrementally as the changes to the EJB component cannot be pre-determined.

4. The specifications must support the feature of "inference": The formal methods or the formal language used to prove the behavior of the system must provide some way of validating the specifications. In our example, the theorem prover provided by the Z/Eves tool verifies the Z-file and the Z' file before it is deployed into the EJB server.

Behavior modeling largely focuses on the functionality modeling. Z language provides a model oriented specification [29] for mapping the component model to mathematical constructs. The logical assertions are needed to prove a certain kind of behavior of the component, given a set of constraints. We use axiomatic specifications to define operations and Z schemas to show the temporal logic specifications and concurrent specifications.

4.2 Steps for implementing the behavior change and studying the effect of the change

In Chapter 3 we designed a methodology for combining the Active Interfaces callbacks with an EJB component. This Chapter gives some test cases to reflect the changed behavior of an adapted EJB component. There are three steps for implementing the design:

1. Validating (check the type and domain)the Z-notations for the Z-file and the Z' file.
2. Mapping changes in Z to Active Interfaces callbacks and the glue code.
3. Studying the effect on life-cycle of the EJB objects and the glue code objects.

4.2.1 Validating the Z-notations

We designed some test cases in Z to model the adaptations. We designed a user-specific adaptation. For example, if the user requesting the transactions is named *Chris* we roll-back the transaction and deny the debit transaction to the user. The requirements for such a behavior is specified by making appropriate changes in the Z' file that is deployed in the EJB server. The Z' file shows that the schema for the business methods (debit and credit) remains unchanged; schemas such as `AdaptedcreditCase1` are added. The schema matches the username to *Chris* using the schema `GetUserName`. This check is executed before the debit method is executed. The schema `SwitchContext` is used to switch the context of the bean's instance from *Normal* to *Adapted*. The process of inserting the callbacks is done in the schema `ComponentAdapter`. If the username matches any of the names in the blocked-user list, it returns an instance of local exception and the functionality is denied. The method debit is never executed for the user *Chris*. The additions made to the Z file to get the Z' file are also checked for the type and domain. The schemas that describe the behavior change map to the callbacks. Thus, the Z and Z'

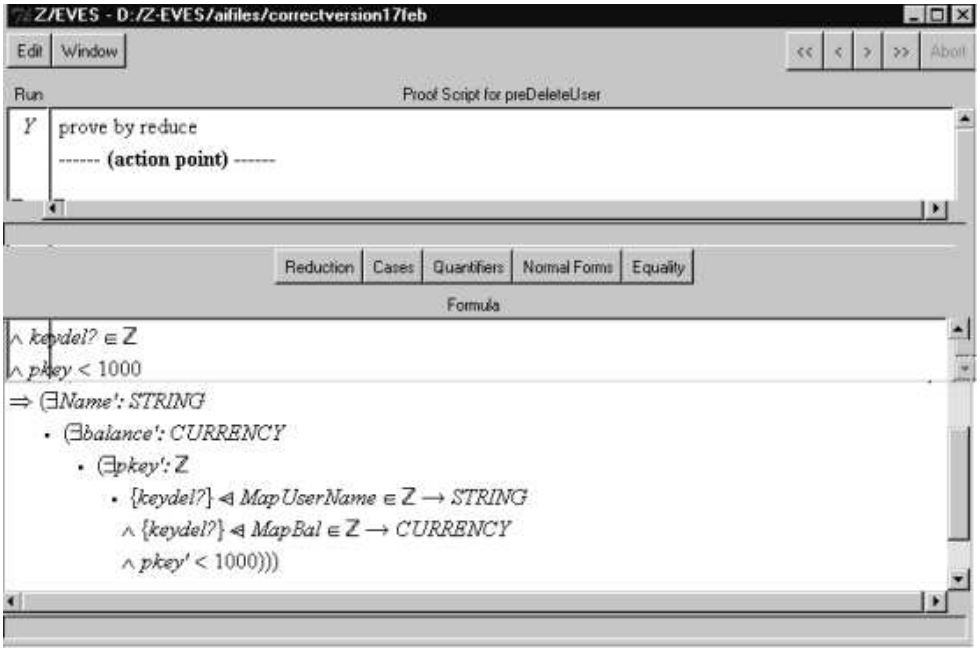

Figure 4.1: Validating the Z-notations using Z/Eves theorem prover

state the behavior of the EJB component before and after inserting the callbacks. We run the Z file and the Z' file through the Z/Eves type checker that points out the type errors. This helps us to model the system correctly. We run the specifications through the theorem prover that evaluates the predicates and assigns the value to the statement. The "Y" value besides the statement and the predicates specify that they are syntactically correct after type checking and domain checking. The Figure 4.1 shows one of the specification statement edited in the theorem prover window of Z/Eves. The lower window performs the formula substitution and type checking for all the variables involved in the statement. The value "Y" indicates the value of the statement. If it encounters any type errors or formula errors it outputs a value of "N" .

We designed a test case for the *Augmenting* the functionality of the component. The value of the parameters are changed for the business methods as designed in `AdapteddebitCase1` schema. The *Overriding* of functionality would change the type of the parameters of the source method or the number of the parameters. We have modeled this case using

the Z-notations, but the implementation of this aspect of adaptation is open for future work. The schema `OverridingFunctionality` assigns a new set of parameters to the method. The debit method may be changed to just work with integers instead of double values. Also the schema `SetOverdraftLimit` in the original Z specifications sets the overdraft amount for all users. If we want to adapt the bean to modify this limit for a specific user, we add the mapping of account number-to-overdraft limit to give a changed design of the schema as shown in `ChangedAccountDB`. The schema `ChangeOverdraftLimitForUser` overrides the original overdraft limit for a specific user. Implementing this aspect of adaptation should not break the inter-component contracts [8]. The adapter is the person who understands the complete working of the components and adapts the component's behavior for suiting his needs without breaking the component-dependencies. There are some classes of errors that the programming-language type checker cannot catch, which can be caught by the Z tool's type checker. For example, the domain checking of the predicates points out any dangling tuples[1] in the database. In our design if the user account is present in the domains of the relations `MapBal` and `MapUsername` only then is the user considered valid.

4.2.2 Observed Features of Active Interfaces

[6] enumerates the critical features of component adaptation. In the example we observed the following features regarding Active Interfaces adaptation technique:

1. BlackBox : The enterprise bean developer only needs to know the interfaces defined in the `RemoteInterface` class. Once the callbacks are inserted the component is unaware of its working.

2. Transparent: The EJB client is completely unaware of the adaptation.

[1] A tuple in a database that does not participate in a natural join is called a dangling tuple.

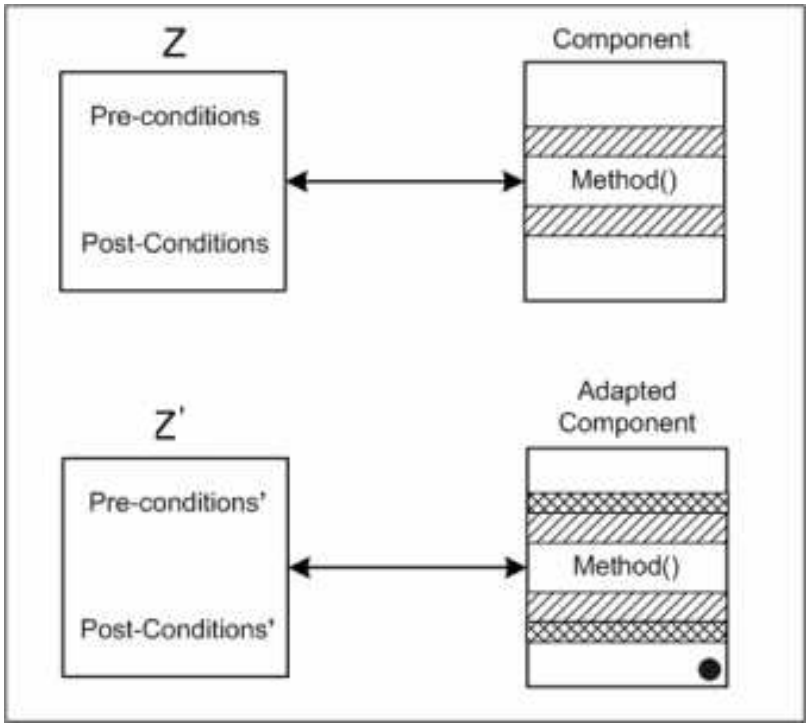

Figure 4.2: Mapping the pre and post conditions specified in Z' to Active Interfaces callbacks.

3. Flexible: A wide range of adaptations is supported by Active Interfaces. We have developed the test cases for all these features.

4. Embedded: The adaptation mechanism is in-built. The bean implementation class registers the bean to be adaptable and inserts Active Interfaces callbacks. The registry class then specifies the glue object that is responsible for implementing the callback methods.

5. Reusable: Anyone can reuse the code written to change the behavior of the EJB component.

Section 4.2.4 shows the results in adapting an EJB. There are two main aspects of component adaptation functional and behavioral change [16] in the component. The functional change in the component includes addition, subtraction and replacement of

the interfaces as a whole or the interface parameters. This change in functionality is visible and can affect the inter component dependencies. The behavioral change in the component is not visible because the signature of all the methods exposed by the component remains unchanged. The *override* feature of Active Interfaces corresponds to the functional change where as the *deny* and *augment* features correspond to the behavioral change. We model the component and the desired behavioral change in the component using Z and the Z' files. Thus we can infer the following equation:

Z' = Z + behavior additions in the Z file and

Z' = Z - behavior subtractions in the Z file

4.2.3 Studying the effect of adaptation on object life-cycles

We chose a component model that largely supports scalability. The Z specifications state that an EJB server can serve multiple clients and there can be multiple server instances for the load-balancing. We evaluate this feature of EJB by invoking multiple clients asking for transactions on same entity bean. In our example the database state needs to be consistent if two or more clients access the same bean. Hence the container blocks and queues all the transaction requests on the entity bean until the current transaction is not committed to the database. Figure 4.3 shows the client queue for processing multiple and simultaneous requests on the same bean.

The Z' file increases in the number of schemas that it contains depending on the delta specifications.

The bean registers itself to be *adaptable*. This registry takes place in the bean's implementation class. There is a performance issue of invoking the registry method in the following EJB methods:

- `ejbCreate()`: If ejbCreate() registers the bean with the Component Adapter, the first time the client invokes any method the bean will be registered for adapta-

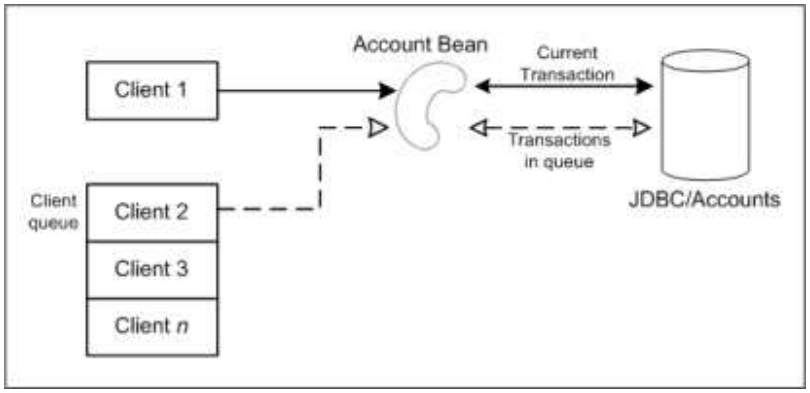

Figure 4.3: Client Queue for serving simultaneous requests on the same entity bean

tion. But if the EJB server evicts the bean from the memory for pooling or EJB container passivates it, all the subsequent calls to the business methods will go through the `ejbActivate()`. Thus, the bean invokes only single instance of `ComponentAdapter` at create time. The component adapter object does not hook on to the bean's instance because it is not its member object. Thus the life-cycle of component adapter ends when the bean is passivated.

- `ejbActivate()`: If the registry for the bean takes place in ejbActivate(), even if the container passivates the bean every time a new instance of `ComponentAdapter` gets instantiated for inserting the callbacks. It may be possible that during a session, the bean's instance is activated multiple times. In such a case, we have a flag (a boolean that is native and serializable) that is *false* for every `ejbActivate()` after the first one. Thus we ensure that not more than one `ComponentAdapter` gets associated with the bean.

The instance diagram for all the objects involved in the EJB application is as shown in Figure 4.4.

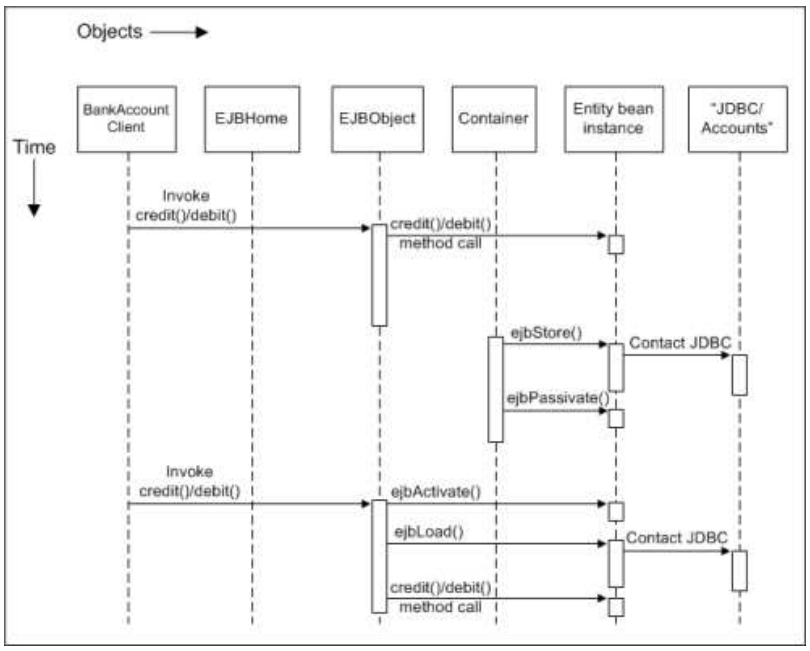

Figure 4.4: Life-cycle Diagram of the objects involved in the example EJB application

4.2.4 Classes and the Nature of Adaptations supported

There are certain classes of adaptations that the idea presented cannot fulfill. We do not support any changes to the basic assumptions that are central to the component's functioning. For example, an adapted component cannot change the component's code to execute in a differennt Operating System. Also, adapted component must be in the same programming language. Though we stated as one of the advantages of using Active Interfaces is that it is language independent, it means that the original component and the *adapted* component are developed using the same language.

4.2.5 Implementation Results

The steps for running the example and the corresponding results are as follows:

- Step 1. Run the scripts for setting up the environment variables and setting up the `$ADAPT_HOME` and `$AIDE_HOME`.

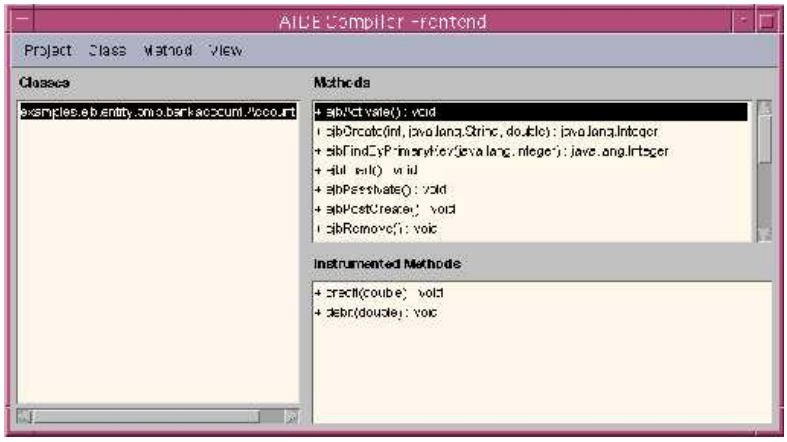

Figure 4.5: AIDE front-end tool to instrument business methods in account EJB.

- Step 2. Build the EJB system.

  Results of Step 2:

```
athena:/export/home/kavita/ipas3/demos/AIDEExample> java iportal.build
IONA iPortal Application Server - Version 3.0 Standard Edition Update 1 [20010531-2021]
Copyright (c) IONA Technologies plc., 1999-2002. All Rights Reserved.
Java 1.3.0 on SunOS 5.8 (sparc)

Building ejb-jar bankaccount.jar
Building ear AIDEExample.ear
Validating bankaccount.jar

BUILD SUCCESSFUL
```

- Step 3. Run the front end tool to select the methods to instrument: The command is: `java aide.frontend.Frontend`

  Results of Step 3:

```
Java Active Interface Pre-processor Version 1.1:  Java program parsed successfully.
Compiler exited with code: 0
Wrote output to probe/examples/ejb/entity/bmp/bankaccount/AccountBean.java
```

  The probe directory is created with the same structure as the EJB directory. The `src` directory contains the Java source files and the `tmp` contains the class files and

the `jar` file. An example of instrumented code is shown in Appendix C. The normal `debit()` method in the `AccountBean.java`, after instrumentation, gets transformed to the one shown in Appendix C. The front-end tool instruments only the methods selected from the set of methods of the class. We deal with the business methods such as `credit()` and `debit()` for instrumentation.

- Step 4. Run the script adaptClass: `adaptClass bankaccount.jar AccountBean.java`
  Results of Step 4:

```
Making probe directory
Compiling examples/ejb/entity/bmp/bankaccount/AccountBean.java
Removing dead code
Final Compilation
Done.
```

  This script compiles the instrumented code using the JavaCC compiler in the `$ADAPT_HOME` directory. It checks the environment entries for the compilation process and reports errors, if any. The first parameter to the script is the `jar` file containing the bean's classes. The second parameter contains the Java source file that is instrumented. The file `AccountBean.java` is the bean's implementation file. It sets the appropriate classpath for the execution of the compiled instrumented class instead of the bean's original class.

- Step 5. Writing and compiling the glue code according to the Z' specification. The `AccountBeanGlue.java` is shown in Appendix C. The Z' specification is shown in Chapter 3. The glue code implementa the before and the after callbacks for the instrumented methods.
  Result of Step 5:

```
Successful compilation of the glue code.
```

- Step 6. Run the script updateJar `updateJar bankaccount.jar`

  Results of step 6:

```
Updating class files in target bean...
adding: examples/ejb/entity/bmp/bankaccount/AccountBean.class \\
(in = 6549) (out= 3229)(deflated 50%)
Updating glue files in target bean...
adding: glue/AccountBeanGlue.class(in = 1347) (out= 723)(deflated 46%)
adding: glue/examples/ejb/entity/bmp/bankaccount/AccountBeanRegistry.class\\
(in = 1747) (out= 997)(deflated 42%)
adding: glue/AdaptationRegistry.class(in = 161) (out= 123)(deflated 23%)
copying examples/ejb/entity/bmp/bankaccount/AccountBean.class into the \\
 /classes/ directory.
bankaccount.jar Updated:
updating AIDEExample.ear file
adding: bankaccount.jar(in = 44788) (out= 17584)\\
(deflated 60%)
done.
```

  This script updates the `jar` file that contains the bean's classes to include the compiled instrumented file and the glue classes. It also updates the EAR file that is deployed.

- Step 7. Start IONA's ipas 3.0 EJB server: `java iportal.server`

  Results of Step 7:

```
IONA iPortal Application Server - Version 3.0 Standard Edition Update 1 [20010531-2021]
Copyright (c) IONA Technologies plc., 1999-2002. All Rights Reserved.
Java 1.3.0 on SunOS 5.8 (sparc)

Starting Embedded Service: Locator...done
Starting Embedded Service: NameService...done
Starting Embedded Service: iPortal Administrator Management Service...done
iPortal Administrator's Web Adaptor available on "http://localhost:3085"
Starting HTTP Listener on port 9000 ... done
Connecting to iPortal Administrator's Management Server...done
Loading: AIDEExample  Ear from repository...
```

Figure 4.6: IONA's Deployment wizard.

```
Deploying new EJB with name Account
done
Ready...
```

- Step 8. deploy the EJB application `java iportal.deploy`

```
IONA iPortal Application Server - Version 3.0 Standard Edition Update 1 [20010531-2021]
Copyright (c) IONA Technologies plc., 1999-2002. All Rights Reserved.
Java 1.3.0 on SunOS 5.8 (sparc)

Reading the ear file...Done
Deploying application /export/home/kavita/ipas3/demos/AIDEExample/AIDEExample.ear...
Trying to validate a directory
Validating bankaccount.jar
Trying to contact iPAS server...Done
Looking up remote deployment interface...Done
Using application protocol ear:...Done
Installing into new container called AIDEExample...Done
Applying container configuration to AIDEExampls...Done
Done
```

The server-side result of deploying:

```
Deploying new EJB with name Account
```

- Step 9. Running the client (case 1)

```
java iportal.client -m bankaccountclient.jar -c cc.xml -e
AIDEExample.ear
```

The server-side results:

```
in ejbActivate()
Proceed to inserting callback
in AccountBeanRegistry::INIT()
Before inserting callback in AccountBeanRegistry
glue object created...
after inserting callback in AccountBeanRegistry
ComponentAdapter::invokeCallback.
[Chain:[AbstractCallback:public void glue.AccountBeanGlue.preCredit(adapt.AbstractContext)]]
in  preCredit: accessing the args
Trying to credit:50.0
Demo of "Augmenting the functionality" settting the value to 0.0
```

The client-side results:

```
Looking up home interface
Getting account
Found an existing account
Setting overdraft
Credit the account with 50.0
Account balance before credit = 4700.0
Account balance after credit = 4700.0
```

Note: The amount before and after the credit transaction remains the same. This reflects the *augmentation* of functionality or changing the value of parameters for the instrumented method.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The goal of the thesis was to come up with a design methodology for adapting the behavior of Beans that conform to an industry standard software component model. Active Interfaces[11] are modeled formally for the first time. Chapter 3 describes the methodology and explains the use of Z and Z' for adapting the EJB. The Z file specifies all the aspects of the EJB component. It specifies the EJB modeling, server and the container configurations along with the Home and Remote interfaces. It declares the bean's implementation domain and the source file (AccountBean). The Z file shows the schema modeling for the business logic. The Z' is gets deployed with the *adapted* bean after specifying the adaptations. It contains the original Z file that describes the EJB application and the business logic. In addition, the Z' file describes the changes in the bean's implementation, the schemas corresponding to the glue code, and the schemas that show the insertion of Active Interfaces callbacks into the EJB source methods. It also describes three forms of adaptation : augmentation, denial and overriding of the behavior of the original component. Thus the overall goal to map the Z specifications to a working EJB and then

mapping the Z' to the EJB plus the glue and registry code is achieved. The original Enterprise Archive (EAR) file contains the bean's `HomeInterface, RemoteInterface` and the implementation class, the jar file with all the classes and the Z specifications. After running the scripts shown in Appendix B, we get the EAR' file that contains the `HomeInterface, RemoteInterface`, the bean's instrumented and compiled class, the glue and registry classes, the modified jar file and the Z' specification file.

5.2 Future Work

Figure 5.1 and Figure 5.2 explains the overall design technique of automating the manual work of mapping the Z' specifications to the glue-code.

5.2.1 Using Refinement

The entire system that is modeled in Z-specifications can be refined [30] to generate the glue code. The desired changes in the behavior are specified in the Z' file. There are stages in the refinement process that ultimately lead to the code-generation. The abstract model generated by the refinement in Z, has a mechanism to declare the variables as invariants. The refinements "zoom" the specification at every step and the final stage gives the code. There are cycles of refinement and at the end of every cycle the Z-specifications go closer to the implementation.

5.2.2 A utility for extracting the delta changes from the Z' file

A utility that parses the Z and Z' file and *understands* the changes, can be used to automate the process of mapping the changes in the Z' file to the glue code. A stream editor tool may be the used for that purpose. It is possible to compare the changes from the Z' file to the original Z file and generate the *changes* in the bean's implementation. It can be

possible to determine all the schemas indicating the changes in the methods that are to be instrumented. Once the business methods that need to be instrumented are known, the bean implementation class can be edited and the callbacks can be inserted. In this thesis, we have not investigated the extent to which the behavioral changes can be specified in Z notations.

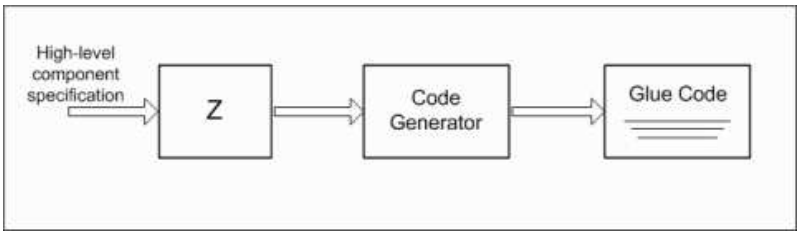

Figure 5.1: Future Work - part A.

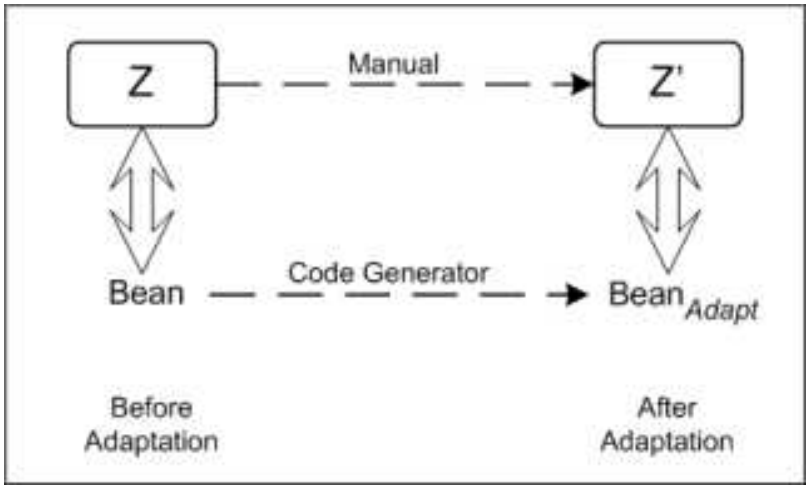

Figure 5.2: Future Work - part B.

Appendix A

Interfaces and Classes used in EJB

```
package javax.ejb

Interfaces:
public interface EJBContext
public interface EJBHome
public interface EJBLocalHome
public interface EJBLocalObject
public interface EJBMetaData
public interface EJBObject
public interface EnterpriseBean
public interface EntityBean
public interface EntityContext
public interface Handle
public interface HomeHandle
public interface MessageDrivenBean
public interface MessageDrivenContext
public interface SessionBean
public interface SessionContext
public interface SessionSynchronization

Classes:
public class AccessLocalException
public class CreateException
public class DuplicateKeyException
public class EJBException
public class FinderException
public class NoSuchEntityException
public class NoSuchObjectLocalException
public class ObjectNotFoundException
public class RemoveException
public class TransactionRequiredLocalException
public class TransactionRolledbackLocalException
```

Appendix B

Scripts used for Adaptation

B.1 adaptClass.sh

```
#!/bin/sh

# Usage: adaptClass <jarName> <className>
#
# Note: This runs the compiler; the front-end GUI should be
# used to selectively instrument methods.

if [ "x$AIDE_HOME" = "x" ]
then
  echo "AIDE_HOME variable not set."
  exit 1
fi

if [ $# -ne 2 ]
then
  echo "Usage: adaptClass <jarName> <className>"
  exit 1
fi

BEAN=$1
CLASSNAME=$2

TESTSRC=‘echo $CLASSNAME| cut -f1 -d.‘
TESTJAVA=‘echo $CLASSNAME| cut -f2 -d.‘
if [ "x$TESTSRC" = "x" ]
then
  echo "Error: invalid class name: $CLASSNAME"
  exit 1
fi

if [ ! "x$TESTJAVA" = "xjava" ]
then
  echo "Error: invalid class name: $CLASSNAME"
  exit 1
fi

BEANNAME=‘echo $BEAN | cut -f1 -d.‘

if [ "x$BEANNAME" = "" ]
then
  echo "Error: $BEAN is an invalid Bean name."
  exit 1
fi
```

```
# The DIR is the directory one level above the probe directory
DIR="src/$BEAN/src/"
if [ ! -d $DIR ]
then
  echo "Error: can't locate directory $DIR"
  exit 1
fi

# Enable the compiler code to be able to find the current location
CLASSPATH=$CLASSPATH:$PWD/tmp/$BEAN/classes
export CLASSPATH

# Go into the directory and compile the bean.
cd $DIR


/usr/local/bin/perl $AIDE_HOME/bin/compile.pl \
                    examples/ejb/entity/bmp/$BEANNAME/$CLASSNAME
```

B.2 updateJar.sh

```
#!/bin/sh

# Usage: updateJar <beanName>
#
#    1. Extract .class files from the glue directory (Note: these
#       glue files will only be removed from the Bean jar file using
#       the restoreJar command.)
#
#    2. Within the glue directory, there will be a replicated hierarchy
#       to match the bean which is being adapted. The EJB searches for the
#       class glue.X where X is the name of the EJB itself.
#
#    3. The .ear file must be updated because that file is
#       deployed.

if [ $# -ne 1 ]
then
  echo "Usage: updateJar <beanName>"
  exit 1
fi

BEAN=$1

DIR=src/$BEAN/src/probe

if [ ! -d $DIR ]
then
  echo "unable to locate probe directory for $BEAN"
  exit 1
fi

# replace .class file in the tmp directory
TARGET=$PWD/tmp/$BEAN/$BEAN

if [ ! -f $TARGET ]
then
  echo "Error: Unable to locate target bean file: $TARGET"
  exit 1
fi

# replace the files in the Jar
```

```
HERE=‘pwd‘
cd $DIR

CLASSES=‘find examples -name "*.class" -print‘

if [ "x$CLASSES" = "x" ]
then
  echo "Nothing to update. You must run AIDE first."
  exit 0
fi

# Update class files
echo "Updating class files in target bean..."
jar uvf $TARGET $CLASSES

# Grab all classes in the glue hierarchy, including both
# registry and glue files.

# Update glue files
echo "Updating glue files in target bean..."
(cd $HERE; jar uvf $TARGET ‘find glue -name "*.class" -print‘)

# Update classes file in the bean directory.
# The references to tmp/*.jar/classes/ are to be removed.
# The instrumented classes are stored in this directory.

CLASSESDIR=‘dirname $TARGET‘/classes
echo "copying $CLASSES into the /classes/ directory."

(cd $HERE; cd $DIR; cp -r examples $CLASSESDIR)

echo
echo "$BEAN Updated:"
ls -l $TARGET

echo
echo "updating AIDEExample.ear file"
(cd ‘dirname $TARGET‘; jar uvf $HERE/AIDEExample.ear $BEAN)

# Now must delete the entire probe directory. Go up one level to
# be able to delete the directory. DIR is relative to HERE.
cd $HERE; rm -fr $DIR
echo "done."
```

Appendix C

Source Files

C.1 Glue Code

C.1.1 AccountBeanGlue.java

```
package glue;

import adapt.*;

/**
 * This file contains the glue code for adapting the AccountBean class.
 */
public class AccountBeanGlue {

    /** Default constructor
    */
    public AccountBeanGlue () { }

    /**
     * This is the method that will be invoked PRE the debit method
     * in the AccountBean class.
     */
    public void preDebit(adapt.AbstractContext context)  {
    // getting the args
    System.out.println("in PRE debit: accessing args");
    Object args[] = context.getArgs();

    // first argument contains a Double parameter
    double dval = ((Double) args[0]).doubleValue();

    //For Augmenting changing the original value of the parameter.
    args[0] =new Double(0.0);

    }
}
```

C.1.2 AccountBeanRegistry.java

```
/*
 * This registers the bean to be adaptable.
 */
package glue.examples.ejb.entity.bmp.bankaccount;
```

```
import adapt.*;
import glue.AdaptationRegistry;

public class AccountBeanRegistry implements glue.AdaptationRegistry {

    /*
     * Receive the object which is adaptable and register a glue code.
     */
    public boolean init (Object obj) {
    if (!(obj instanceof Adaptable)) {
        System.out.println("Returning false for obj instanceof Adaptable");
    }
    else System.out.println("Proceeding to inserting callback");

    Adaptable adaptable = (Adaptable) obj;

    //The component adapter should not be NULL

    ComponentAdapter ca1 ;
    ca1 = new adapt.ComponentAdapter();
    adaptable.setAdapter (ca1);

    System.out.println ("in AccountBeanRegistry::init()");
    Class [] args = new Class [1];
    args[0] = new Double(1.0f).getClass();
    System.out.println("Before inserting callback in AccountBeanRegistry");

    //Creating glue object that implements the preDebit method

    glue.AccountBeanGlue agl = new glue.AccountBeanGlue();

    try{
        ca1.insertHook(adapt.ComponentAdapter.BEFORE,"debit","double",\
                                    agl,"preDebit","adapt.AbstractContext");

        ca1.insertHook(adapt.ComponentAdapter.BEFORE,"credit","double", \
                                    agl,"preCredit","adapt.AbstractContext");

        System.out.println("after successfylly inserting the callback ");
    }
    catch(Exception e){ System.out.println(e.toString());}
    return true;
    }
}
```

C.1.3 The Instrumented `debit()` method

```
public void debit  (  double amt   ) throws OverdraftExceededException,\\
                                            InvalidAmountException  {
 {
//BEFORE-START: before Callback --------------------]]
if (adapter != null) {
        Object a__rgs__[] = new Object[1];
        a__rgs__[0] =  new Double (amt);
        Object __o__ = adapter.invokeCallback (adapt.ComponentAdapter.BEFORE,\\
                                                "debit(double)",  this, a__rgs__ );
        // [Deny] Do not process this method
        if (__o__ instanceof Exception) return;
        // [Override] future work
        if (__o__ == null) return;
        //  [Augment] arguments can be modified and used by this function

        //              Note: final variables cannot be modified and are silently left out.
```

```
        amt = ((Double) a__rgs__[0]).doubleValue();

        }
// BEFORE-END: before Callback ----------------------]]
}

{
if (amt<0) throw new InvalidAmountException();
balance -= amt;
 if (balance < -overdraft) {
             ctx.setRollbackOnly();             throw new OverdraftExceededException();
}

{
//AFTER-START: before Callback --------------------]]
//
if (adapter != null) {
        Object a__rgs__[] = new Object[1];
        a__rgs__[0] =  new Double (amt);
        adapter.invokeCallback (adapt.ComponentAdapter.AFTER, "debit(double)",  this, a__rgs__ );
}
//
// AFTER-END: before Callback ----------------------]]
}
```

C.2 EJB Code

C.2.1 AccountBean.java

These are the methods in `AccountBean.java` that are needed apart from the business logic implementations to register the EJB to be adaptable.

```
/**
 * <dl>
 * <dt> This class implements EJB business logic
 *      Reference: http://www.iona.com
 * </dl>
 */

import javax.transaction.*;
import javax.naming.*;
import javax.ejb.*;
import javax.sql.*;
import java.util.*;
import java.io.*;
import java.sql.*;

public class AccountBean implements EntityBean {

  private String holder;
  private double balance;
  private double overdraft;
  private boolean flag ;
  private EntityContext ctx;

  final static String RESOURCE_NAME="jdbc/Accounts";

  /**
   * @see Account#getHolder
   */
```

```
  public String getHolder() {
    return holder;
  }

  /**
   * @see Account#getBalance
   */
  public double getBalance() {
    return balance;
  }

  /**
   * @see Account#setOverdraft
   */
  public void setOverdraft(double overdraft) throws InvalidAmountException, OverdraftExceededException {
    if (overdraft < 0) throw new InvalidAmountException();
    if (balance < -overdraft) throw new OverdraftExceededException();
    this.overdraft = overdraft;
  }

  /**
   * @see Account#getOverdraft
   */
  public double getOverdraft() {

    return overdraft;
  }

  /**
   * @see Account#credit
   */
  public void credit(double amt) throws InvalidAmountException {
    if (amt<0) throw new InvalidAmountException();
    balance += amt;
  }

  /**
   * @see Account#debit
   */
    public void debit(double amt) throws OverdraftExceededException, InvalidAmountException {

System.err.println ("in AccountBean.java::debut()");

if (amt<0) throw new InvalidAmountException();
balance -= amt;
if (balance < -overdraft) {
    ctx.setRollbackOnly();
    throw new OverdraftExceededException();
}
    }

    public void setEntityContext(EntityContext ctx) {
this.ctx = ctx;
    }

    public void unsetEntityContext() {
this.ctx = null;
    }

    public void ejbActivate() {
     if (flag)
 instrumentBean();
     flag = false;
}
```

```
    public void ejbPassivate() {}

    private Connection getConnection() {
try {
    return ((DataSource)new InitialContext().lookup("java:comp/env/"+RESOURCE_NAME)).getConnection();
} catch (Exception ex) {
    throw new EJBException(ex);
}
    }

  public Integer ejbCreate(int id, String holder, double balance) throws CreateException {
      this.holder = holder;
      this.balance = balance;

     final int VENDOR_SPECIFIC_ERROR_CODE_UNIQUE_CONSTRAINT = 1;

      Connection conn = getConnection();
      try {
      PreparedStatement pstmt = conn.prepareStatement("INSERT INTO BANKACCOUNT VALUES (?,?,?,?)");
      pstmt.setInt(1, id);
      pstmt.setString(2, holder);
      pstmt.setDouble(3, balance);
      pstmt.setDouble(4, overdraft);
      pstmt.executeUpdate();
      return new Integer(id);
    } catch (SQLException ex) {
      if(ex.getErrorCode() == VENDOR_SPECIFIC_ERROR_CODE_UNIQUE_CONSTRAINT) {
        throw new DuplicateKeyException("Account with primary key of "+id+" already exists");
      } else {
        throw new CreateException(ex.getMessage());
      }
    } catch (Exception ex) {
      throw new EJBException(ex);
    } finally {
      try {
        conn.close();
      } catch (SQLException ex) {}
    }
  }

    public void ejbPostCreate(int id, String holder, double balance) {
     System.out.println("in ejb post create");
// instrunment in all the ejb methods for testing
instrumentBean();
    }


    //************Here is the instrument method******//

    private void instrumentBean() {
      try {
  Class registryClass = Class.forName ("glue." + this.getClass().getName() + "Registry");

  // Once the registry class is found, call its init() method to have the callbacks dynamically
  // inserted.

  glue.AdaptationRegistry reg = (glue.AdaptationRegistry) registryClass.newInstance();

  reg.init (this);
      } catch (Exception e) {
  System.err.println (e.toString());
      }
    }

  public Integer ejbFindByPrimaryKey(Integer pk) throws FinderException {
```

```
  Connection conn = getConnection();
  try {
    PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM BANKACCOUNT WHERE ID = ?");
    pstmt.setInt(1, pk.intValue());
    ResultSet rs = pstmt.executeQuery();
    if (!rs.next()) throw new NoSuchEntityException("No such account found with primary key "+pk);
    return pk;
  } catch (SQLException ex) {
    throw new FinderException(ex.toString());
  } finally {
    try {
      conn.close();
    } catch (SQLException ex) {}
  }
}

public void ejbLoad() {
  Integer pk = (Integer)ctx.getPrimaryKey();
  Connection conn = getConnection();
  try {
    PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM BANKACCOUNT WHERE ID = ?");
    pstmt.setInt(1, pk.intValue());
    ResultSet rs = pstmt.executeQuery();
    if (!rs.next()) throw new EJBException("Account with primary key "+pk+" not found");
    holder = rs.getString("holder");
    balance = rs.getDouble("balance");
    overdraft = rs.getDouble("overdraft");
  } catch (Exception ex) {
    throw new EJBException(ex);
  } finally {
    try {
      conn.close();
    } catch (SQLException ex) {}
  }

  System.out.println("in ejb load");


  // Calling this method doesnot affect life-cycle of any
  //EJB objects
  instrumentBean();
}

public void ejbStore() {
  Integer pk = (Integer)ctx.getPrimaryKey();
  Connection conn = getConnection();
  try {
    PreparedStatement pstmt = conn.prepareStatement("UPDATE BANKACCOUNT SET HOLDER=?, BALANCE=?,\\
                                                     OVERDRAFT=? WHERE ID=?");
    pstmt.setString(1, holder);
    pstmt.setDouble(2, balance);
    pstmt.setDouble(3, overdraft);
    pstmt.setInt(4, pk.intValue());
    pstmt.executeUpdate();
  } catch (Exception ex) {
    throw new EJBException(ex);
  } finally {
    try {
      conn.close();
    } catch (SQLException ex) {}
  }
}

public void ejbRemove() {
  Integer pk = (Integer)ctx.getPrimaryKey();
```

```
    Connection conn = getConnection();
    try {
      PreparedStatement pstmt = conn.prepareStatement("DELETE FROM BANKACCOUNT WHERE ID = ?");
      pstmt.setInt(1, pk.intValue());
      pstmt.executeUpdate();
    } catch (Exception ex) {
      throw new EJBException(ex);
    } finally {
      try {
        conn.close();
      } catch (SQLException ex) {}
    }
  }

}
```

C.2.2 AccountHome.java

```
package examples.ejb.entity.bmp.bankaccount;

import javax.ejb.*;
import java.rmi.RemoteException;
/**
 * <dl>
 * <dt> This interface gives the EJB specific methods
 *      Reference: http://www.iona.com
 * </dl>
 */
public interface AccountHome extends EJBHome {
 /**
   * Method for creating an account for a new user.
   * @param id is an integer that maps to the primary key
   * @param holder is the String that represents holder's name
   * @param balance is the starting balance
   * @exception CreateException is thrown if the holder is
   *                              already in the database
   * @exception RemoteException is a remote exception thrown
   *                             in case of a transaction roll back
   */
  Account create(int id, String holder, double balance)
                          throws CreateException, RemoteException;

 /**
   * Method for finding an existing account holder
   * @param id is an integer that maps to the primary key
   * @exception FinderException is thrown when the user cannot be located
   * @exception RemoteException is a remote exception thrown in case
   *                                      of a transaction roll back
   */
  Account findByPrimaryKey(Integer pk)  throws FinderException, RemoteException;
}
```

C.2.3 Account.java

```
package examples.ejb.entity.bmp.bankaccount;

import javax.ejb.*;
import java.rmi.RemoteException;
/**
 *  <dl>
 *  <dt> Purpose: This class states all the business methods
 *       supported by the EJB server.
```

```
 *       It does not have any methods that deal with Active Interfaces
 *       Reference : http://www.iona.com
 *  </dl>
 */

public interface Account extends EJBObject {

  /**
   * Return the account holder's name
   * @return the fullname of the account holder
   */
  public String getHolder() throws RemoteException;

  /**
   * Return the balance.
   * @return the current balance of the account
   */
  public double getBalance() throws RemoteException;


  /**
   * Set the overdraft limit.
   * @param overdraft a positive number representing the amount the bank is
   *                                                  willing to lend
   * @exception InvalidAmountException if the overdraft is negative
   * @exception OverdraftExceededException if the new overdraft is already exceeded
   */
  public void setOverdraft(double overdraft)
        throws InvalidAmountException, OverdraftExceededException, RemoteException;

  /**
   * Get the overdraft limit.
   */
  public double getOverdraft() throws RemoteException;

  /**
   * Credit the account.
   * @param amt amount to credit the account by
   * @exception InvalidAmountException if the amount is negative
   */
  public void credit(double amt) throws InvalidAmountException, RemoteException;

  /**
   * Debit the account.
   * @param amt amount to debit the account by
   * @exception OverdraftExceededException if the overdraft has been exceeded
   */
  public void debit(double amt)
     throws OverdraftExceededException, InvalidAmountException, RemoteException;
}
```

Appendix D

The XML files required for the EJB application

D.1 application.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.\
             //DTD J2EE Application 1.2//EN"\
             "http://java.sun.com/j2ee/dtds/application_1_2.dtd">

<application>
  <display-name>AIDEExample</display-name>
  <description>An example demo containing an entity bean </description>

  <module>
    <ejb>bankaccount.jar</ejb>
  </module>

  <module>
    <java>bankaccountclient.jar</java>
  </module>

</application>
```

D.2 application-client.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE application-client PUBLIC '-//Sun Microsystems, Inc.\
              //DTD J2EE Application Client 1.2//EN'\
              'http://java.sun.com/j2ee/dtds/application-client_1_2.dtd'>

<application-client>
  <display-name>AccountClient</display-name>
  <description></description>
  <ejb-ref>
    <ejb-ref-name>ejb/Account</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <home>examples.ejb.entity.bmp.bankaccount.AccountHome</home>
    <remote>examples.ejb.entity.bmp.bankaccount.Account</remote>
    <ejb-link>Account</ejb-link>
```

```
  </ejb-ref>
</application-client>
```

D.3 cc.xml

This file is automatically generated by the "iportal.deploy" tool provided by IONA's ipas 3.0 application server.

```
<!-- Container Configuration for iPortal Application Server
     Copyright (c)2000-2001 IONA Technologies PLC.All Rights Reserved.

     For more information on container configurations,
     please consult the iPortal Application Server User Guide.  -->

<configuration>

  <enterprise-beans>

    <entity>
      <ejb-name>Account</ejb-name>
      <jndi-name>iona/ipas/simple/Account</jndi-name>
      <jndi-source-name>CosNaming</jndi-source-name>
      <resource-ref>
        <res-ref-name>jdbc/Accounts</res-ref-name>
        <res-ref-link>JDBCAccounts</res-ref-link>
      </resource-ref>
    </entity>

  <application-client>
    <application-client-name>AccountClient</application-client-name>
  </application-client>

  <resources>

    <resource>
      <resource-name>JDBCAccounts</resource-name>
      <jndi-name>iona:cloudscape/demos</jndi-name>
    </resource>

  </resources>

  <jndi-sources>
    <jndi-source>
      <jndi-source-name>CosNaming</jndi-source-name>
      <property>
        <prop-name>java.naming.factory.initial</prop-name>
        <prop-value>com.sun.jndi.cosnaming.CNCtxFactory</prop-value>
      </property>
    </jndi-source>
  </jndi-sources>

</configuration>
```

D.4 ejbjar.xml

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.\
                 //DTD Enterprise JavaBeans 1.1//EN"\
                 "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
```

```
<description>A jar containing one entity bean with bean
             managed persistence.
</description>
<display-name>BankAccount Example</display-name>

<enterprise-beans>
  <entity>
    <description>An entity bean for banking system and bean
    managed persistence
    </description>
    <display-name>Account</display-name>

    <ejb-name>Account</ejb-name>
    <home>examples.ejb.entity.bmp.bankaccount.AccountHome</home>
    <remote>examples.ejb.entity.bmp.bankaccount.Account</remote>
    <ejb-class>examples.ejb.entity.bmp.bankaccount.AccountBean
    </ejb-class>

    <persistence-type>Bean</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>True</reentrant>

    <resource-ref>
      <res-ref-name>jdbc/Accounts</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
    </resource-ref>

  </entity>
</enterprise-beans>

<assembly-descriptor>

  <container-transaction>
    <description></description>
    <method>
      <ejb-name>Account</ejb-name>
      <method-intf>Home</method-intf>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>Account</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>getHolder</method-name>
    </method>
    <method>
      <ejb-name>Account</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>getBalance</method-name>
    </method>
    <method>
      <ejb-name>Account</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>setOverdraft</method-name>
    </method>
    <method>
      <ejb-name>Account</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>getOverdraft</method-name>
    </method>
    <method>
      <ejb-name>Account</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>remove</method-name>
    </method>
```

```
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>

    <container-transaction>
      <method>
        <ejb-name>Account</ejb-name>
        <method-intf></method-intf>
        <method-name>credit</method-name>
      </method>
      <method>
        <ejb-name>Account</ejb-name>
        <method-intf></method-intf>
        <method-name>debit</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>

  </assembly-descriptor>
</ejb-jar>
```

Bibliography

[1] George T. Heineman and William T. Councill. *Component-Based Software Engineering*. Addison Wesley Longman, Inc., June 2001.

[2] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Technical Concepts of Component-Based Software Engineering. Technical report, Carnegie Mellon University - Software Engineering Institute, May 2000.

[3] George T. Heineman. An Experimental Evaluation of Component Adaptation Techniques. Technical report, Worcester Polytechnic Institute, November 1999.

[4] Microsoft. COM Specifications. Website - http://www.microsoft.com/com, 2000.

[5] Sun Microsystems Inc. Ejb 1.1 Specifications. Website - ftp://ftp.java.sun.com/pub/ejb/1_1final-129822/ejb1_1-spec.pdf, November 1999.

[6] George T. Heineman. Adaptation of Software Components. Technical report, Department of Computer Science, Worcester Polytechnic Institute, USA, April 1999.

[7] Wojtek Kozaczynski. Composite Nature of Component. Technical report, Rational Software, USA, 1999.

[8] Jan Bosch. Superimposition: A Component Adaptation Technique. *Information and Software Technology*, 1999.

[9] Rahul Sharma. Java2 Enterprise Edition J2EE™ Connector Architecture Specification Version1.0. Technical report, February 2002.

[10] Ralph Keller and Urs Hölzle. Binary Component Adaptation. Technical report, University of California, Santa Barbara, USA, December 1997.

[11] George T. Heineman. A Model for Designing Adaptable Software Components. pages 121 – 127, Vienna, Austria, August 1998. 22nd Annual International Computer Science and Application Conference (COMPSAC-98).

[12] Erich Gamma, Richard Helm, Ralf Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, October 1994.

[13] Kyle Brown. Rules and Patterns for Session Facades. Technical report, IBM, June 2001.

[14] Paul Calnan. Extensible Treansformation and Compiler Technology. Master's thesis, Worcester Polytechnic Institute, Worcester, MA 01609, 2002.

[15] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison Wesley, second edition, 1999.

[16] Helgo M. Ohlenbusch. Software Component Adaptation Mechanisms. Master's thesis, Worcester Polytechnic Institute, 1999.

[17] Axel van Lamsweerde. Formal Specification: A Roadmap. Technical report, Département d'Ingénierie Informatique, Université catholique de Louvain, 2000.

[18] Lavern Hall and ReUse/OO-Component Team. COTS-based OO-Component Approach for Software Inter-operability and Reuse. Technical report, California Institute of Technology, 4800 Oak Grove Drive, CA 91109-8099, USA, 2000.

[19] J. M. Spivey. *The Z Notation: A Reference Manual*. Programming Research Group, University of Oxford, second edition, 1998.

[20] Cadiz Reference Manual. Website - http://www-users.cs.york.ac.uk/ ian/cadiz, January 2002.

[21] Henri Jubin, Jürgen Friedrichs, and The Jalapeno Team. *Enterprise JavaBeans by Example*. Prentice Hall, August 1999.

[22] Http Protocol. Website - http://www.w3.org, 2001.

[23] Extensible Markup Language (xml). Website - http://www.w3.org/XML, 2002.

[24] iPortal Application Server Developer's Guide. Website - http://www.iona.com, May 2001.

[25] Robert Vieira. *Professional SQL Server Programming*. Wrox, 2002.

[26] Cloudscape. Website - http://www.cloudscape.com, 2002.

[27] Java Naming Directory Interface. Website - http://java.sun.com/jndi, 2002.

[28] Il Hyung Cho and John D. McGregor. Component Specification and Testing Interoperation of Components. Technical report, Department of Computer Science, Clemson University, 1999.

[29] Dorel D. Baluta. A formal Specification in Z of the Relational Data Model, Version 2, of E.F. Codd. Master's thesis, Concordia University, Montreal, Quebec, Canada, March 1995.

[30] Richard Paige. Specification and Refinement using a Heterogeneous Notation for Concurrency and Communication. Technical report, Department of Computer Science, York University, Toronto, Canada, 1998.