DISSERTATION

Adaptation and Composition Techniques for Component-Based Software Engineering

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften

unter der Leitung von

o. Univ.-Prof. Dipl.-Ing. Dr.techn. Mehdi Jazayeri
E184-1
Institut für Informationssysteme

eingereicht an der

Technischen Universität Wien
Technisch-Naturwissenschaftliche Fakultät

von

Dipl.-Ing. Thomas Gschwind
tom@infosys.tuwien.ac.at
Matrikelnummer: 9225658
Laudongasse 28/14
A-1080 Wien
Österreich

Wien, im Februar 2002

Meiner Mutter

Zusammenfassung

Die Bedeutung von Softwarekomponenten wurde in den letzten Jahren auch von der Industrie erkannt. Dies wird durch die Anzahl der kommerziell entwickelten Komponentenmodelle deutlich. Das Ziel von Softwarekomponenten ist die vermehrte Softwarewiederverwendung, sowie die Vereinfachung der Implementierung und Komposition von Komponenten. Heutige Komponentenmodelle beschäftigen sich primär mit der Spezifikation von Komponenten und beachten die Adaption dieser nur wenig. Daher ist es noch immer notwendig, die einzelnen Komponenten manuell zusammenzusetzen, sofern sie nicht füreinander entwickelt worden sind. Da einer der Vorteile von Softwarekomponenten die unabhängige Entwicklung der Komponenten sein soll, kann man im allgemeinen nicht davon ausgehen, daß die Komponenten füreinander entwickelt worden sind und daher kompatibel zueinander sind.

Um die Komposition von Softwarekomponenten zu vereinfachen, ist es notwendig diese zu adaptieren. Leider beschäftigen sich nur wenige Ansätze mit der automatischen Adaption von Komponenten. Jene Ansätze die sich mit der automatischen Adaption beschäftigen, wie zum Beispiel Generische Programmierung (*generic programming*), beschäftigen sich nur mit der Erzeugung von Komponenten, die bestimmte Anforderungen erfüllen sollen oder nur mit der Optimierung eines aus Softwarekomponenten bestehenden Systems, nicht aber mit der Anpassung der von einer Softwarekomponente zur Verfügung gestellten Schnittstelle.

Um das Problem der automatischen Adaptierung der Schnittstelle einer Komponente zu lösen, haben wir Typbasierte Adaptierung (*type-based adaptation*) entwickelt, die in dieser Arbeit beschrieben wird. Typbasierte Adaptierung setzt auf den Schnittstellen auf, die von einer Komponente zur Verfügung gestellt werden. Nahezu alle heutigen Komponententechnologien sind stark typisiert und stellen einen Mechanismus zur Verfügung, um die Schnittstellen, die von einer Komponente implementiert werden, abzufragen. Daher kann Typbasierte Adaptierung leicht in ein bestehendes System integriert werden.

Basierend auf der Vorstellung der unterschiedlichen Adaptionstechniken wird eine Klassifikation gegeben. Diese Klassifikation ermöglicht es, die unterschiedlichen Techniken in Relation zueinander zu stellen und ermöglicht Entwicklern die richtige Adaptionstechnik für die Lösung eines Problemes auszuwählen.

Schlagworte: software components, software engineering, adaptation, composition, classification, assessment.

Abstract

Component models have received much attention recently both from the software engineering research community and from industry. This is apparent through the sheer number of component models that have been developed in the last few years. The goal of each of these models is to increase reuse and to simplify the implementation and composition of new software. All these models focus on the specification and packaging of components but provide almost no support for the adaptation and easy composition of components. If components have not been written with each other in mind, their composition still has to be carried out programmatically. Since one of the advantages of software components is independent development, they will rarely be entirely compatible with each other.

Hence, it is imperative to focus on the adaptation of software components to simplify their composition. Though many research projects exist in this area, they try to target different aspects of adaptation. While some of these projects support the automated generation of components based on a requirements specification or the automated optimization of a given composition, none of these techniques supports automated adaptation to simplify the composition process itself.

To solve this problem, we have developed *type-based adaptation*, a novel adaptation technique that is based on a simple but powerful and efficient formal description of a component, namely its type. Type-based adaptation exploits a property shared by almost all of today's component models, namely, that they use strongly typed components and support querying a component's type using reflection or a similar approach. This allows type-based adaptation to be added transparently to an existing system without having to modify the components for that system. We provide a reference implementation for different environments to show the feasibility of our approach.

We will also present our classification of today's adaptation techniques. Such a classification is important since it allows researchers to put the different approaches in relation to each other and allows developers to select an appropriate adaptation algorithm to solve a given problem. The classification is based on a series of case studies we have performed to evaluate the different adaptation and composition approaches.

Keywords: software components, software engineering, adaptation, composition, classification, assessment.

Acknowledgments

This thesis is dedicated to my mother because I would have never been able to accomplish it without her love and support.
I would like to thank my brother for his invaluable help and advice and my supervisor Mehdi Jazayeri for his comments, advice, and discussions. Finally, I would like to thank Pankaj K. Garg of Hewlett Packard Laboratories for our many interesting discussions about type-based adaptation and its use for transaction monitoring, Metin Feridun of IBM Research for our discussions about service adaptation in the context of Mobile Agents, and Pedrick Moore for proofreading this thesis.

Additionally, I would like to thank IBM Research, Zurich Research Laboratory for sponsoring this thesis as part of a University Partnership Award and the European Union for sponsoring this thesis as part of the EASYCOMP project (IST 1999-14191).

Contents

1 Introduction **1**
1.1 Contribution 2
1.2 Motivation 3
1.3 Outline 3

2 Component Models **5**
2.1 Libraries 5
2.2 JavaBeans 6
2.2.1 Properties 8
2.2.2 Events 9
2.2.3 Methods 10
2.2.4 Introspection 10
2.3 Enterprise JavaBeans 10
2.3.1 Architecture 11
2.3.2 Beans 12
2.3.3 Packaging and Deployment 16
2.4 The CORBA Component Model (CCM) 17
2.4.1 Component Specification 18
2.4.2 Component Implementation 20
2.4.3 Container Model and Architecture 21
2.4.4 Client Programming Model 23
2.4.5 Component Assembly and Packaging 23
2.4.6 Component Deployment 24
2.5 Summary 24
2.5.1 Desktop Component Models 24
2.5.2 Server-Side Component Models 25

3 Adaptation of Components **26**
3.1 Adaptation for Composition 26
3.1.1 Scripting Languages 26
3.1.2 Composition Languages 28
3.1.3 Wrapping 29

3.1.4 The Software Bus . . . 30
3.2 Separation of Concerns . . . 32
3.2.1 Configuration Languages . . . 32
3.2.2 Generic Programming . . . 33
3.2.3 Generative Programming . . . 34
3.2.4 Aspect-Oriented Programming . . . 34
3.3 Performance Adaptation . . . 36
3.3.1 Simplicissimus . . . 36
3.3.2 COMPOST . . . 37

4 Towards A Classification 38
4.1 Objectives of Adaptation . . . 38
4.2 Adaptation Transparency . . . 39
4.3 Application Domain . . . 41
4.4 Adaptation Time . . . 41
4.5 Degree of Automation . . . 42
4.6 Summary . . . 42

5 Type-Based Adaptation 45
5.1 Fundamentals . . . 45
5.2 Adaptation Model . . . 48
5.3 Requirements . . . 50
5.3.1 Adapter Description . . . 51
5.3.2 Packaging . . . 51
5.3.3 The Adapter Repository . . . 52
5.3.4 The Adaptation Component . . . 52
5.4 Requirements for Server-Side Component Models . . . 53
5.4.1 Trading Services . . . 54
5.4.2 Adapter Location and Usage . . . 54
5.4.3 Security Considerations . . . 56
5.5 Requirements for Desktop Component Models . . . 57
5.5.1 Granularity of Adaptation . . . 57
5.5.2 The Adaptation Component . . . 59
5.5.3 Adapter Description . . . 59
5.5.4 Performance Considerations . . . 59

6 Evaluation 62
6.1 Server-Side Component Models and Web Services . . . 62
6.1.1 Address Book Component . . . 63
6.1.2 Weather-Service Component . . . 64
6.2 Desktop Component Models . . . 65
6.2.1 The Component Work Bench . . . 65

6.3 Mobile Agent Systems . . . 69
6.3.1 Mobile Agent Definitions . . . 69
6.3.2 The AgentBean Development Kit . . . 69
6.3.3 The Calendar Agent . . . 72
6.4 Summary . . . 73

7 Related Work **75**

8 Conclusions **78**
8.1 Contributions . . . 79
8.2 Future Research Directions . . . 79

A Glossary **81**

B Examples **83**
B.1 JavaBeans . . . 83
B.1.1 Bean Class . . . 83
B.1.2 BeanInfo Class . . . 83
B.2 Enterprise JavaBeans . . . 84
B.2.1 Session Bean Example . . . 84
B.2.2 Entity Bean Example . . . 87
B.2.3 Message Driven Bean Example . . . 90

List of Figures

2.1 The AgentBean Development Kit 7
2.2 A Simple JavaBean 7
2.3 The Enterprise JavaBean Architecture 11
2.4 UML Diagram of an EJB Session Bean 12
2.5 The CORBA Container Architecture 22
2.6 Typical Web Service Architecture 25

3.1 Defining Point-Cuts in AspectJ 35

4.1 Objectives of Adaptation 38

5.1 Basic Subtyping Rules 46
5.2 Subtyping Rule for Functions 47
5.3 Subtyping Rule for Object and Interface Types 48
5.4 A Sample Adapter Repository 50
5.5 Adapter Working on the Interface Level 52
5.6 Accessing a Service in a Distributed Component System 55
5.7 Typical Client Code in a Distributed Component System 55
5.8 Adaptation on Method Basis 58
5.9 Adapter Working on the Signature Level 60

6.1 Internet Shopping Application 63
6.2 Integrated Development Environment 65
6.3 Connectors of the Component Workbench 66
6.4 Type-Based Adaptation Connection Wizard for the Component Workbench 68
6.5 Example Illustrating the Three Component Categories 70
6.6 Based the ADK the Developer Combines Components to Build the Agent 70

List of Tables

2.1 Standard Methods to Read and Write Properties 8
2.2 KeyListener Interface 9
2.3 Standard Methods to Subscribe to and to Unsubscribe from Events . . 10
2.4 Methods Declared in the Session Bean's Home Interface 13
2.5 Methods Declared in the SessionBean Interface 13
2.6 Methods Declared in the Entity Bean's Home Interface 15
2.7 Additional Methods Declared in the EntityBean Interface 16
2.8 Methods Declared in the Entity Bean's Home Interface 17

4.1 Classification of Adaptation Techniques 43

5.1 Methods Provided by the Adaptation Component 53
5.2 Methods provided by the Trading Service 56

Chapter 1

Introduction

The purpose of software components is to increase software reuse and to simplify the creation of new software through composition of existing components [Cox90]. Although the term software component is in widespread use today, there is still a lack of common understanding. This is due to the fact that many people are working in this area on similar but nevertheless different issues. A look at some definitions of *software component* given by different researchers will illustrate just how broadly the term is used.

> A generalization of objects that extends the primitives for realizing interaction to include distributed components, graphical user interfaces, databases, robots, and virtual reality. [Weg93]
>
> A static abstraction with plugs. By static, we mean that a software component is a long-lived entity that can be stored in a software base, independently of the applications in which it has been used. By abstraction, we mean that a component puts a more or less opaque boundary around the software it encapsulates. With plugs means that there are well-defined ways to interact and communicate with the component. [NT95]
>
> Reusable software components are self-contained, clearly identifiable artifacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status. [Sam97]
>
> Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system. Composite systems composed of software components are called component software. The requirement for independence and binary form rules out many software abstractions, such as type declarations, C macros, C++ templates, or Smalltalk blocks. Other abstractions, such as procedures, classes, modules, or even entire applications, could form components, as long as they are in a binary form that remains composable. [Szy97]

In this thesis, we use the term *software component* to refer to a piece of software that exhibits well-defined interfaces, does not require knowledge of its implementation, is easier to reuse than to reimplement, and can be used for the implementation of other software systems. This definition is sufficiently strict and matches the general idea of most of the definitions above. A detailed glossary with the terms and their definitions used in this thesis can be found in Appendix A.

Although many software component models exist today [Ham97, MH00, OMG99a], it is still necessary to compose such components manually. Plug and play composition is not yet possible because two components can only interact with each other if the interface provided by one component matches the interface required by the other component.

Software components are sometimes compared to hardware components, and the implication is that the composition of software components should be as easy as that of hardware components. But experience has shown that even though hardware components can be plugged together, they do not necessarily interact, or "play" together, unless the operating system contains the appropriate device drivers. Similarly, software components cannot simply be plugged together if they do not have the same interface.

1.1 Contribution

Our approach to simplifying the adaptation and composition of software components is type-based adaptation. The goal of type-based adaptation is to provide an infrastructure that enables components provided by different suppliers to interact with each other transparently. Type-based adaptation can deal with different kinds of components. They can be traditional software components, components in a distributed component environment, or simple Web services, as long as the provided and required interfaces can be determined. On the basis of this information and an adapter repository, explained in Chapter 5, type-based adaptation facilitates the translation of an interface provided by one component to the interface required by another.

Besides type-based adaptation, other adaptation techniques exist that try to tackle different parts of the problem. Again, different techniques reflect the different goals that different software engineering researchers try to achieve. For instance, Generative Programming [BSST93] or Aspect-Oriented Programming [KLM+97] focus on the aspect level of a software component. Simplicissimus [SGML01], on the other hand, focuses on the performance optimization of a given composition. This thesis presents our classification of different adaptation approaches that clarifies the relationship among these research efforts. Such a classification has been lacking until now.

1.2 Motivation

Type-based adaptation can be used for various application domains such as dynamic distributed component systems [Grü00] which are the successor of dynamic distributed object systems, or builder tools such as Sun's Bean Development Kit [Sun98a]. Some application scenarios for these domains are illustrated by the following examples:

- Allowing shoppers at amazon.com to use an external address book service (for instance, provided by Yahoo or an application on the user's local computer). This would allow shoppers on the Internet to maintain a single address book instead of having to reenter billing and shipment addresses in all the different proprietary address books used by different Internet sites.

- Supporting users of an application builder to set up the connections between components. Instead of having to write application code, in many situations, it should be sufficient to configure a connection manager.

- In order to support backwards compatibility, the component has to provide both the old and new version of its interface. Typically, this leads to the entangling of legacy code necessary for providing backwards compatibility and code providing the component's functionality. Using type-based adaptation, however, the legacy code can be separated.

1.3 Outline

In Chapter 2 we present today's state-of-the-art component models and illustrate how their components fit the description of a software component. These component models range from simple desktop component models to server-side component models that can be used to implement services on the Internet.

Chapter 3 gives an overview of the adaptation techniques available today and the problems they try to address. These adaptation techniques range from scripting languages to performance optimizations.

On the basis of this overview, chapter 4 gives a classification of the adaptation techniques presented in the previous chapters. Although our classification is not yet comprehensive, it defines already key criteria. This puts the adaptation techniques into a coherent relation to each other and allows developers to decide which technique is suitable for their problems.

Chapter 5 presents type-based adaptation, its architecture and how it can be implemented for several application domains. Due to the differences in the component models used by different application domains, they pose different requirements. For

instance, in distributed component environments, component discovery occurs at run-time and frequently no developer is available. In the case of IDEs, however, components are selected and composed by a developer at the system's design time.

Chapter 6 describes the experiments we have used to verify the feasibility of type-based adaptation and to evaluate the benefits it provides. We have evaluated type-based adaptation with regards to local and distributed component environments as well as in the context of mobile agents.

Chapter 7 presents related work and how it influenced our type-based adaptation approach. Finally, we draw our conclusions in Chapter 8.

Chapter 2

Component Models

A component model defines the characteristics of a set of components. These characteristics usually include the packaging of a component to simplify its installation, the user-configurable parts of the component, and its interfaces.

One of the earliest component models is provided by Unix systems, though it is typically not regarded as such. Programs can be viewed as the components which process data read from `stdin` and return the result on `stdout`. The components are composed using pipes which themselves are created by the Unix shell. If the output of one component is different from the input expected by another component, it can be transformed using the stream editor `sed`.

In the following sections we limit our discussion to higher level component models. As we will describe, a component model can be suitable for either a local computing environment resembling a traditional programming model or for a distributed computing environment.

2.1 Libraries

Software libraries or parts thereof are the oldest well-known software components. Software libraries can be composed using the programming language for which they have been written. Frequently, such libraries also come with bindings for scripting, composition, and configuration languages such as Guile [Gal96], Perl [WCO00], Python [Ros99], or TCL [Ous94]. A more detailed discussion of these languages can be found in Sections 3.1.1, 3.1.2, and 3.2.1.

A library consisting of software components can be called a catalog. According to [Jaz95] catalogs and their components have to fulfill the following requirements:

Systematic Taxonomy: The components within a catalog must be closely related to a common application domain and adhere to a systematic taxonomy. This allows a user to identify easily whether the catalog contains a specific component.

Comprehensiveness: The application domain covered by a catalog needs to be covered comprehensively. Otherwise, programmers would have to reimplement the same functionality over and over again and would stop using the catalog.

Genericity: The catalog should try to minimize the number of components it contains while maintaining the same functionality. The fewer components the catalog provides, the easier it is to use.

Efficiency: Since components are likely to be used for the implementation of other components, it is important that they provide the best performance possible.

While each of these requirements sounds simple, it is hard to fulfill them all, especially since genericity frequently has a negative effect on efficiency and vice-versa. Another challenge is that many catalogs are efficient if used on their own, but their performance degrades when used in combination with other catalogs [SGML01]. Hence, it is necessary to understand the concepts of an application domain perfectly before developing a catalog for it. The catalog itself has to be useful repeatedly to make the effort of studying it worthwhile.

As explained in [Jaz95], components do not have to form an inheritance relationship to each other. In fact, many catalogs such as standard I/O libraries or C++'s Standard Template Library (STL) [Str97, ISO98] use little or no inheritance at all.

Today's component models, however, focus on large-grained application-oriented components that have to fulfill many more requirements. Although they employ an object-oriented model, inheritance is mostly restricted to the sole use of type-inheritance (sometimes also referred to as interface-inheritance).

2.2 JavaBeans

In [Ham97], a JavaBean is defined as a reusable software component that can be visually manipulated in a builder tool. A screenshot of such a builder tool, the Bean Development Kit [Sun98a] provided by Sun Microsystems, is shown in Figure 2.1. Although JavaBeans have been designed to be used within a builder tool, they can be used like any other Java class.

Any Java class that implements the `Serializable` interface and that provides a zero-argument constructor is a JavaBean. The first requirement simplifies the handling of beans within builder tools and the second allows a bean to be instantiated using `Beans.instantiate()`. Neither of these requirements, however, is enforced by the JavaBeans component model.

The most important features of JavaBeans are the exposed set of properties, the support for a set of events, and the set of methods they provide. This information is either exposed implicitly by naming conventions and reflection [Sun98b] or by a

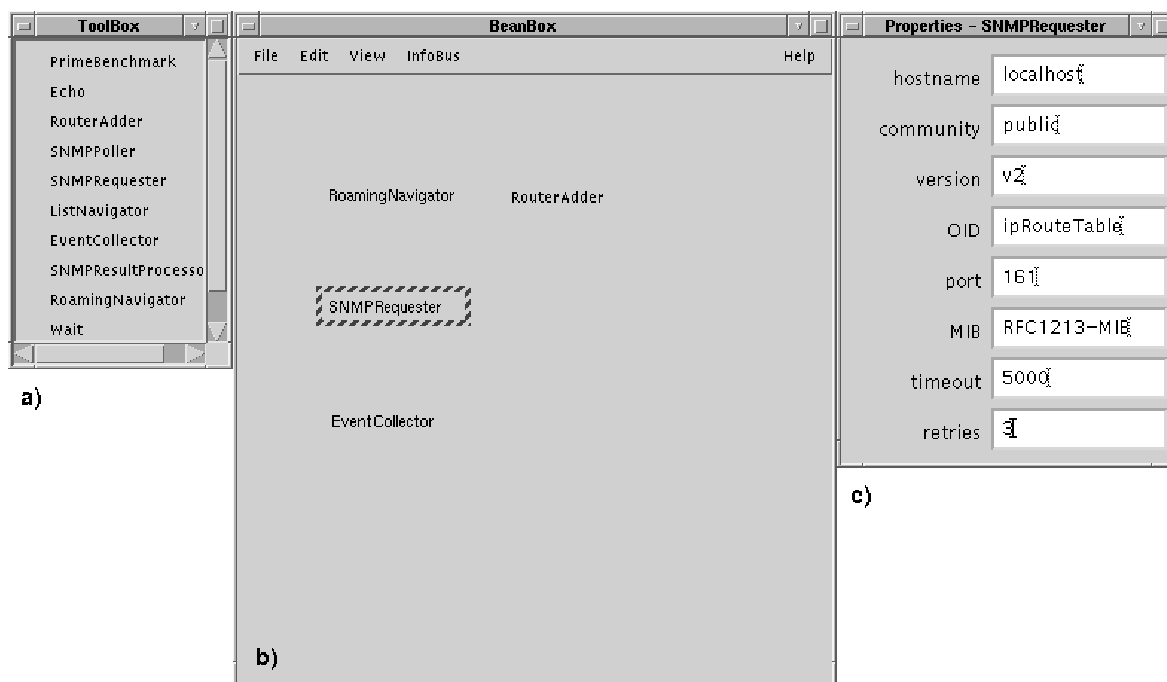


Figure 2.1: The AgentBean Development Kit with a) the Available JavaBeans, b) the Composition Area, and c) the SNMPRequester's Property Sheet

```
1  package at.ac.tuwien.infosys.examples.jb;
2
3  class DrawingArea implements Serializable {
4    private Color _color;
5    private boolean _drawing;
6
7    /* color property */
8    public void setColor(Color c) { _color=c; }
9    public Color getColor() { return _color; }
10
11   /* pen is drawing */
12   public void setDrawing(boolean b) { _drawing=b; }
13   public boolean isDrawing() { return _drawing; }
14
15   /* mouse listener */
16   public void addMouseListener(MouseListener l) { /* ... */ }
17   public void removeMouseListener(MouseListener l) { /* ... */ }
18 }
```

Figure 2.2: A Simple JavaBean

BeanInfo class that provides such types of meta-information (see Section 2.2.4). A sample JavaBean is shown in Figure 2.2.

2.2.1 Properties

Properties are named attributes associated with a JavaBean that can be read and written using a set of getter and setter methods. Properties can also be marked read-only or write-only by specifying only the setter or getter methods.

`void set`*`Property`*`(`*`Type`*` p)`
: Write a property with name *property* and of type *Type*.

`Type`` get`*`Property`*`()`
: Read a property with name *property* and of type *Type*.

`boolean is`*`Property`*`()`
: Read a boolean property with name *property*. This can be provided in addition to or instead of the `get`*`Property`* method.

`void set`*`Property`*`(int i, `*`Type`*` v)`
: Write the i-th element of an indexed property with name *property* and of type *Type*.

`Type`` get`*`Property`*`(int i)`
: Read the i-th element of an indexed property with name *property* and of type *Type*.

Table 2.1: Standard Methods to Read and Write Properties

Properties can be identified transparently if the getter and setter methods adhere to the JavaBeans naming convention as shown in Table 2.1. Otherwise, these properties and their setter and getter methods have to be specified within the JavaBean's `BeanInfo` class. Properties that can take a whole array of arguments are called *indexed properties*. Additional methods can be provided for such properties that read and write only a specific element of the array.

Sometimes it can be useful to notify other components if a property changes. To provide that functionality, a JavaBean can provide a change notification service to other components by emitting a `PropertyChangeEvent` whenever a property changes. Typically, such properties are referred to as *bound properties*. If other components are allowed to veto such a property change, the property is referred to as a *constrained property*.

When a JavaBean is used in a builder tool, properties can be edited and changed visually. For instance, the properties of the *SNMPRequester* bean are shown in Figure 2.1 on the right-hand side. The editor to be used for a property is managed by the `java.beans.PropertyEditorManager` class, which derives the editor to be used from the property's type.

2.2.2 Events

Events provide a way for components to be plugged together by application builders. Whenever the state of a component changes it can trigger an event. This event can be caught by another component which can react to the event.

`void KeyPressed(KeyEvent e)`
This method is invoked when a key has been pressed.

`void KeyReleased(KeyEvent e)`
This method is invoked when a key has been released.

`void KeyTyped(KeyEvent e)`
This method is invoked when a key has been typed.

Table 2.2: `KeyListener` Interface

The JavaBeans component model collects a set of related events within a single listener interface. For instance, the `KeyLister` interface shown in Table 2.2 contains all the events associated with keyboard input. This interface has to be implemented by any component that wants to subscribe to any of these events.

If a component does not implement the interface, an adapter class has to be provided that implements the listener interface and invokes the corresponding method on the target component. Typically, such an adapter class is generated by the application builder after the user has selected the event and the method to be invoked on the target component. Unfortunately, the adapters generated are very simple and can only call target methods that match the event's signature or target methods that take no parameters at all. In the latter case, the event's parameters are discarded. This problem, however, can be easily solved using type-based adaptation. Type-based adaptation will be presented in Chapter 5.

A component that can emit a specific event has to provide methods to allow other components or classes to subscribe to the reception of the events within an event set. These methods are shown in Table 2.3. Typically, multiple components can subscribe to receive notification of events. If the subscription method can throw the

`void add`*Foo*`Listener(Listener l)`
This method adds the listener `l` to the bean's *foo*-event listeners.

`void remove`*Foo*`Listener(Listener l)`
This method removes the listener `l` from the bean's `Foo`-event listeners.

Table 2.3: Standard Methods to Subscribe to and to Unsubscribe from Events

`TooManyListenersException`, however, it is assumed that only a single class or component can subscribe to the event set.

2.2.3 Methods

Methods are like any other Java methods which can be called from other components. The methods of a bean do not have to adhere to any naming conventions. If only a subset of methods should be made available, the methods provided by a bean can be listed within the `BeanInfo` class.

2.2.4 Introspection

In addition to using the naming conventions presented so far, the JavaBeans component model allows properties, events, and methods to be listed explicitly in a `BeanInfo` class. If this class is provided, it will be used instead of deriving the properties, events, and methods from the naming conventions. Using a `BeanInfo` class allows the developer of a JavaBean to provide additional information such as an icon to represent the JavaBean or a specific customizer that can be used to configure the JavaBean. The `BeanInfo` class has the same name as the class it describes plus `BeanInfo` appended to it. The complete description of the API of this class, is given in [Ham97]. A sample JavaBean along with its `BeanInfo` class is shown in Appendix B.1.

An advantage of using separate classes for the meta-information and all of the property editors and customizers is that those helper classes only need to be available at design time, but not after the application has been generated by the development environment.

2.3 Enterprise JavaBeans

While JavaBeans are intended for local component environments, Enterprise JavaBeans are intended for distributed component environments. Even though, remote method

invocation (RMI) [Sun99a] can be used for the implementation of distributed application environments, it burdens the programmer to implement repeatedly such common services as support for security, persistence, and transaction management. To overcome this situation, Sun Microsystems introduced the Enterprise JavaBeans (EJB) component model that supports these services natively [MH00, DYK01].

EJBs allow the developer to focus on the application logic without having to deal with the typical issues present in distributed applications. Besides increasing productivity, EJBs also increase the interoperability of distributed applications, since each Enterprise JavaBean uses the same type of transaction management. The EJB component model has been adopted by many commercial and non-commercial implementors such as BEA's WebLogic server [BEA01] or the JBoss group's JBoss server [JBo].

2.3.1 Architecture

Figure 2.3 shows the architecture of the EJB component model. Each Enterprise JavaBean is hosted by its own EJB container running within an EJB server. The container acts as a proxy for the bean and thus is able to provide services such as security, persistence, and transaction management. Within one container, however, multiple instances of the same bean may exist.

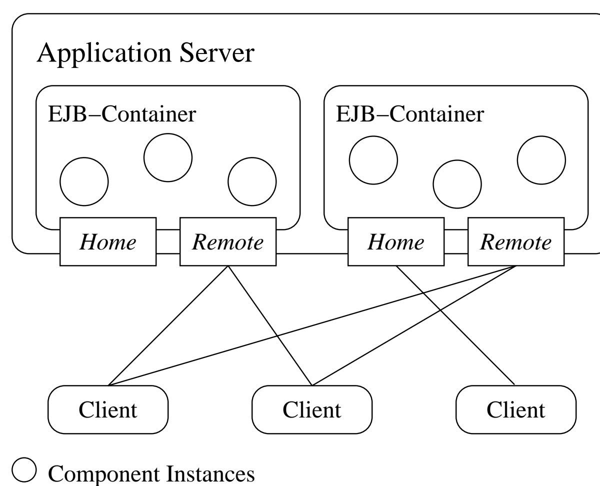


Figure 2.3: The Enterprise JavaBean Architecture

Depending on the functionality provided by the bean, we can differentiate between *session beans*, *entity beans*, and, since EJB 2.0, *message-driven beans*. Session beans can be seen as an extension of the client application, entity beans provide an object-oriented view of data stored on persistent storage, and message-driven beans are asynchronous message consumers. As we will describe in the following section, however, a typical client interacts only with session beans.

On the client side an EJB is used like an RMI object, except that instead of a native cast, the `PortableRemoteObject` is used to cast a remote object. If the client does not know the methods provided by a bean, it can also use the `javax.ejb.EJBMetaData` interface to query a bean's capabilities. This functionality allows application builders to discover information about a bean and to present it to the developer. Since several clients can access the same EJB concurrently, the EJB server synchronizes the state of these EJBs as necessary.

2.3.2 Beans

Session and entity beans both have to provide a *home interface* which acts as a factory to create and remove instances of the corresponding bean and a *remote interface* which declares the business functions (methods) that can be executed by the clients. Message-driven beans, however, do not have to implement these interfaces because they directly react to messages received by the Java Message Service, JMS [HBS99].

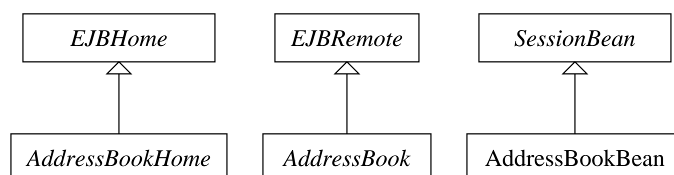


Figure 2.4: UML Diagram of an EJB Session Bean

While a bean's home interface has to extend `javax.ejb.EJBHome`, a bean's remote interface has to extend `javax.ejb.EJBObject`. The relation between these classes is shown in Figure 2.4 for a session bean. For entity beans, this relation looks analogously.

Session Beans

Session beans model a workflow and can be seen as an extension of the client application. They are responsible for the management of processes and tasks. For instance, a `ShoppingCart` bean might provide all the functionality necessary to select and buy goods from an Internet super-store. Session beans are short-lived; their instances of session beans are created and removed by their clients. A session bean's home interface extends the `javax.ejb.EJBHome` interface and provides methods to create and remove beans as shown in Table 2.4.

The remote interface of a session bean has to extend `javax.ejb.EJBObject` and declares the methods supported by the session bean. These methods can be executed by clients having a reference to an instance of the bean.

The implementation class of the bean implements the `javax.ejb.SessionBean` interface. While the implementation class implements the methods declared in the

`Remote createMethod(`$arg_1, arg_2, \ldots, arg_n$`)`
: This method allows clients to create new beans. The arguments have to be legal types for RMI-IIOP. These methods are mapped onto the corresponding `void ejbCreateMethod` methods of the bean's implementation class.

`void remove(Handle handle)`
: This method allows the client to remove a bean identified by a specific handle. This method is mapped onto the `void ejbRemove()` method of the bean's implementation class.

Table 2.4: Methods Declared in the Session Bean's Home Interface

remote interface, it does not have to declare the remote interface in its `implements` clause. Additionally, the bean provides the implementation for the methods declared in the home interface as well as the callback methods shown in Table 2.5 for the bean's persistence management. If no special serialization is required, the implementation of these methods may be left empty.

`void ejbPassivate()`
: This method is called before the container passivates a session bean to disk. It serves as a hook for state that cannot be serialized using the standard serialization mechanism.

`void ejbActivate()`
: This method is called after the container has been activated and is used to initialize state that cannot be handled using the standard serialization mechanism.

Table 2.5: Methods Declared in the `SessionBean` Interface

Session beans can be stateless or stateful. Stateful beans maintain a conversational state between client calls. Depending on whether a state needs to be preserved between client calls, the EJB server applies a different pooling algorithm to the instantiated beans. While stateless beans can be reused for the execution of another client call, stateful beans can only be reused after the client has released the session bean. An example of a stateful session bean is shown in Appendix B.2.1.

Entity Beans

While a session bean can be seen as the extension of a client and executes on behalf of a single client, an entity bean provides an object-oriented view of data stored in a database. Although the data encapsulated by the entity bean can be accessed from several different clients, they are typically accessed by session beans taking care of the application's business logic. Each entity bean is identified by a *primary key* and its data consistency is guaranteed by its container.

An entity bean can be either container-managed or bean-managed. Although the database fields of a container-managed bean are handled by the container, the database fields of a bean-managed bean need to be handled by the bean itself. Since the same principles apply to both types of entity beans, we restrict ourselves to the discussion of container-managed entity beans. A more detailed discussion of entity beans can be found in [DYK01].

The entity bean's home interface has to provide the methods to create, remove, and find an entity bean, as shown in Table 2.6. Additionally, the home interface of an entity bean may declare home methods (static methods) that are not specific to an instance of an entity bean. These methods are prefixed by `ejbHome` in the bean's implementation class and must not start with `create`, `find`, or `remove`.

The remote interface of an entity bean has to extend `javax.ejb.EJBObject` and declares the methods supported by the entity bean. These methods can be executed by clients having a reference to the bean.

The implementation class of the bean implements the `javax.ejb.EntityBean` interface. Additionally, the implementation class may be abstract, if the accessor methods for the bean's properties should be provided by the bean's container. While the implementation class implements the methods declared in the remote interface, it does not have to declare the remote interface in its `implements` clause. Additionally, it provides the following methods:

- The implementation of the methods declared in the home interface as indicated in Table 2.6.
- The declaration of the accessor methods for the bean's container-managed persistent fields and relationship fields using the JavaBeans [Ham97] naming conventions for properties. These methods must be `public`. They must also be `abstract` if their implementation should be provided by the bean's container.
- The implementation of the callback methods necessary for the persistence management (Tables 2.5 and 2.7). If no special persistence management is required, the implementation of these methods may be left empty.

An example of a container-managed entity bean is shown in Appendix B.2.2.

`Remote createMethod` $(arg_1, arg_2, \ldots, arg_n)$
: This method allows clients to create new beans. The arguments have to be legal types for RMI-IIOP. These methods are mapped onto the corresponding `void ejbCreateMethod` and `void ejbPostCreateMethod` methods of the bean's implementation class.

`Remote findByPrimaryKey(KeyType primKey)`
: This method looks up an entity bean on the basis of its primary key. This method must be present for all entity beans and must not be overloaded. The implementation for this method is generated by the EJB server.

`Remote findMethod` $(arg_1, arg_2, \ldots, arg_n)$
: This method looks up an entity bean on the basis of the arguments arg_i. The arguments must be legal RMI-IIOP types. The code for these methods is generated by the EJB server on the basis of the queries provided by the bean's deployment descriptor.

`Collection findMethod` $(arg_1, arg_2, \ldots, arg_n)$
: This method looks up a set of entity beans on the basis of the arguments. The arguments must be legal RMI-IIOP types. The code for these methods is generated by the EJB server on the basis of the queries provided by the bean's deployment descriptor.

`void remove(KeyType primKey)`
: This method removes an entity bean on the basis of its primary key. This method is mapped onto the `void ejbRemove()` method of the bean's implementation class.

`void remove(Handle handle)`
: This method allows the client to remove a bean identified by a specific handle. This method is mapped onto the `void ejbRemove()` method of the bean's implementation class.

Table 2.6: Methods Declared in the Entity Bean's Home Interface

`void ejbLoad()`
: This method is called when the container needs to synchronize the state of a bean. When the method is being called, the instance's persistent state has already been updated by the container. This method allows the bean to update computed values that depend on database data.

`void ejbStore()`
: This method is called before the database data will be updated on the basis of the bean's persistent state. The bean, however, should use the accessor methods when updating the persistent state.

Table 2.7: Additional Methods Declared in the `EntityBean` Interface

Message-Driven Beans

Message-driven beans have been added in the EJB 2.0 specification for the integration of the Java Message Service, JMS [HBS99]. Message-driven beans are similar to session beans but simplify the development of an EJB that is asynchronously invoked to handle the processing of an incoming JMS message. A client interacts with a message-driven bean by sending a message to a JMS destination for which the bean is a listener. Since a message-driven bean is never accessed by a client directly, it has neither a home nor a remote interface.

A message-driven bean must implement the `javax.ejb.MessageDrivenBean` and the `javax.ejb.MessageListener` interfaces. Thus, it has to implement the methods shown in Table 2.8. An example of a message-driven bean is shown in Appendix B.2.3.

2.3.3 Packaging and Deployment

Enterprise JavaBeans are packaged into a `.jar`-file. This file consists of the bean's implementation classes as well as the bean's deployment descriptor. The deployment specifies the home and remote interfaces of a bean as well as the mapping of the entity bean's find methods to the corresponding database queries. Sample property descriptors for all three types of beans are shown in Appendices B.2.1–B.2.3.

Deployment is the Enterprise JavaBeans term for installing the Enterprise JavaBeans components into an Enterprise JavaBeans container. Typically, an EJB is employed by placing its `.jar`-file into a deployment directory. Afterwards, the EJB server reads the deployment descriptor(s) and installs the beans accordingly.

`void setMessageDrivenContext(MessageDrivenContext ctx)`
: Sets the context of the bean.

`void ejbCreate()`
: This method is called after a new instance of the message driven bean has been created.

`void ejbRemove()`
: This method is called before the bean is removed by the container.

`void onMessage(Message msg)`
: This method is called after the arrival of a JMS message.

Table 2.8: Methods Declared in the Entity Bean's Home Interface

2.4 The CORBA Component Model (CCM)

CORBA focuses mainly on the description of interfaces and services. The CORBA Component Model (CCM) [OMG99a, OMG99b, OMG99c], however, takes the next step and adds the ability to define CORBA Components. The CCM provides an Abstract Component Model, a Packaging and Deployment Model, a Container Model, a mapping to the EJB component model, and a model for the integration of persistence and transactions. It specifies server-side components that may be used by clients or other servers, hence allowing a distributed enterprise computing architecture. Unfortunately, however, no implementation of the CCM exists so far. The following discussion is therefore limited to the conceptual level of the CCM.

CORBA provides a myriad of features for building enterprise-scale applications. A few patterns for building these applications, however, have widespread applicability. If applications are built using these patterns, much of the work can be solved by the design tools defined by the CCM (similar to the stub generation of IDL compilers). While the CORBA component model is similar to the EJB component model, its key advantage is language independence.

The typical use model is as follows:

Analysis/Design Phase: The CCM does not provide any assistance for this task.

Component Declaration: CORBA Components are defined using an extended IDL. The IDL generates the stubs and meta-data comprising the client view of the components.

Component Implementation: The Component Implementation Definition Language (CIDL) is used to describe the server part of the component. It fulfills a similar role to IDL but from the component implementer's view. It enables platform- and language-independent specification of features such as transactions, persistence, and events.

Component Packaging: A component archive contains the component implementation and a component descriptor.

Component Assembly: This stage customizes the component and connects it to other components. It is a description of a collection of prototypical components along with their relationships. This task might be performed using a visual composition tool.

Component Deployment and Installation: Typically, a tool reads a component archive or an assembly archive and installs the required components. The result is a set of installed components.

Instance Activation: Once installed, the components are ready to be instantiated using the standard CORBA ORB activation mechanism.

2.4.1 Component Specification

The CCM supports two levels of components: *basic components* and *extended components*. While basic components are just a mechanism to componentize ordinary CORBA objects, extended components make use of all the features provided by the CCM.

Each component type has a *component home* that acts as its manager. Primary keys may be associated with components by a component home. In the CCM, primary keys are a means for clients to identify component instances and obtain references to them. At execution time, a component instance is managed by a single home object.

A component is a collection of specific named features that can be described by an IDL component definition and/or a corresponding structure in an Interface Repository. The CCM describes a variety of these features:

Facets: A component may have multiple interfaces, each represented by a different object reference. The advantage of facets is that they make possible the modularization of components. If only a single facet is required by a client, only the corresponding object has to be activated. CORBA components, however, still have a single distinguished reference whose interface conforms to the component definition and supports the component's equivalence interface.

Receptables: Receptables help to describe the connections between components. Receptables may be simplex managing single or multiplex managing multiple object references.

Event Sources and Sinks: The CCM supports a publish subscribe model which is a subset of CORBA's notification service. Event sources may be publishers and emitters. Publishers have multiple subscribers and are the only publishing source. Emitters have only one subscriber and listen to possibly several emitting sources. An event sink describes the potential for a component to receive events of a specified type. Event sinks do not differentiate between subscription (publish) and connection (emit).

Primary Key: A component may expose a primary key which may be used by clients to locate, create, and destroy the component instance associated with that primary key.

Home Interfaces: When a component type is deployed, an object called *Home* is created which manages instances of that component type. The home objects provide factory and finder operations needed to create and look up a component instance plus any number of type-specific factory/finder operations defined.

Attributes and Configuration: The CCM supports the notion of dividing the component life cycle into two different phases: the *configuration* and the *operational* phase. The CCM provides features that enables the developer to distinguish between features intended for the configuration or the operational phase. Since this distinction is somewhat arbitrary, however, the enforcement is left to the user.

Inheritance: The CCM uses the generic `CCMObject` interface to aggregate the generic `Navigation`, `Receptable`, and `Events` interfaces and the `CCMHome` interface to specify the capabilities of the component's home object. The type-specific interfaces that correspond to a specific component definition are generated on the basis of the rules for component inheritance.

Components belong to one of the following categories:

Service: This component has only behavior. It is useful for tasks that require only the single independent execution of an operation.

Session: This component has behavior and a transient state. Its state is preserved while the client interacts with the component.

Process: A process component has a behavior and a persistent state not visible to its clients. It is useful for modeling business processes rather than entities.

Entity: An entity component has behavior, a persistent state, and an identity that can be accessed via its primary key. It is useful for modeling real things (e.g., customers or accounts).

Persistence and transaction management can either be provided by the CCM or be implemented by the component itself. After the component has been implemented, the component has to be compiled into a component package that may be loaded into an application or into a tool (e.g., an application builder) and used to construct a component that becomes a package itself.

2.4.2 Component Implementation

The Component Implementation Definition Language (CIDL) is used to describe the implementation of the components and their homes. On the basis of that description, the CIDL compiler generates the skeletons that automate many of the basic behaviors of a component such as navigation, identity, inquiries, activation, and state management. Finally, the component builder extends the skeletons to implement the application.

CIDL is a superset of the Persistent State Definition Language (PSDL). PSDL is used to declare the persistent state of an object to enable the persistent state service to transparently store and retrieve objects. CIDL adds support the specification of an association between an abstract storage type and the form of an internal state encapsulated by a component. The Component Implementation Framework (CIF) and the container then cooperate to manage the component's persistent state automatically.

The following terminology is used by the CORBA component model:

Executer: The programming artifact used to implement the component is denoted as *Executer*. A monolithic executer is made up of a single programming artifact and a segmented executer is made up of several programming artifacts, each representing a different part of the component's state. Typically, each part corresponds to a different facet. This enables a request to a facet to be serviced by bringing up just that facet of the component.

Executer Definition: An Executer Definition defines the name of an Executer, the segmentation it consists of, the generation of operation implementations managing stateful features such as receptables, and a delegation declaration to describe the relationship between particular stateful features.

Composition: A component implementation is made up of several different elements such as the Component Home, the Home Executer, and the Component Executer as well as optionally an abstract storage home binding, a delegation specification, and a proxy home.

Composition Structure: A minimal composition consists of a name for the composition, the component's category (service, session, process, or entity), the home type, a name for the generated home executer, and a name for the generated component executer.

2.4.3 Container Model and Architecture

A component server is a process which provides an arbitrary number of component containers. A container manages a specific component category and provides its runtime execution environment. While external API types define the contract between the component and its clients (IDL), container API types define the internal interfaces and callback interfaces used by the component developer (CIDL).

The container is a server-side framework built on top of the ORB, the Portable Object Adapter (POA), and a set of CORBA services. The interaction with the ORB, POA, and CORBA services is defined by the CORBA usage model.

The usage model is controlled by policies which select between interaction patterns with the POA and other CORBA services. It is defined in CIDL and augmented using XML. The CCM predefines three different models which differ in their interactions with the POA. The *stateless model* uses transient references with a POA server that can support any object. The *conversational model* also uses transient references but with a POA dedicated to a specific object. The *durable model* uses a persistent reference with a POA dedicated to a specific object.

After a component has been deployed into a CCM server, the server creates a container for the component. The container itself is created by a container manager determining the appropriate set of POA policies, a container API type, and a set of CORBA service bindings to be used by the container. All this information forms the container specification. Container managers are themselves created as part of the installation and deployment process. Figure 2.5 depicts this architecture.

The CCM defines seven different container categories (API types): four corresponding to the four container categories, two for EJB container API types and an empty container to support user-defined frameworks:

Service manages access to stateless components.

Session manages the session of components with a transient state.

Process manages stateful process components which encapsulate data access in the server.

Entity manages stateful entity components which share data access responsibility between the client and the server.

EJBSession manages EJB session beans.

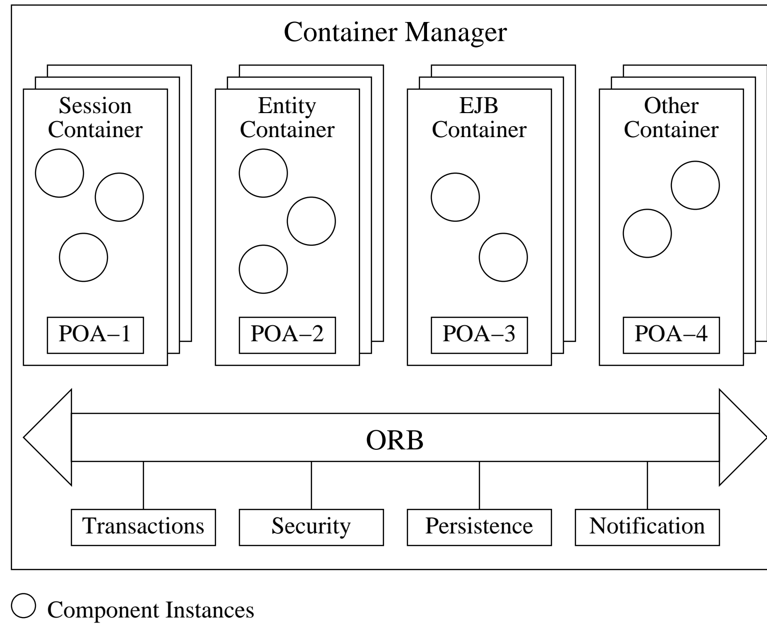


Figure 2.5: The CORBA Container Architecture

EJBEntity manages EJB entity beans.

Empty provides no automatic management, but makes the standard CORBA 3.0 interfaces available to the component implementation.

As for components, the CCM supports two levels of component containers: basic and extended. However, this affects only object reference management and the availability of supporting CORBA services. For both levels of containers, references to these services are obtained using `resolve_initial_references`.

Containers also support different component thread safety policies: serialized and multithreaded. Basic components, however, only support the serialized threading policy.

Basic containers can use CORBA's security, transaction, and naming service. For basic CORBA component containers and the EJB container API types, the container provider must manage the object reference creation. The containers provide access to the factory and finder operations declared in IDL and interact with the Transaction Service on behalf of the component. The containers rely on CORBA security to implement access policies and apply it to all the container's components.

Extended containers can also use the notification service as well as the persistent state service. Extended containers are responsible for mapping the extended events of a component (defined in IDL) to the CORBA Notification service and for delivering the specified event types via a notification channel. Containers are responsible for setting

up the channels, accepting a component event, pushing it to a channel as a structured event, and vice-versa (receiving a structured event and converting it to a component event).

Persistence is supported by the process, entity, and EJBEntity containers which provide access to the CORBA persistent state service in order to allow the component developer to implement self-managed persistence. Entity containers also support a `get_primary_key` operation.

Persistence may be managed by either the container (*container-managed*) or the component itself (*self-managed*). In the case of self-managed, persistence the component cooperates directly with the persistence provider. In the case of container managed persistence, however, the container provider cooperates with the persistence provider, and the component only provides the functions for loading and storing the component's state as well as the factory and finder operations. These can be generated automatically, if the component's state is described using the Persistent State Definition Language (PSDL).

2.4.4 Client Programming Model

Clients do not have to be component-aware. No matter whether they are component-aware or not, however, they resolve their initial references using `resolve_initial_references`.

Component-unaware clients interact with CORBA components using the home interface or one of the application interfaces but do not profit from the additional features defined by the CCM. The home of a component either creates a new component or, if the component has a primary key assigned to it, locates an existing component via a finder.

Component-aware clients may use the features introduced by the CCM. They can obtain references for the NameService, TransactionCurrent, SecurityCurrent, NotificationService, InterfaceRepository, and the HomeFinder services. They also can navigate among multiple interfaces and emit and consume events using the component APIs defined in the CCM. Security is provided via the CORBA security mechanism.

2.4.5 Component Assembly and Packaging

A component package is the vehicle for deploying a single component implementation. It includes one or more implementations of a component. Each implementation implements the same component, but with characteristics that can differ in the implementation language, operating system, or even runtime behaviors. In general, it consists of a set of files and one or more XML descriptors containing the package's characteristics and their dependencies. The collection of files and descriptors may be grouped together into a ZIP archive file or kept separately.

A component assembly package is the vehicle for deploying a set of interrelated component implementations. It consists of a set of component packages and an assembly descriptor which gives the components, partitioning constraints, and connections. It is a template or pattern for instantiating a set of components and homes, and for introducing them to each other.

2.4.6 Component Deployment

Components are installed on the target hosts using a deployment tool. The deployment tool interprets the component's deployment descriptor, selects the appropriate component implementation, initializes the component's properties and connects it to other components. Much of this process, however, is left over as an implementation issue for tool vendors.

2.5 Summary

On the basis of the explanations in the previous sections, we can identify two different types of component models: desktop (or local) component models that allow the specification of components for traditional software engineering and server-side (or distributed) component models suitable for distributed computing environments such as provided by the Internet.

2.5.1 Desktop Component Models

Desktop component models provide software components designed for inter-operation within the same address space. Components in the form of traditional program libraries are ideal for software development with little or no tool support. Hence, components of desktop component models are not distributed and are all available on the same computer.

More advanced component models such as provided by JavaBeans, however, come with a meta-description of the software component. This meta-description identifies features (or characteristics) of the component models such as the available properties, events, or methods. On the basis of this meta-description, application builders can present developers with a list of available components as well as their features. Hence, they can guide developers in the creation of new applications.

After the developer has finished defining the application, the application builder links all the components employed by the user into a single program. In that sense, desktop component models are very similar to software libraries. The program generated by the builder does not require any services from another computer as long as the components employed are not internally dependent on such a service.

2.5.2 Server-Side Component Models

Server-side component models, on the other hand, describe components that provide a specific service that can be requested from clients located on a different computer. Server-side component models only provide a framework for the description of a service. The framework then takes care of tasks such as persistence management, transaction management, or the provision of the appropriate security services. These component models, however, do not facilitate the programming of clients using such a service.

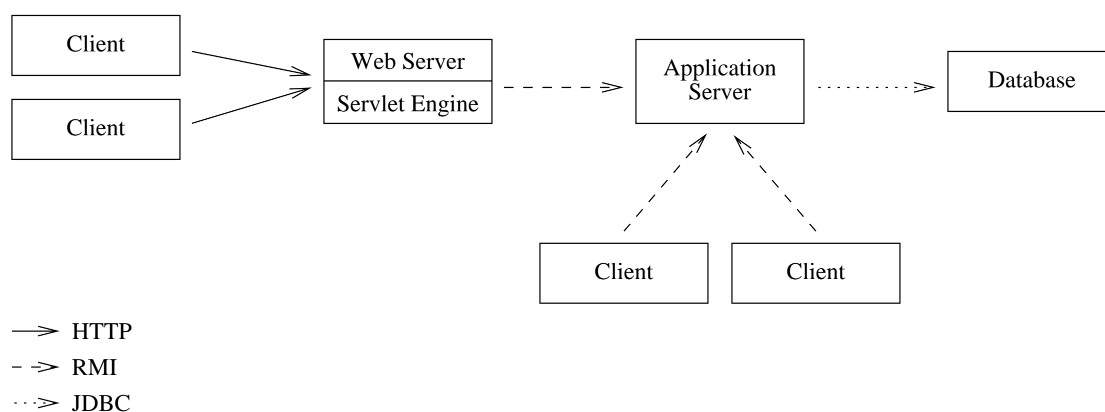


Figure 2.6: Typical Web Service Architecture

Server-side components are most often used as application servers that can be accessed by clients and Web services. A sample such architecture is shown in Figure 2.6. While this figure is based on the EJB component model, the principle applies equally well to the CORBA component model.

The advantage of this architecture is that customers can access the service (for instance, a banking application) via a Web interface provided by the servlet engine. On the other hand, employees can interact with the banking service using a native interface. Typically, native applications provide better performance and are not limited to HTTP's pull model.

While the above architecture simplifies program development, it needs improvement from a performance point of view because of the additional network delay arising from the separation of the servlet engine and the application server. For performance reasons, most application servers can be executed within the same address space as the servlet engine. This is why BEA WebLogic or JBoss are each shipped with a servlet engine [BEA01, JBo]. In this case, the RMI call can be replaced with a local method call.

A drawback of the EJB or the CORBA component models is that they do not provide any support for the implementation of the Web service. Hence, the application has to be implemented twice: once as a Web service and once as a native application.

Chapter 3

Adaptation of Components

Although software components are in widespread use today, they are still used in ways much like program libraries. This might be due to the fact that program libraries are a kind of software component. The goal of today's component technology, however, should be the simplification of the composition of components. Such components are frequently referred to as *Commercial Off The Shelf* (COTS) components. So far, the composition process has only been simplified in the rare case that the interfaces of the components match each other. Whenever this is not the case, the composition still has to be performed programmatically by manually implementing the necessary glue code.

The composition process could be simplified if components could be adapted more easily. This is the focus of many current research projects which we present in the following sections. We categorized the projects into those focusing primarily on the composition of components, on the separation of concerns, and on the optimization of systems already built from components.

3.1 Adaptation for Composition

In the following sections, we present adaptation techniques that focus on the composition of components providing a specific functionality, such as an address book or a spell checker.

3.1.1 Scripting Languages

The use of scripting languages, sometimes incorrectly referred to as composition or configuration languages, is one of the oldest approaches used for the composition of software components. While the components themselves are written in traditional languages such as C or C++, the scripting language is used to plug the components together.

Scripting languages are similar to traditional programming languages except that they are typically interpreted instead of being compiled. Since scripting languages are interpreted languages they are easier to use. Code can be tested immediately and debugged more quickly, thus allowing a rapid prototyping approach. The programs do not have to be recompiled after each bug fix and memory accesses are checked by the interpreter which makes bugs immediately visible at run-time and hence, avoids debugging nightmares encoutered in compiled languages such as C [Ous98]. Components are made available to the scripting language dynamically by loading shared libraries or statically by linking the interpreter with the component libraries.

The developer uses the scripting language only to specify how the components interact with each other and how data structures exchanged between the components have to be transformed. While scripting languages could be used to write application code, they should only be used for small functions supplying application logic missing in the original components. Writing a whole application with scripting languages is discouraged because code written in scripting languages is frequently less structured and less readable.

Although scripting languages are easy to use, they have certain disadvantages, too. They are less efficient than compiled languages, such as C, which usually is of major importance for number-crunching components, gaming environments, or other components operating on huge data sets. Scripting languages do not provide low-level functionality that might be necessary for the implementation of a database component, for instance. Additionally, scripting languages use a dynamic type system which is typical of interpreted languages and defers the detection of type errors until execution time. Thus, components themselves are rarely implemented using scripting languages.

Today, many COTS components come with bindings for many popular scripting languages. Since different developers use different scripting languages, however, it is important for component developers to provide bindings for their components for all the different scripting languages. Fortunately, this process can be somewhat simplified using generators such as SWIG [Bea96]. SWIG reads in a C or C++ header file declaring the functions that need to be made available and generates the appropriate bindings. In some cases, however, it is still necessary to implement additional code for a finer-grained access to the component's data structures.

Unix Shells

The various kinds of Unix shells such as the Bourne shell or C shell are the predecessors of today's scripting languages. These shells provide environment variables to store state and provide simple control constructs such as `if`-statements or `while`-loops. Additionally, they allow other programs to be executed and the output from one program to be forwarded to the input of another program.

On the basis of that functionality and in combination with text manipulation programs such as `sed` (the stream editor), Unix shells can be used to compose existing

programs into more powerful ones. As an example, many programs executed at system start-up or when the user logs in are still written using shell scripts.

Perl

Perl [WS91, WCO00] is one of the oldest scripting languages and is still used by many developers. Perl 1.0 was posted to the USENET newsgroup comp.sources in 1987 and the first book [WS91] on Perl was published by O'Reilly & Associates in 1991. Compared to shells, Perl provides more data types and a wide variety of commands to match and operate on strings. This makes Perl the perfect choice for analyzing and manipulating textual data. Typically, such tasks are present in Web applications that handle an HTTP client request, access a database component, and return the result back as an HTML document. Perl has been criticized, however, for its syntax which is concise but cryptic and incomprehensible.

Tcl/Tk

Tcl [Ous94] is a very simple scripting language that operates solely on strings. For simplicity reasons, other data types have been removed from the language. Even numbers are stored using a string representation and converted to their binary representation whenever necessary. Tcl provides elaborate string manipulation functions which make it easy for the developer to use.

Tcl also provides a simple API that allows the user to write bindings for components written in C or C++. An extension of Tcl is Tk, which provides a component to create complete user-interfaces with buttons, text-boxes, and other frequently used widgets. Typically, Tk is bundled together with Tcl and is commonly known under the name Tcl/Tk. The reason for the success of Tcl/Tk seems to be its ease of use compared to Perl and the fact that it was the first scripting language that allowed to creation of user interfaces. Even many newer scripting languages such as Python use Tk underneath for the actual handling of the user interface.

3.1.2 Composition Languages

Object-oriented programming languages while focusing on the object level, fall short of providing a compositional view of an application [NM95]. Composition languages, however, operate on a higher level of abstraction, hence providing a framework allowing the composition of components. Conceptually, a composition language must lie between Smalltalk and Perl.

A composition language must be flexible enough to cope with both *objects* and *components*. Components must be pluggable, so the interfaces that a component provides or requires need to be specified. Since components are frequently distributed, a composition language must be able to view objects as processes. This requirement rules

out the use of the λ calculus as the formal basis for composition languages. According to [NM95, SL96, LSNA97], however, the π calculus, a calculus of *mobile processes*, is a good choice for the formal basis of composition languages.

Initial experimentations using PICT, a general-purpose programming language based on the π calculus, as a composition language were presented in [SL96]. While PICT was a good choice for initial experimentations, it was too restrictive for more sophisticated examples. Hence PICCOLA, a new composition language, has been presented in [LSNA97, ALSN01]. PICCOLA is a very small language which is able to support a variety of component models through the definition of different component composition styles [ALSN01]. While PICCOLA allows the composition of components the adaptation of the components still has to be done programmatically. So far, PICCOLA seems to be the only composition language available based on a formal model.

Another composition language, the Bean Markup Language (BML), has been presented in [WCD+01]. BML is a declarative language based on XML and allows developers to specify the composition of JavaBeans components. BML supports the configuration of JavaBean properties and the specification of the glue code to be executed in terms of scripts. BML supports both the definition of applications and new JavaBean components. An approach similar to BML is the Long-Term Persistence for JavaBeans specification [Sun01] which provides a similar composition mechanism except that it does not support the specification of glue code using scripts.

3.1.3 Wrapping

Wrapping is the traditional approach used to extend an existing class or to transform the interface of a class into a new one. The basic forms of wrapping are subclassing and aggregation (class- and object-based composition). An evaluation of these two approaches can also be found in [HO99].

Subclassing

If the component to be adapted is instantiated by the developer, it is possible to use inheritance to extend the component. The developer subclasses the wrapper from the original class and uses the subclass instead of the original one. An advantage is that the wrapped component may be passed back to the component library. If the component is passed back to the original component library, however, it is important to adhere to the Liskov substitution principle [Lis87, LW94]. Otherwise, the system's reliability cannot be guaranteed.

It is important to note that some restrictions apply. The class that needs to be wrapped must not be `final` and only methods that have been declared `virtual` can be overridden. If a method is overridden, the system will always execute the overriding method. Thus, it is not possible to override a method selectively depending on whether

the subclass is handled by caller A or caller B. This problem, however, is addressed by the Composition Filter approach presented in [ABV92].

Aggregation

As an alternative to inheritance, it is possible to use a wrapper that has an aggregate relationship with the wrapped object. In this case, the wrapper provides the new interface and translates all the requests into the API provided by the wrapped object. This approach is also referred to as adapter design pattern and is presented in [GHJV95].

The drawback of aggregation are that the type of the wrapper and of the aggregate object are unrelated to each other and the wrapper can only be used for translation, not to override methods of the original class. Additionally, it is necessary to wrap and unwrap the wrapped objects whenever the aggregate object is required.

Composition Filters

The composition filter model [ABV92] is similar to Java's object model but distinguishes between internal and external objects and adds states and filters to an object. While internal objects are owned by the object, external ones are not and can be shared with other objects. Message invocations of objects are first evaluated by the filters controlled by the states and then dispatched to an appropriate method. The selected method can be one of the object's methods, or a method of one of its internal or external objects. Depending on the state of an object, different filters are active and hence different aspects are provided to the object's client.

Composite Adapters

Another approach based on wrapping is the composite adapter design pattern presented in [SML99]. The goal of the composite adapters is to allow the independent development of an application and the framework models the application uses, hence enabling an application to use the framework in different configurations.

The idea of the composite adapter is to implement all wrapper classes adapting the application classes to the framework as inner classes of a composite adapter class. While the composite adapter class takes care of the instantiation of the wrapper classes, the inner classes only implement the adaptation code. To simplify the implementation of the adapters, a new scoping (`adapter`) construct is proposed.

3.1.4 The Software Bus

The software bus is an abstraction of a shared medium used by any two components to communicate. The analogy to a hardware bus where components can be connected and disconnected easily is intentional. While a software bus simplifies the communication

between the participants, it increases the system's complexity, since theoretically every participant can communicate with every other participant.

The software bus per se does not solve any of the adaptation problems. If one component puts messages onto the bus that are not understood by the participants, interoperation between the components is not possible. The problem is simply moved from the interface or the data type level to the software bus's message type level.

Polylith

The Polylith software bus approach presented in [Pur94] is similar to the composition language approach but with a focus on distributed systems. Polylith uses a Module Interconnection Language (MIL) to describe the externally visible interfaces of a module. This language is similar to an IDL definition as used by RPC. On the basis of this description an MIL compiler can generate bindings for the relevant module. These bindings can be used to glue the modules together within the same address space or in different address spaces in the case of a distributed computing environment.

If the modules are executed in a distributed environment, the software bus can provide additional functionality. One example is the ability to share the information exchanged between two modules with other modules such as debuggers. Another functionality is to look up services to find a suitable target component.

An interesting aspect of Polylith is NIMBLE [PA90, PA91], a technique that enables the external adaptation of a module. NIMBLE allows developers to describe how the parameters in a procedure call need to be coerced to match the callee's signature without changing the source code of the modules involved. Based on the description of the actual procedure call and the procedure provided by another module, the developer specifies the translation which is instantiated by Polylith at run-time. Hence, NIMBLE makes it possible to decouple the development of the program application and the underlying libraries.

Bart

Other software bus implementations such as Bart use a publish/subscribe-style interaction between the components [Bea92]. Components may subscribe to messages from the software bus and publish messages of their own type onto the software bus. Whenever a component publishes a message, the software bus delivers the message to its subscribers.

One of Bart's layers allows components to share data in relational form. An interesting feature of Bart is a glue language (SGL) similar to Prolog that allows developers to define the relationship between data models in different components. The glue language allows the transformation, filtration, and combination of tuples performing a functionality similar to Polylith's Module Interconnection Language (MIL) and NIM-

BLE. Programs written in SGL are compiled and executed in the process where the data resides.

3.2 Separation of Concerns

Some programming problems cannot be solved easily using a procedural or object-oriented programming style in combination with design patterns [GHJV95]. Such problems are typically associated with aspects (or concerns) that cross-cut several different classes or software modules. Frequently, such concerns deal with performance, security, or failure-handling problems.

A simple solution to this problem is the implementation of a monolithic component that supports all of the relevant concerns using a configuration file. Currently, XML [BPSM98] is popular for this purpose. While this is sufficient for software components with a small number of configuration options, it does not solve the problem itself.

A better way to deal with these concerns is to separate them. Although the principle of dealing only with a single concern at a time was already identified in [Dij76], no foolproof solution has yet been found. Today, several approaches exist that solve this issue for various problem domains.

Efficient management of the different concerns is also interesting from a marketing point of view, since it supports the generation of different products from the same code base. Depending on the configuration that a customer wants, a different product can be generated. Such a set of products typically is referred to as a product family [Par76]. This is also interesting in terms of performance because the application does not have to deal with concerns in which the user is not interested.

3.2.1 Configuration Languages

Configuration languages (sometimes also referred to as extension languages) are used for the configuration of a software component. Configuration languages are similar to scripting languages and are almost always interpreted or compiled at load time. Because of this similarity, many scripting languages can be used as configuration languages as well (e.g., Tcl or Python). Emacs lisp, used by the emacs [Sta99] editor, was one of the first configuration languages and is still one of the most widely known. The potential of configuration languages has also been presented in [Ous98].

Configuration languages allow users to extend and adapt the functionality provided by a component. The advantage is that instead of having to learn the internals of the component, users only need to understand the extension language and the access points provided by the component. These access points are available in the form of functions and hooks and have to be provided by the component. While the functions are similar to functions in any other programming language, hooks are callbacks that allow users

to specify their own code to be executed at certain points. Since the user only has to be aware of these hooks and functions, configuration languages are a black-box adaptation technique.

Configuration languages expose their full power when all the components within the system can be configured using the same configuration language. This allows users to place global definitions, such as defintions concerning the look and feel of all visual components, into a central file that can be read by all components. Hence, the user does not have to change each individual component to keep a common look and feel across all of the components.

While configuration languages increase the flexibility of a component, this support should be considered during the component's design time. Retrofitting a software component to support a configuration language frequently requires a complete redesign of the component. Although configuration languages provide the maximum flexibility while still maintaining the component's black-box characteristics, they are difficult to implement, since the developer of the component needs to select carefully which functionality should be available to the configuration language.

3.2.2 Generic Programming

Generic Programming [JLM00] is a programming design method that tries to separate different aspects of the implementation using abstract interface specifications. These interface specifications define the hooks available for adding new aspects and parameterizing the various parts of the system. Depending on the programming language used, these interfaces can either be dynamic — using inheritance (dynamic polymorphism) — or static — using templates (static or parametric polymorphism).

While inheritance enables the parameterization to be changed at run-time, templates provide better performance. Since many configuration options of an application are known at compile time, parametric polymorphism should be used whenever possible. Good examples of libraries using generic programming without limiting the user's flexibility are the C++ Standard Template Library (STL) [MS96] and the Boost Graph Library [SLL02]. The flexibility of the STL has also been shown by the Persistent Standard Template Library [Gsc01b], a plug-compatible replacement for STL using persistent memory.

When using templates it is necessary to have access either to the component's source code or to an intermediary object format identifying the template parameters. The user of the component does not have to be aware of the component's implementation details, however, to be able to parameterize it. Hence, we consider generic programming a gray-box adaptation technique.

3.2.3 Generative Programming

Similar to generic programming, generative programming separates concerns and aspects into different building blocks (or layers) with a well defined interface. Generative programming, however, exploits the dependencies between the variabilities, separates the problem space and solution space, and uses configuration knowledge to map between these spaces. Configuration knowledge is provided by generators that are able to compose the building blocks according to the requirements specification of the programmer [CE00].

Generative programming has also been used by GenVoca [BSST93, BCRW00], a system that is able to compose building blocks describing basic data structures or synchronization directives into more complicated data structures. As shown in [BCRW00], it is even possible to provide consistency checking for a given composition and to provide a textual description of the composed system. Since each aspect is encapsulated within its own module, these modules appear as black-box entities to the developer. Depending on the implementation of the generator, it has to be aware of the building blocks' implementation. For instance, if the final system should be optimized using partial evaluation [DGT96], the source code of the components has to be available.

3.2.4 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [KLM+97] is an approach that tries to separate the development of a component and the associated concerns. This is achieved by the modularization of the cross-cutting concerns. The core advantages of AOP are:

- The developer can ship different versions of the application (forming a product family) without having to include all the concerns in all the different applications and thus without compromising the application's performance.
- The aspect or concern can be developed and maintained separately from the original program, thus reducing the cost of development and maintenance.

The original approach of aspect-oriented programming [KLM+97] is to write the application using a component programming language. This allows the developer to capture the application logic on a higher level without having to deal with low level issues such as performance optimizations. The low level optimizations are described with an aspect language, thus separating the aspect from the application code. A special compiler (weaver) is then used to weave the aspects described in the aspect language into the original application.

Typically, the component programming language is an extension of a real programming language. The extensions of the programming language allow the weaver to identify certain structures in the original program. On the basis of the aspect program

the weaver can apply some transformations to the program, such as merging loops, that a native compiler would have been unable to optimize.

The disadvantage of this approach is that a new compiler needs to be written for the component language. A different approach using plain Java as component language is taken by AspectJ.

AspectJ

In contrast to the approach presented in [KLM+97], AspectJ [GHH+] uses the Java Programing Language [AG97] as component language. Thus the programmer of the core application does not have to learn a new programming language, nor is it necessary to write a new compiler or extend an existing one for the component language.

The aspect programming language of AspectJ, however, is an extended version of the Java Programming Language. New constructs had to be added to allow the user to specify point-cuts. A point-cut specifies the location in the core application where an aspect has to be woven into. This allows an aspect to be added to a component available in source form even if the aspect has not been foreseen by the developer of the component.

```
aspect TraceMyClasses {
  pointcut myClasses() : within(MyClass1) || within(MyClass2);
  pointcut myConstructs() : myClasses() && executions(new(..));
  pointcut myMethods() : myClasses() && executions(* *(..));

  /* catch calls */
  static before(): myConstructs() {
    System.err.println(thisStaticJoinPoint.getSignature()+" {");
  }

  static after(): myConstructs() {
    System.err.println("} "+thisStaticJoinPoint.getSignature());
  }
}
```

Figure 3.1: Defining Point-Cuts in AspectJ

A sample aspect program illustrating how an aspect has to be woven into an application is shown in Figure 3.1. The aspect programming language uses a kind of regular expression for method calls for the specification of the point-cuts. The constructs shown in Figure 3.1 specify an aspect that requires some code to be added before and after each invocation of the constructors of classes `MyClass1` and `MyClass2`.

While AspectJ provides more flexibility to programmers, it is a white-box adaptation technique since it requires knowledge about the implementation of the program to

be adapted. In [Kic96], this approach is presented as an open implementation. This principle is in direct contrast to the open-closed [Mey88, LW94] principle.

AspectJ is not an aspect-oriented programming language in the sense defined previously, since it only allows the addition of new code to existing classes. It does not allow enhanced transformation on the abstract syntax tree, which would be necessary for the application of user-defined optimizations.

COMPOST

COMPOST [LH00, ALNH01] is a program transformation environment whose goal is the adaptation and composition of components into applications. COMPOST is similar to AspectJ but provides the ability to perform arbitrary code transformation, hence operating on a lower layer. The core layer of the COMPOST environment is the RECODER [Lud01], a source-to-source transformation library for Java programs. COMPOST is still a work in progress, however, and so far only simple code transformations, such as consistently renaming methods in a software engineering project, are supported. In future releases, the authors plan to support the weaving of new aspects into a program, or the refactoring of a program to enable performance optimizations (see also Section 3.3.2).

3.3 Performance Adaptation

Components should be general and reusable in a large set of different applications. Unfortunately, this comes at the cost of performance. Sometimes components are simply too bloated for a specific application, especially when only a small functionality provided by them is necessary. In other cases, components might suffer from performance problems because they have not been optimized to interact with each other, or simply because the compiler is missing some information of how to optimize their interaction.

3.3.1 Simplicissimus

The Simplicissimus project tries to optimize the performance of systems composed from multiple components or libraries. Simplicissimus is especially suited for libraries making use of templates and written for C++. Today, many such libraries exist, such as the Matrix Template Library (MTL), or the BOOST Graph Library (BGL). While these libraries provide good performance if used in combination with the compiler's built-in types, they perform poorly if used in combination with user-defined types. This is due to the fact that, unlike for built-in types, the compiler is unaware of many optimizations possible for user-defined types.

The approach taken by Simplicissimus [SGML01] is to allow library developers to specify optimizations possible for the types defined within their libraries. This specifi-

cation can be read in by the compiler and used for subsequent optimizations. Typical optimizations are the mathematical transformation of a term or the elimination of temporary variables by means of special purpose functions instead of operators. For instance, $a = p * q$ stores $p * q$ in a temporary variable and copies it to a using C++'s assignment operator. If a user-defined type provides a $\text{mul}(a, p, q)$ function, however, it would be more efficient to use this instead of the previous expression. Simplicissimus allows compilers to take advantage of such optimizations for user-defined types. So far, the system has been implemented for the GNU C++ and the Pro64 compilers.

3.3.2 COMPOST

As explained in Section 3.2.4, COMPOST is a term-rewriting tool that can be used to optimize existing compositions. By analyzing the composition of the components, it is possible to identify the parts of a component that are unneccessary for the final application. For instance, a doubly linked list that is only used as a stack does not require methods such as `insert`() or `delete`(). Removing these methods reduces the size of the application but does not significantly improve the system's performance. The goal of COMPOST is to go one step further and simplify the data structure used by the component. Thus, in the case of the doubly linked list, COMPOST should be able to convert the list into a singly linked list, hence improving the performance of the `push_front`() and `pop_front`() [Goo01] methods.

Chapter 4

Towards A Classification of Adaptation Techniques

In the previous chapters, we presented a set of adaptation techniques and their inner workings. In this chapter we compare the characteristics of these adaptation techniques to be able to put the different approaches in relation to each other and to select an appropriate adaptation algorithm for solving a given problem.

4.1 Objectives of Adaptation

Adaptation and composition techniques attempt to achieve one or more of three objectives. The first objective is the adaptation of different components for their composition. Hence, the focus here is the inter-operability of components providing different parts of an application or a system. Adaptation is necessary because the components are typically implemented independently by different vendors, so it is unlikely that components will be able to interact with each other directly (for instance, a word processor and a dictionary component). This objective is mainly addressed by composition languages, the software bus architecture, and type-based adaptation, which is our approach that will be explained in Chapter 5.

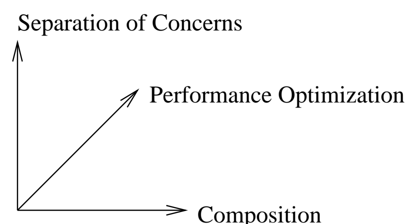


Figure 4.1: Objectives of Adaptation

The second objective is to allow the developer to exchange or add to the set of aspects (or concerns) that a component exhibits. Such an aspect could be support for persistence, security, or transaction management. While such support can be accomplished using configuration files, it is not always a good choice from a performance point of view. Other approaches are the modification of the source code itself. Although these techniques are more complicated to implement, their advantage is that they produce slim components providing only the necessary functionality. In [CE00], the first objective is referred to as horizontal composition and the second as vertical composition.

Finally, the third objective is addressed by adaptation techniques that focus on the optimization of a composed system. Depending on the optimization technique, it may be applied to horizontal and vertical adaptation and composition techniques. For instance, Simplicissimus [SGML01] enables the compiler to optimize horizontal compositions, while partial evaluation supports the optimization of vertical compositions by fusing different aspect definitions. Term-rewriting tools, on the other hand, enable vertical and horizontal systems to be optimized [Goo01].

4.2 Adaptation Transparency

Another important issue is whether an adaptation technique operates on the component's source code, hence requiring white-box components, or whether it uses only the interfaces exported by a component in which case it is able to adapt black-box components. While components can be easily categorized as white-box or black-box, this is not the case for adaptation techniques. Many adaptation techniques fall somewhere inbetween and are referred to in the literature as gray-box adaptation techniques. Since the terms black-box, gray-box, and white-box are not always used consistently in the literature, we will stick to the following definitions:

Black-Box Adaptation: The implementation of the components is entirely concealed from the adaptation technique and the component is accessed only via its interface. Hence, knowledge of the component's source code is not required.

Gray-Box Adaptation: The implementation of the component is concealed from the developer. Hence, the developer does not have to understand the source code. The toolkits and compilers performing the adaptation of the component, however, do have access to the source code and are allowed to modify it and use this knowledge for further optimizations.

White-Box Adaptation: The implementation of the components has to be known to both the developer and the tools working on the source code. Replacing a component with a newer version might break existing adaptations [Gsc01a].

Most black-box adaptation techniques use some form of wrapping for the adaptation of the component. This is due to the fact that only the interfaces provided by the component are available and that, if one ignores binary adaptation techniques, the component's implementation cannot be modified by the adaptation technique.

Gray-box and white-box adaptation are typically chosen by adaptation techniques that deal with the vertical composition or the separation of concerns. This is due to the fact that the different concerns are relatively small compared to the size of a component and encapsulating each concern into a different class would lead to a prohibitively expensive system.

Gray-box adaptation is used by generic programming as provided by the C++ programming language [Str97]. C++ templates provide the component's source code to the compiler as well as the type and the template parameters a component can be parameterized with. This allows the compiler to instantiate different versions of the component and optimize its performance. A somewhat similar approach is taken by generative programming, where the different concerns are implemented by different modules. These modules are composed by a generator which knows the different configurations possible. Although these adaptation techniques require availability of the source code, the developer does not have to deal with it. The source code is only necessary for the compiler or generator to be able to understand and compose the individual modules.

While the adaptation of black-box and gray-box adaptation techniques is transparent to the developer, the adaptation of white-box adaptation techniques is not. Hence, white-box adaptation has also been described as *open implementation* by [Kic96]. An adaptation technique that is based on white-box adaptation is AspectJ, an aspect-oriented programming language. White-box adaptation techniques unfortunately have some weaknesses that still need to be resolved.

For the integration of an aspect, the programmer needs to be familiar with the implementation of the component itself. AspectJ describes where a given aspect has to be woven into a core application by specifying the method calls where the aspect needs to be woven in. Thus, AspectJ provides a white-box adaptation technique where knowledge about the component's implementation is crucial.

When a component is being enhanced, however, it is likely that its internal structure will change and that the invocation of methods that an aspect relies on will be moved or changed. This is typically done either by renaming a method, by splitting the functionality provided by a single method into two different methods, or by adding more functionality to an existing method. Depending on how an aspect has been defined, it is possible that the point-cuts defined in AspectJ no longer apply and thus the aspect might only be woven partially into the enhanced component. Since AspectJ is a relatively new programming language, more research is necessary to identify how an aspect should be defined and how the compiler can identify whether an aspect can be applied cleanly to a newer version of the core program.

4.3 Application Domain

Not every adaptation technique can be applied to every component model since most techniques assume a specific granularity of components. Server-side component models such as the CORBA Component Model or the Enterprise JavaBean component model focus on large-grained application-oriented components. These component models only specify the interfaces that a component implements. This is due to the fact that the same instance of a server-side component can be used by many clients at the same time, which requires the component to provide a stable interface. Thus, these components can only be composed using black-box adaptation techniques, even though white- and gray-box techniques can be used for the development of a family of such components.

On the other hand, desktop component models exist in both black-box and white-box form, so that, depending on the component model, gray- and white-box adaptation techniques such as AspectJ, COMPOST, or C++ templates can be used. Additionally, unlike for server-side components, the implementation of desktop components may be changed. This is due to the fact that the instance of a modified component will only be used locally by other components within the same application.

4.4 Adaptation Time

Depending on the technique used, the adaptation of components can be performed during different phases of the product development process. We can differentiate between the following times of adaptation:

Design time: The adaptation of the component is manifested as part of the system's design. This is the case for wrapping or generative programming, for instance. Wrappers have to be taken care of during the design of the system and generative programming generates the required components from the requirements specification [BCRW00].

Compile time: The adaptation of the component takes place during the compilation of the component. This is typically the case for aspect-oriented programming where the aspects are woven in by a special compiler or by performance optimization techniques that analyze the components used by the application.

Run time: The adaptation occurs during the component's execution. This is the case, for instance, for configuration languages and type-based adaptation. The configuration code specified by a configuration language is typically loaded into a component during its run-time. In the case of type-based adaptation, the adaptation is performed when the client requests another component from a naming or trading service whose type was not known during design time or compile time.

4.5 Degree of Automation

Almost all of the adaptation techniques presented in Chapter 3 require the programmer to perform the adaptation manually by specifying how the component has to be transformed. Although tools such as SWIG automate the generation of language bindings for scripting languages, the script code still has to be implemented by hand.

With the current state of the art in computer science, automated composition is only possible if the semantics of two components does not have to be understood by the computer. Hence, the computer does not have to decide on the basis of the interface or implementation of the components alone whether and how two components can be composed. Additionally, automated adaptation can only be used in combination with black-box and gray-box adaptation techniques. It cannot be used in combination with white-box adaptation techniques since they require knowledge by the developer, which stands in contradiction to the requirement for an automated adaptation process.

Generic programming is an adaptation technique that takes the above limitations into account and supports the automated adaptation for vertical composition. Knowledge of the different configurations is encapsulated in the generator [BCRW00].

Type-based adaptation, on the other hand, is the first adaptation technique that supports automated adaptation for horizontal composition while taking today's limitations into account. The idea is to provide a repository of reusable adapters and an algorithm that chooses the best adapter. The adapter repository stores the adapters together with the required meta-information to provide for efficient selection of the adapters required. This allows the adaptation process to be reduced to the selection of appropriate adapters, which in turn is a simple graph problem of finding the best path.

4.6 Summary

Table 4.1 gives a brief classification of the various adaptation techniques and their characteristics. On the basis of this table we can determine that all the horizontal composition techniques are based on black-box adaptation. This is due to the fact that component implementation and component composition occur at different design stages and that the component integrator should not have to deal with the component's implementation. Performance optimization techniques, on the other hand, are based on gray-box adaptation since they typically require the availability of the system's source code.

The second column of the table shows that only black-box adaptation techniques can be used in combination with server-side components. This is due to the fact that a server-side component must not be modified after it has been instantiated, since it might be accessed by other clients. Only during the design and the implementation of such a component can gray- and white-box adaptation techniques be used.

Adaptation technique	Objective of Adapt.	App. Domain	Adaptation time	Adaptation Transparency	Automated
Scripting Language	composition	both	design time	black	no
Composition Language	composition	both	design time	black	no
Software Bus	composition	both	run time	black	no
Type Based Adaptation	composition	both	design time & run time	black	supported
Wrapping	composition & SOC[a]	both	design time	black	no
Configuration Languages	SOC	both	run time	black	no
Macros	SOC	desktop	compile time	white	no
Generic Programming	SOC	desktop	design time	gray	no
Generative Programming	SOC	desktop	design time	gray	supported
AOP	SOC	desktop	compile time	gray & white	no
COMPOST	perf.-opt.	desktop	compile time	gray	yes
Simplicissimus	perf.-opt.	desktop	compile time	gray	yes

[a]SOC=Separation of Concerns

Table 4.1: Classification of Adaptation Techniques

Additionally, we can see that almost all white- or gray-box adaptation techniques are applied when the application is compiled. Generative programming is the only exception to this. Generative programming generates a component based on its requirements specification. Since a generator could also be viewed as a special compiler operating on the component's specification, we could also have said that the adaptation occurs during the component's compile time. Design time, however, is more appropriate in this case, since the requirements specification is, unlike source code, part of the component's design time.

Finally, the classification shows that automated composition is not possible for white-box adaptation techniques. As we explained previously, white-box adaptation and automated composition are contradicting terms. Automated adaptation requires the adaptation to be performed automatically, hence without requiring any knowledge from the user. On the other hand an adaptation technique is defined as a white-box technique if it requires the developer to have knowledge about the component's implementation.

Chapter 5

Type-Based Adaptation

Most component models such as the CORBA Component Model (CCM) [OMG99a, OMG99b, OMG99c], the JavaBeans component model [Ham97], the Enterprise JavaBeans (EJB) component model [MH00, DYK01], COM [EE98], or .NET, rely on black-box components with well-defined and publicly available interfaces. Depending on the developer's knowledge of these interfaces, the components can be composed to interact with each other. In a *dynamic distributed system* [Grü00] such as the Internet, one component might request, for example, a weather service from a naming or trading service. If the two components have not been designed to interact with each other, there is currently no means for them to do so.

In this chapter, we show how this problem can be addressed using *type-based adaptation*. Type-based adaptation builds on the type that a component *provides* and the type that it *expects* from another component. The type of a component is provided by all the component models we have presented so far and we will show how to extract the type expected by a component in Sections 5.4 and 5.5. Since a type defines a contract that has to be fulfilled by the component and its clients [Mey92], it is sufficient to have the component's type information plus a repository of adapters specifying how the individual types need to be translated.

Type-based adaptation does not rely on any additional description of the semantics of the component as could be provided by semantic description frameworks such as the DARPA Agent Markup Language (DAML) [DAM02]. Semantic description frameworks are not yet readily available and rely heavily on standardization, as we will explain in Section 5.2.

5.1 Fundamentals

Programming languages use types for different purposes: to help programmers identify programming errors in an early stage of the software development cycle, to classify an instance of an object, to describe its characteristics and its possible uses, and oth-

Reflexiveness:

$$\frac{\Gamma \vdash A}{\vdash A <: A}$$

Transitivity:

$$\frac{\Gamma \vdash A <: B \quad \Gamma \vdash B <: C}{\Gamma \vdash A <: C}$$

Subsumption:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A <: B}{\Gamma \vdash a : B}$$

Figure 5.1: Basic Subtyping Rules

ers. Many programming languages distinguish between built-in types, classes, and interfaces [Weg90]. Similar to programming languages, almost all component models available today use types to distinguish the functionality provided by different components. Hence, we present the necessary type theory before presenting the adaptation model underlying type-based adaptation

On the set of available types, we can define the subtype relation ($<:$) such that $A <: B$ denotes A as a subtype of B. The subtype relation indicates that any variable of type A can be viewed as type B if $A <: B$ (*subsumption*). The subtype relation is reflexive, antisymmetric, and transitive [AC96]. These basic rules of subtyping are shown in Figure 5.1 for the environment Γ that consists of typing assumptions for variables, each of the form $v : T$.

A function type F with input parameters of the types P_i with $1 \leq i \leq k$ and output parameters of the types Q_j with $1 \leq j \leq l$ can be written as follows:

$$F := P_1 \times P_2 \times \cdots \times P_k \rightarrow Q_1 \times Q_2 \times \cdots \times Q_l$$

Now we can define the subtyping rule for function types. By subsumption we can pass input parameters of type P'_i with $1 \leq i \leq k$ and $P'_i <: P_i$ and output parameters of type Q'_i with $1 \leq j \leq l$ and $Q_i <: Q'_i$ to a function of type F. A function type F is a subtype of F'' ($F <: F''$) if F'' can be substituted with F. This is the case if a function of type F accepts for each input parameter a superset and for each output parameter a subset of the set of types accepted by F''.

$$F'' := P''_1 \times P''_2 \times \cdots \times P''_k \rightarrow Q''_1 \times Q''_2 \times \cdots \times Q''_l$$

Hence, F has to accept input parameters of type P'_i (with $P_i <: P'_i$) and output parameters of type Q'_i (with $Q'_i <: Q_i$). We can observe that the input parameters are

$$\frac{\Gamma \vdash A <: A' \quad \Gamma \vdash B <: B'}{\Gamma \vdash A \times B <: A' \times B'}$$

$$\frac{\Gamma \vdash P'' <: P \quad \Gamma \vdash Q <: Q''}{\Gamma \vdash P \rightarrow Q <: P'' \rightarrow Q''}$$

Figure 5.2: Subtyping Rule for Functions

contravariant ($P_i'' <: P_i$) and that the output parameters are covariant ($Q_i <: Q_i''$). Parameters that are both input and output parameters (i.e., they occur on both sides of the function) are invariant. The subtyping rule for function types simplified on the basis of the subtyping rule for tuple types ($\times$) is shown in Figure 5.2.

On the basis of the above definitions, we can define an object as a set of attributes identified by a label l_i and having type T_i with $1 \leq i \leq n$. An object type can be written as follows:

$$O := \{l_i : T_i\} \quad i \in \{1, \ldots, n\}$$

The subtype relationship between two objects specifies that an object O having a superset of the attributes of an object O' is a subtype of O' ($O <: O'$). According to the subtype relationship defined for function types, the attributes l_i in O may be replaced with subtypes if constants are represented as $() \rightarrow T$ and variables as $T \rightarrow T$. Again, input parameter types are contravariant, constants and output parameter types are covariant, and variables are invariant. For instance, O is a subtype of the following objects O' and O'':

$$O' := \{l_i : T_i\} \text{ with } i \in \{1, \ldots, n - m\}$$

$$O'' := \{l_i : T_i''\} \text{ with } T_i <: T_i'' \text{ and } i \in \{1, 3, 5, \ldots, n\}$$

Analogously to object types, we can define an interface type I with the difference that the individual attributes T_i are restricted to function types. Additionally, these function types must have a parameter of type I that is both an input and output parameter. The subtyping rule of Figure 5.3 also applies to interface types and combination of interface and object types.

Finally, by extending the environment Γ with the definition for the subtype relation ($<:$) such that $\Gamma, u <: v \vdash u <: v$ we can define the subtype relationship for recursive types. This allows us to use type-based adaptation on recursive types, as used, for instance, by a linked list.

$$\frac{\Gamma, u <: v \vdash T <: S \quad u \in FV(S, \Gamma) \quad v \in FV(T, \Gamma)}{\Gamma \vdash \mu u.T <: \mu v.S}$$

$$\frac{\Gamma \vdash T_i <: T_i' \quad \forall i \in J \quad J \subseteq I}{\Gamma \vdash \{l_i : T_i\}^{i \in I} <: \{l_i : T_i'\}^{i \in J}}$$

Figure 5.3: Subtyping Rule for Object and Interface Types

So far, we have only considered structural equivalence between types. In typical object-oriented programming languages as used by today's component models, a type is also identified by a name such that for two types to be equal not only their structure but also their names have to be equal. Now we can define the type of a component as a collection of interface or object types and an implementation thereof. A component C provides a set of interfaces I_i such that $C <: I_i$. In the case of a desktop component model a component C may also be of an object type O.

Therefore, a component has a given type and is bound to the contract [Mey92] defined by the types it implements. Using the rule of subsumption we can identify that a subtype of a component has to fulfill the same requirements as its supertype. This rule leads to the Liskov substitution principle [LW93, LW94] that states that a program accepting a type T has to exhibit the same behavior when operating on a type T' being a subtype of T.

5.2 Adaptation Model

In all the component models we have discussed, the functionality provided by a component is specified using a type or a set of types that is implement by one or more programming artifacts. For type-based adaptation, only the type information about a component is of importance. The mapping information from one type to another has to be carried out by a human capable of understanding the different types that need to be mapped. A human has to decide whether two types may and can be translated into each other, and, if so, to specify their translation. Only the presence of this information makes automatic adaptation possible.

To provide the mapping information, type-based adaptation uses *adapters*. These adapters algorithmically describe how to translate different types into each other. The adapters can be written in a typical programming language such as C++ or Java. Depending on the application domain, the types that an adapter translates vary. In the case of server-side components, the interfaces provided and expected by the components are the only type information required. In the case of an IDE the adapters may also operate on a per-method level, specifying how a given method has to be mapped into another method. These adapters also have to take the mapping of the method's parameter types into account. For a detailed discussion of type-based adaptation for

server-side and desktop component models, please refer to Sections 5.4 and 5.5. An adapter a that maps a type T_{from} into a type T_{to} can be written as follows:

$$a : T_{from} \succ T_{to}$$

An adapter can be implemented either by simulation or by type conversion. If the adapter a simulates T_{to} based on T_{from}, then the source component (or client) interacts with the adapter a providing T_{to} and the adapter in turn interacts with the target component providing T_{from}. Hence, the adapter is active as long as the client interacts with the target component.

Alternatively, the adapter may convert the data provided by the target component providing T_{from} into a translated component providing T_{to}. In this case the adapter performs the conversion when the source component initiates the interaction with the target component, and afterwards the source component (or client) interacts directly with the conversion.

An important aspect of type-based adaptation is that the adapters are stored in a repository with some meta-information about the adapters, such as the types they translate. On the basis of the adapter's meta-information it is possible to retrieve an adapter that maps one type into another type. Additionally, adapters can be concatenated (chained) on the basis of their meta-description stored in the repository. Using subsumption we can define the concatenation operator ($\circ$) between different adapters as follows:

$$\frac{\Gamma \vdash a : T_{from} \succ T_{to} \quad \Gamma \vdash b : T'_{from} \succ T'_{to} \quad \Gamma \vdash T_{to} <: T'_{from}}{\Gamma \vdash a \circ b : T_{from} \succ T'_{to}}$$

Our approach can be seen as an extension of the adapter pattern [GHJV95, MSL00]. The difference, however, is that the adapters are first-class objects described on their own and that there is an adapter repository which has full knowledge about the adapters available and the transformations they describe. The repository stores such information as the interfaces that the adapters translate, and optionally their performance characteristics or whether the quality of the the information provided by a component deteriorates in translation from, for example, a `gif`-image to a `jpeg`-image. On the basis of this information the adaptation process can be automated.

In fact, code that needs to be written for these adapters already exists in many of today's software systems in the form of wrappers or in the form of subclasses, the basic form of wrapping. Unfortunately, in such code, the adapter is part of a bigger component and thus cannot exist on its own, especially when subclassing is being used. To be used in combination with the adapter repository, such code has to be factored out into a separate class, the adapter. Afterwards, it can be used in combination with the adapter repository and thus can be reused in other systems where similar interface transformations are necessary.

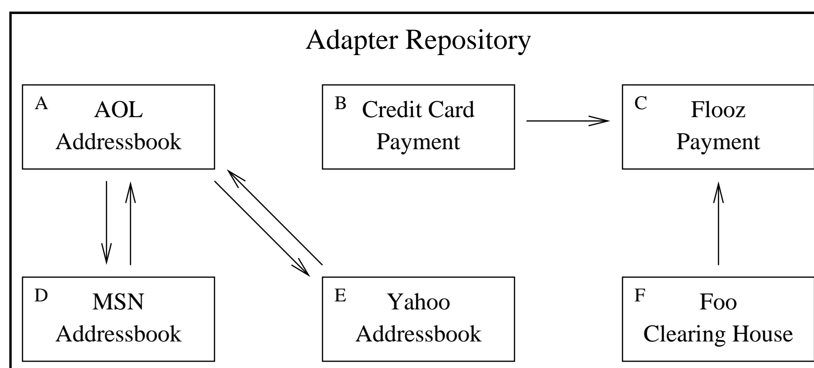


Figure 5.4: A Sample Adapter Repository

As shown by the sample adapter repository in Figure 5.4, an adapter repository forms a directed graph G where the interfaces are represented by the vertices ($V(G) = \{A, B, C, D, E, F\}$) and the adapters by the edges ($E(G) = \{AD, DA, AE, EA, BC, FC\}$) between the interfaces they can translate. If a required adapter is missing, adapters can be combined. For instance, the missing adapter DE can be provided using $DA \circ AE$. To find a suitable combination of adapters a simple shortest path algorithm is sufficient. Additionally, the algorithm can be tuned to prefer adapters having specific characteristics by applying different weights to each edge (adapter). For instance, if the user will not accept any deterioration of quality during the adaptation, the weight of adapters that lead to deterioration can be set to ∞.

A different approach to the adaptation problem is the use of a semantic description framework that supports the description of the interfaces, its methods and data structures using a common ontology. Such an approach might be implemented using DAML [DAM02]. We claim, however, that this approach only adds another level of indirection since no commonly accepted ontology for the semantic description exists. Different ontologies will be unavoidable since companies are unwilling to release any information about their products in their early development states. Such information would be crucial for the standardization of a common ontology.

5.3 Requirements

The requirements of type-based adaptation depend heavily on the application domain it is applied to. For distributed systems such as a distributed object system or the world-wide Web it has to be designed differently than for a development environment. This is due to the difference in the granularity (method vs. interface level) of the systems' object models.

In any case an implementation will consist of the following parts:

1. An *Adapter Description* defining the adapter's properties and characteristics.
2. A *Package Format* that comprises an adapter and its description.
3. An *Adapter Repository* storing the adapters.
4. An *Adaptation Component* finding an optimal translation chain of adapters.

5.3.1 Adapter Description

The description of an adapter has to specify the interface that the adapter translates from and the interface that it maps to. The description should be able to supply additional information about the adapter such as the performance complexity or other properties.

This description can be maintained by the adapter class itself and accessed using introspection as employed by the JavaBeans component model [Ham97]. Alternatively, the description can be stored externally using a configuration file. The adapter repository should support both choices since this keeps the adapters simple and enables the repository to be ported to other application domains easily where the use of introspection might not be possible. For the configuration file, XML [BPSM98, Har01] can be used. XML has the advantage that existing tools can be used to parse the adapter specification. Only a document type definition [BPSM98] or XML Schema that describes the syntax of the adapter's specification has to be provided.

A sample adapter description is shown in Figure 5.5. The `SampleAdapter` converts from a fictitious `com.yahoo.AddressDatabase` interface to a fictitious `com.amazon.AddressBook` interface. The specification also indicates that the adapter's implementation is provided by the `at.ac.tuwien.infosys.tba.SampleAdapter` class and that the transformation will not result in the loss of any information provided by the original component. Thus, when the adapter is applied, some information about an address might be lost.

5.3.2 Packaging

To simplify the adapter's installation and transfer to another site, all the class and resource files required for the adapter should be put into a single archive. In general, we recommend using a format similar to the format already exploited by one of the existing component models. For instance, an implementation that is based on Java should use a `jar`-archive for its implementation. In the case of the CORBA Component Model a `zip`-file containing the adapters and their specification files should be preferred.

```
1  <?xml version="1.0"?>
2
3  <!DOCTYPE adapter-description PUBLIC
4   "http://www.infosys.tuwien.ac.at/Staff/tom/babelfish.dtd">
5
6  <adapter name="SampleAdapter">
7    <mapsfrom>
8      <interface>com.yahoo.AddressDatabase</interface>
9    </mapsfrom>
10   <mapsto>
11     <interface>com.amazon.AddressBook</interface>
12   </mapsto>
13
14   <implementation type="classname">
15     at.ac.tuwien.infosys.tba.SampleAdapter
16   </implementation>
17
18   <lossless>false</lossless>
19 </adapter>
```

Figure 5.5: Adapter Working on the Interface Level

5.3.3 The Adapter Repository

The adapter repository is responsible for storing all the available adapters and their meta-information. The more adapters are stored within the repository, the more powerful type-based adaptation is. The adapter repository itself is straightforward since any data structure that can store a graph with parallel edges (multiple edges connecting the same two nodes) is sufficient. Since the graph will consist of a large number of blocks (independent components within the graph) of semantically equivalent interfaces, an adjacency list that stores all the out-edges for a given vertex v should be used.

5.3.4 The Adaptation Component

Whenever it is necessary to perform an adaptation the adaptation component can be queried for an adapter or a combination thereof to perform the required translation. Hence, the adaptation component has to provide lookup methods as shown in Table 5.1 that can be used by the client component or component infrastructure whenever a component needs to be adapted. The first method returns an already instantiated chain of adapters and the second method returns an `Adapter` object describing a combination of adapters. The `Adapter` object provides methods that can wrap a service with the according adapter.

`Object getAdapter(Object from, Class type_to)`
: Instantiates an adapter that provides the interface `type_to` to the client and interacts with the service represented by `from`. The object returned is the adapted object.

`Adapter getAdapter(Class from, Class type_to)`
: This method looks up an adapter or combination of adapters that provide the interface `type_to` to its clients and interacts with a service providing interface `from`. The object returned is a factory that can be used to create instances of the adapter.

Table 5.1: Methods Provided by the Adaptation Component

If the lookup of the required adapter is performed transparently by the component infrastructure, it gives the user the impression of having automatic composition at hand. Only if no adapter or chain thereof exists must the developer provide a new adapter for the composition to succeed. After the adapter has been provided, it can be added to the adapter repository and made available to other users of the adaptation component.

For the lookup of the adapters in the repository, Dijkstra's shortest path algorithm is the best one currently available. Dijkstra's algorithm has a complexity of $O(|E(G)|)$ which typically is much smaller than $O(|V(G)|^2)$ [BH89] with $E(G)$ denoting the set of edges and $V(G)$ the set of vertices of the graph G.

5.4 Requirements for Server-Side Component Models

In this section we present the requirements of type-based adaptation for server-side components that have not already been presented in Section 5.3. For server-side component models such as the CORBA component model (CCM) or the Enterprise JavaBeans component model (EJB), the type of a component is represented by the interfaces it implements. This is due to their RPC interaction style using stubs for the communication with server-side components. Thus, the implementation of an adapter A that translates from interface I_{from} to an interface I_{to} is straightforward.

In a distributed system as used for server-side components, the components are distributed over multiple computers, each implementing a different service as shown in Figure 5.6(a). Typically, when one component needs a service from another it queries a name server or trader for the service using a well known identifier. The naming service returns a reference which the client casts to a specific interface as shown in

Figure 5.7. Now the client can interact with the service returned by the naming service. If the service requested by a client does not match the interface expected by the client, however, an exception will be thrown and the client will be unable to communicate with the service.

One approach to solve this problem is to implement a proxy service as shown in Figure 5.6(b). In state-of-the-art distributed systems, however, such a proxy server cannot be operated transparently since the proxy ps has to register using a different identifier. If ps were to register using the identifier of s, all other clients that expected the native interface of s would fail or have to be changed. Additionally, each proxy server incurs an additional network delay. Hence, this solution is expensive, especially if multiple translation servers are necessary.

In a distributed object system, the client's source code looks as depicted in Figure 5.7. On this basis we can ascertain that the `narrow` operation (a system-independent cast operation) is a perfect choice for the integration of type-based adaptation. At this point of execution the interface provided by the server is available as part of the object reference and the interface expected by the client is passed as an argument to the `narrow` operation. Additionally, it allows us to plug type-based adaptation into an existing system transparently by upgrading the middleware layer only.

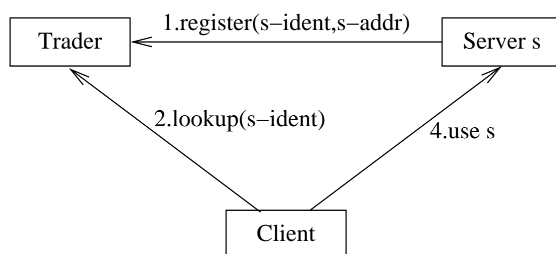
5.4.1 Trading Services

Trading services allow clients to look up a service on the basis of certain properties or just the interface it provides. For instance, a tourist office might request any component that implements the `at.ac.tuwien.Weather` interface to display weather information on its Web site. If the trading service supports type-based adaptation, it is possible to return a different component as long as it can be translated into `at.ac.tuwien.Weather`. This is especially of interest if no component implementing the interface required by the client is registered at the trading service and the client is interested in any component providing the service rather than in a specifc component. Hence, clients can benefit from a trading service that is aware of type-based adaptation.

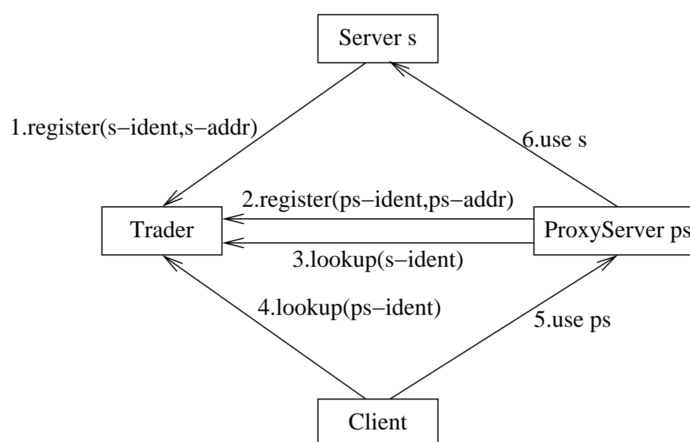
The trading service needs to be able to infer whether a service implementing a different interface can be transformed to the interface requested by the client. To allow this functionality, a trading service should provide methods similar to those presented in Table 5.2 in addition to those it already provides. If the trading service implements a query language that includes support for the selection of a specific interface, that query language should be extended accordingly.

5.4.2 Adapter Location and Usage

Both clients and trading services should be aware of type-based adaptation. A straightforward approach would be to locate an adaptation component and repository at each



(a) Standard Access to a Service



(b) Access of a Server via a Proxy Server

Figure 5.6: Accessing a Service in a Distributed Component System

```
1   try {
2     Context ctx=getInitialContext();
3     Object dobj=ctx.lookup("CookieServer");
4     DemoHome ch=(CookieHome)
5                 PortableRemoteObject.narrow(dobj,CookieHome.class);
6     Cookie c=ch.create();
7     System.out.println(c.getCookie());
8   } catch(Exception e) {
9     e.printStackTrace();
10  }
```

Figure 5.7: Typical Client Code in a Distributed Component System

---

`Reference lookup(Class type_to)`
: Looks up a service that provides the interface `type_to` to its clients or can be translated to that interface.

`Reference lookup(Ident ident, Class type_to)`
: Looks up a service identified by `ident` and provides the corresponding adapter converting the interface to the interface denoted by `type_to`, if possible.

Table 5.2: Methods provided by the Trading Service

site. While placing an adaptation component at each site is possible, using different isolated repositories poses a problem since much of the power of type-based adaptation would be lost. Thus it is necessary to provide a centralized adapter repository or to link the adapter repositories to each other.

For systems that are based on Java and thus simulate a homogeneous environment, mobile code can be used and the necessary adapters can be downloaded from a special repository either by the client or by the server. A similar approach for the use of mobile code is taken by Jini [Sun99b]. Jini does not, however, focus on the adaptation of interfaces but on shielding the wire protocol from the service's clients. The client requests, for instance, a proxy for a printer implementing the printer service and uses Java method calls to interact with the proxy which in turn talks to the printer using a proprietary protocol.

Another difference to the non-distributed application scenario is that each component might be available for a different architecture, thus resembling a heterogenous system and complicating the adaptation problem. This problem can be resolved using one of the following approaches (the exact choice depends on the final implementation):

1. Distribute fat binaries that include the code for each architecture.

2. Distribute the adapters in source form and compile them on the target platform.

3. Use an interpreter or virtual machine for the execution of the adapters.

5.4.3 Security Considerations

Since type-based adaptation unfolds its maximum flexibility if the adapters can be exchanged between the participants, security becomes a major concern. This can be solved by executing the adapters within a safe sandbox environment. The Java Virtual Machine [LY99], for instance, can provide a perfectly safe sandbox environment that

does not allow the downloaded code to execute arbitrary instructions. Fortunately, much work exists in the context of Java and Mobile Agents [Gon98, Gon99, BDS00, HKK00] that can be reused for an implementation of type-based adaptation.

Additionally, it has to be determined where the adapters should be executed and hence whose computing resources should be used. We recommend the adapters to be executed by the party that wants the components to interact with each other. Typically, this is the client requesting a service. This places the control of the adaptation process as well as the security risks with the party benefiting from the composition. Thus, service providers do not suffer any disadvantage.

5.5 Requirements for Desktop Component Models

Although the principle of type-based adaptation remains the same for Desktop Component Models and IDEs, there are some fundamental differences. Mobile code, for instance, is of lesser importance for a development environment than for a distributed system because all the components are available locally. Also, security is of little concern for desktop component models since the adapter repository is developed locally or adapters are bought like other software components, with the same guidelines for components as for adapters.

5.5.1 Granularity of Adaptation

A key difference to server-side component models is the different type system provided by desktop component models. Since all well-known server-side component models are based on RPC, RMI [Sun99a], or a variant thereof, the interfaces they implement are the only type information available. This is because of their interaction style that uses stubs for the communication with server-side components.

Classes

Desktop Component Models, however, have a more complicated type system since their types can be interfaces, classes, or even built-in types. Hence, an adapter A should not only be able to translate from an interface I_{from} to an interface I_{to} but instead from any type T_{from} to any type T_{to} where T may also be type tied to an implementation such as a `class`. Depending on whether the adapter adapts a component to provide the type of another component or whether the adapter converts the information provided by a specific component, different challenges arise.

If an adapter adapts a component to provide the type of another component, the interface I_{to} cannot be replaced by a type T_{to} tied to an implementation. This is due to the fact that an adapter can only provide such a type by subclassing the original class providing T_{to}, hence requiring the availability of T_{to}. If T_{to} were available, however,

T_{to} could be used directly. Additionally, T_{to} must not be a built-in type since a user-defined class cannot provide the same functionality as a built-in type. This is due to the fact that in most programming languages it is not possible to subclass a built-in type. These limitations only arise for T_{to} and not for T_{from} since T_{from} is the component that is available and whose interface does not have to be emulated. If the purpose of the adapter is to convert the information provided by a component, both I_{from} and I_{to} may be of any type.

Methods

While server-side component models are composed on the basis of the interfaces they exhibit, this is rarely the case for desktop component models. In a typical IDE, events are mapped onto method calls. In the case of the JavaBean component model, for instance, the listener methods of similar events such as mouse-related events are grouped into an event set and the signatures of the methods responding to these events (*listener methods*) are specified by a listener interface (Figure 5.8).

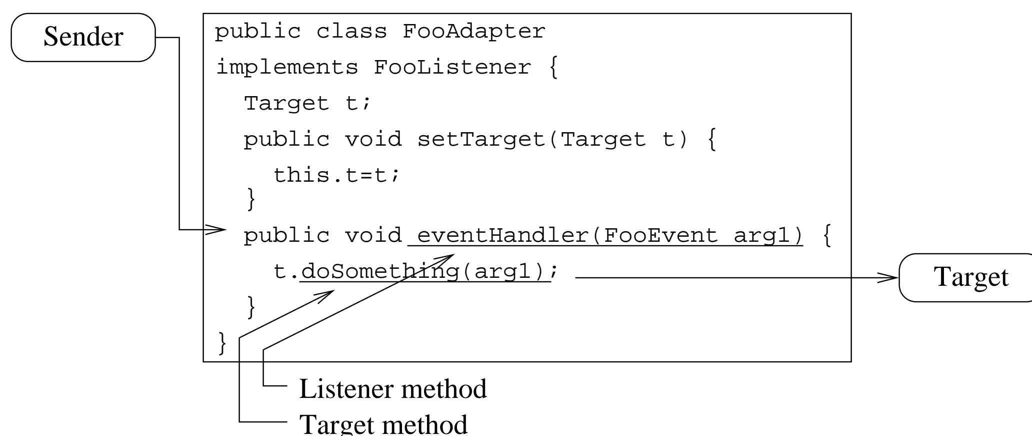


Figure 5.8: Adaptation on Method Basis

If a component b has to execute an action whenever an event occurs in a component a, then b has to implement the whole listener interface, or the IDE has to generate an adapter class implementing the listener interface and calling the target method. Since we cannot generally assume that a component b has been written with another component a in mind, typically the latter will be the case. When this happens, however, the IDE is limited to the generation of an adapter calling a method m that is type-compatible with the listener method l defined in the listener interface such that $m <: l$. Hence, the target method has to have the same number of parameters and each parameter of the target method has to be a supertype of the appropriate listener method's parameter. Other alternatives are the invocation of methods having no parameter at all, or letting the programmer supply the relevant adapter.

If we want to simplify the above using type-based adaptation, the adapters have to operate on a per-method granularity as well, hence influencing the description and implementation of the adapters.

5.5.2 The Adaptation Component

The adaptation component needs to provide the same functionality as the adaptation component in a distributed component system but has to be able to operate on method signatures as well. Hence, in addition to the methods shown in Table 5.1, it has to provide another set of these methods operating on a per-method level. Since the components are available to the IDE and are composed by the IDE, the adaptation component should obviously be an integral part of the IDE.

Another difference because of the per-method granularity is that there will be many different matches of possible adapters. The adaptation component should therefore be able to describe the individual chains of adapters and allow the programmer to choose an appropriate adapter from the different possibilities. Depending on whether all the different adapter chains or just the n shortest adapter chains should be computed, Dijkstra's algorithm needs to be extended to keep track of alternative paths.

5.5.3 Adapter Description

Since IDEs offer a much finer granularity of composition and allow the user to specify the interaction between components on an event/method level, the adapter description presented in Section 5.3.1 needs to be extended. The extended adapter description shown in Figure 5.9 also supports the specification of the method signatures expected and provided by an adapter.

Additionally, the implementation of the adapter should be specified differently. While it is possible to use a class for the adapter's implementation, it barely makes sense because the IDE has to generate an adapter implementing the listener interface of the component generating the event in any case. Since the listener interface is only seldom known during the adapter's implementation, it cannot be specified as part of the adapter. Thus, the adapter's source code should be part of the adapter's XML description.

5.5.4 Performance Considerations

Though performance is of lesser importance with today's computers, it should not be ignored entirely. The communication between desktop components is much cheaper than the communication between server-side components because desktop components are typically executed within the same process. This leads to a tighter coupling between desktop components, hence making it more important to optimize their composition, especially if multiple adapters need to be chained.

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

<!DOCTYPE adapter-description PUBLIC
 "http://www.infosys.tuwien.ac.at/Staff/tom/babelfish.dtd">

<adapter name="PaintAdapter">
  <mapsfrom>
    <returns>void</returns>
    <arguments>
      <arg>java.awt.events.MouseEvent</arg>
    </arguments>
    <exceptions/>
  </mapsfrom>
  <mapsto>
    <returns>void</returns>
    <arguments>
      <arg>java.awt.Point</arg>
    </arguments>
    <exceptions/>
  </mapsto>

  <implementation type="code">
    // ... source code of the adapter ...
  </code>

  <lossless>false</lossless>
  <behavior>false</behavior>
  <sender-state>none</sender-state>
  <editor>at.ac.tuwien.infosys.tba.PaintAdapterEditor</editor>
  <description>
    This adapter provides a method taking a mouse event, extracts the
    mouse cursor's current position as a point, and executes a method
    taking a point as argument.
  </description>
</adapter>
```

Figure 5.9: Adapter Working on the Signature Level

Fortunately, several techniques exist that can be reused for type-based adaptation. If several adapters need to be chained, the method invocations of the adapters may be inlined. This instructs the compiler to merge a large number of method calls into a single one and thus significantly lowers the performance penalty. Alternatively, if many arguments are constant, which is typically the case for computer generated code, partial evaluation [DGT96] can be used to simplify the adapters. Partial evaluation can help to eliminate a considerable number of redundant computations. The exact optimization techniques to be used, however, will depend on the final implementation.

Chapter 6

Evaluation

We performed a set of experiments to evaluate the applicability and feasibility of type-based adaptation for different application domains. In the first experiment, we added support for type-based adaptation to a server-side component model and implemented a set of server-side components. In the second one, we added type-based adaptation to the Component Workbench (CWB) [Obe01, Obe02], an extensible integrated development environment developed at the Technische Universität Wien. In the third system, type-based adaptation was implemented for agent systems. Before presenting the individual systems, however, we will present the relevant application domain without support for type-based adaptation. This allows us to explain more clearly the benefits of using type-based adaptation and how to add this support to the different application domains.

6.1 Server-Side Component Models and Web Services

Typically, server-side components are used as the back-end of Web services. The use case [FS98] in Figure 6.1 shows an *Internet Superstore* where the *User* browses and selects the products available. After the *User* has selected the products he wishes to buy, he selects the shipping address from an *Address Book Service* with which he manages his addresses. When the *User* wishes to pay for the products, the payment is performed via a Payment Component that charges the *User*'s account.

Almost all of today's Internet stores have an internal address book service. This is inconvenient, however, because the *User* has to manage a different address book for each store he wishes to buy goods from. He would prefer to use an external address book service to manage his addresses and simply instruct the *Internet Superstore* (or any other Web site using an *Address Book Service*) to use the address book provided by that service.

As long as the *Address Book Service* implements exactly the interface expected by the *Internet Superstore*, this is possible today. If not, the components are unable to interact and the benefits of the above approach are lost. With multiple service providers

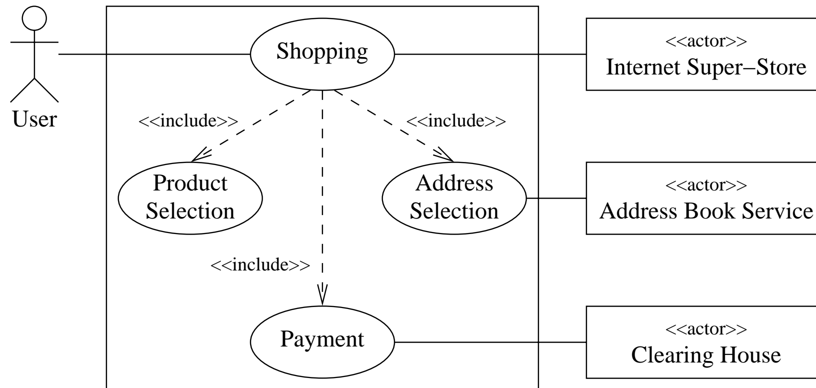


Figure 6.1: Internet Shopping Application

it is unlikely that they all will implement the same interface. This is where type-based adaptation comes in. Type-based adaptation allows for the transparent and automatic adaptation of the interface of one service (e.g., the Address Book service) to match the interface required by another (e.g., the *Internet Superstore*). To verify these claims we implemented our own address book component and tried to integrate it into two existing applications.

6.1.1 Address Book Component

We decided to use EJBs for the implementation of the address book component [DYK01]. For the applications to be used in combination with our own address book component we downloaded and installed the petstore and e-bank Web applications from Sun Microsystem's Web site.

Our first step was to add support for type-based adaptation to the EJB component model. As explained in Section 5.4 this requires the modification of the `narrow`() function. The EJB component model uses the `narrow`() function of Java's `javax.rmi.PortableRemoteObject`, which supports the delegation of this function to a delegate class. The delegate class merely has to implement the `javax.rmi.CORBA.PortableRemoteObjectDelegate` interface and has to be specified by the `javax.rmi.CORBA.PortableRemoteObjectClass` system property. Hence, we decided to use this mechanism for the integration of the adaptation component.

Secondly, we implemented our own address book component and extended both the petstore and e-bank applications to allow customers to specify an external address book component. Instead of having to type in the shipping and mailing addresses over and over again, the customer specifies once the URL of an external address book component. Afterwards, the Web application contacts the address book component specified by the customer instead of the internal one.

Additionally, we implemented a set of adapters to convert between the different address book components. With type-based adaptation, the interface provided by the external address book component is adapted to the interface expected internally by the petstore application and equally to the e-bank application. Then, the application can present a list of the mailing and shipping addresses to the customer.

In both of the applications above, the adaptation was performed as expected. In this implementation, however, the adaptation component poses a minor security risk to the operator of the Web application. Since the Web application looks up the address book component indicated by the customer and we have implemented the adaptation component as part of the `narrow()` function, the adapters are executed by the Web application.

For security reasons, however, the adapters should be executed by the Web browser. Otherwise, the customer might be able to inject malicious code if he has access to the adapter repository. Additionally, executing the adapter within the Web browser allows the customer to control what addresses are transferred to the Web application and thus safeguards the customer's privacy. Since this issue does not directly relate to the adaptation problem, it will be addressed in future versions of our implementation.

6.1.2 Weather-Service Component

The availability of an automated adaptation facility is also important for services providing information such as a weather service. For instance, a travel agent may want to display weather information for the travel destinations it offers. The travel agency does not care what weather service component it uses as long as it provides information for one of its travel destinations. Obviously, the travel agency is neither willing nor able to change its applications whenever a new destination is added that is not covered by its current weather services or worse after one of these weather services fails.

We therefore implemented a set of different weather services providing access to weather information along with a set of adapters able to convert between them. Finally, to see whether type-based adaptation is able to provide a means of transparent adaptation of the different weather services, we implemented a Web application that retrieves weather information from a weather component and displays the information on a Web site.

Our system was able to adapt the different weather components transparently. In some cases, however, the adapters were unable to provide some information expected by the client. This is due to the fact that an adapter cannot generate information that is not provided by the component it adapts. A solution to this problem would be to allow adapters to contact several different services to collect the required information such that an adapter can be represented as follows:

$$a : T_1 \times T_2 \succ T_{to}$$

6.2 Desktop Component Models

Type-based adaptation can be used not only in combination with server-side component models but also in combination with desktop component models as employed by *Integrated Development Environments* (*IDE*). In an IDE, the *Developer* typically selects the components he wishes to use, configures the components via some property sheets and defines their interaction by selecting an event that a component can generate and specifying what has to be done when the event occurs. Usually, the *Developer* has to write code to set up the connection (i.e., to allow the component to interact with another component). Otherwise, the interaction between the components is limited [Ham97].

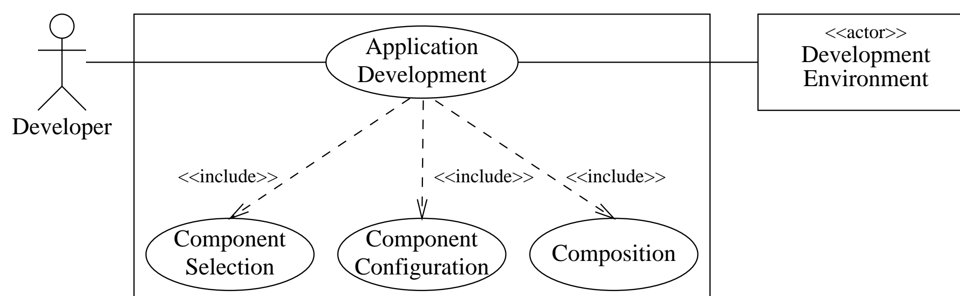


Figure 6.2: Integrated Development Environment

Type-based adaptation alleviates this problem by allowing the *Developer* to reuse connectors that have been written previously. It allows the *IDE* to understand the purpose of the connectors and enables the *IDE* to present the *Developer* with a set of connectors that could be reused for each newly created connection. For instance, a connector toggling a certain property can be reused in different applications to toggle a configuration option in a dialog or in a drawing application to indicate whether the pen is drawing or not.

For the implementation of this experiment, we added type-based adaptation to the Component Workbench (CWB) [Obe01], an extensible integrated development environment. This integration shows the benefits from using type-based adaptation in IDEs as well as the flexibility of the CWB.

6.2.1 The Component Work Bench

During the implementation of the AgentBean Development Kit (see Section 6.3.2), we realized that a modular component construction toolkit that can be tuned to various different application domains was still missing. This insight was the motivation for the implementation of the Component Workbench (CWB) [Obe01, Obe02], a flexible

IDE that supports the composition of various component models such as JavaBeans or Enterprise JavaBeans. Before we present the integration of type-based adaptation into the CWB, it is necessary to present the architecture of the CWB itself.

Architecture

Figure 6.3 shows the architecture of the CWB [Obe01]. The CWB uses a set of user interface modules that interact with a scenario. The scenario represents the current application to be assembled and in turn interacts with component wrappers. Each component in the CWB is wrapped using a component wrapper allowing the CWB to support components of multiple different component models. For each component instantiated by the developer using the CWB a component wrapper is instantiated. All the component wrappers implement a generic component model that is internally used by the CWB. On the basis of this architecture, the CWB even supports the integration of server-side components.

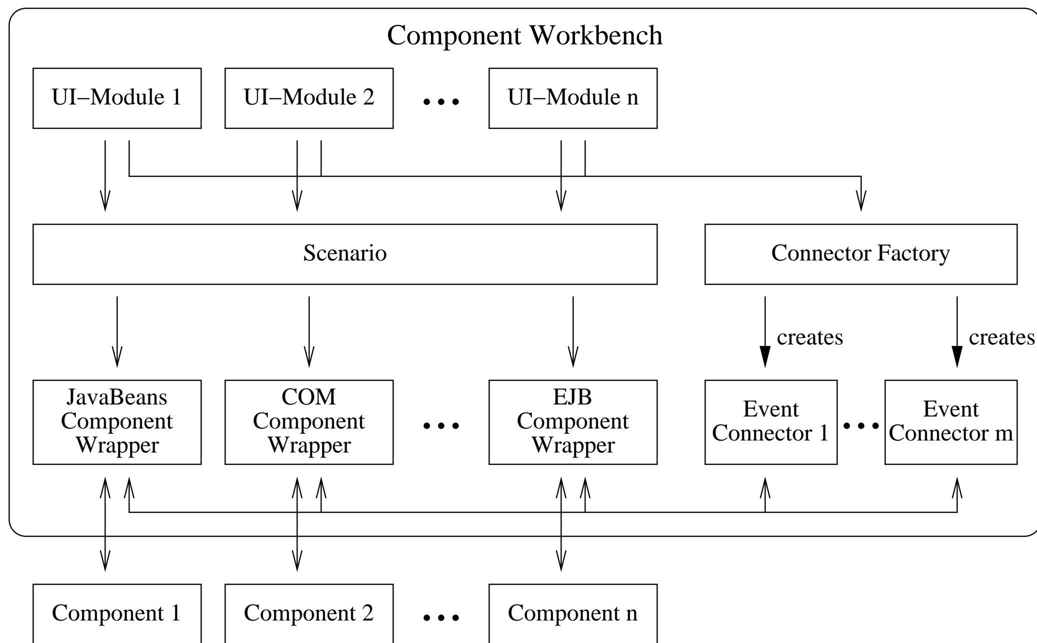


Figure 6.3: Connectors of the Component Workbench

The interaction between different components is managed by connectors that are instantiated by a connector factory when the developer selects two components to interact with each other. The connector factory supports different types of connectors in a similar way as the AgentBean Development Kit [Gsc00a]. On the basis of this design, we were able to integrate type-based adaptation into the CWB without changing the CWB itself.

Type-Based Adaptation Support

In the CWB, components are composed by means of connectors. Since the CWB supports different kinds of connectors, we implemented a new kind of connector using type-based adaptation. To connect two components using type-based adaptation, the developer only has to select the new kind of connector. Unlike for server-side component models the adapters may also operate on the following levels:

Interface Level: On this level, type-based adaptation transforms different components on the interface level. The implementation does not differ from the implementation of type-based adaptation for server-side component models.

Method Level: On the method level, adapters operate on method signatures instead of interfaces. This is important if an event is connected to a specific method of the target component and not to the target component as a whole.

Argument Level: In this case, adapters are applied to the individual arguments of a method call. Providing adaptation on this level can be convenient if no adapters are found on the method level because it allows the developer to generate an adapter based on lower-level adapters.

Depending on the level on which the adaptation should be performed, the developer has to select different parts of the components to be connected. On the interface level, it is necessary to select a source component, a target component, and optionally a source event set. On the argument level it is necessary to specify a source event and a target method as well. Additionally, the developer may specify constraints, that have to be fulfilled by the interaction patterns returned by the adapter repository, independently of the level on which the adaptation is performed.

On the basis of the developer's input, the adaptation component is able to compute a set of possible interaction patterns between two components. Since type-based adaptation operates on a lower level, it is likely that more than a single adapter will match. To allow the developer to distinguish the different matches, we generate a short description based on the adapter descriptions for each interaction pattern. After the developer has selected the desired interaction pattern, the connection wizard displays a subgraph of the adapter repository and the interaction pattern chosen. A graph showing the possible interaction patterns between a text field and a SOAP based prime tester component as well as the one selected by the developer (displayed with stronger lines) is shown in Figure 6.4.

Since a developer uses the IDE, it might be possible to use more powerful adapters supporting the specification of behavior in addition to the type-mappings. For an evaluation of the usefulness of such an extension, however, it is necessary to gain more experience with this new adaptation technique. Hence, in our current implementation, this feature can be deactivated.

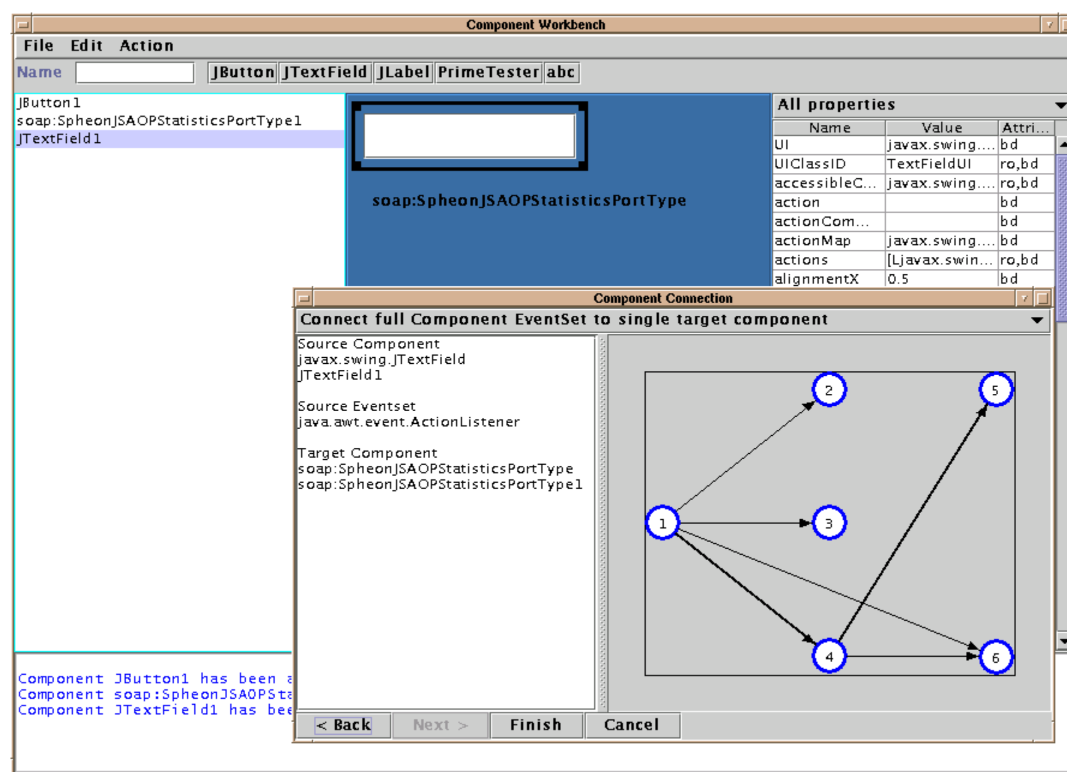


Figure 6.4: Type-Based Adaptation Connection Wizard for the Component Workbench

Sample Applications

For the first application, we used a SOAP prime tester component that we found on the Internet and implemented our own component that checks whether a given number is a prime. Additionally, we implemented some adapters that translate between the two components and some user interface elements such as text-fields and buttons. This allowed us to compose a small application using the new type-based adaptation connectors (Figure 6.4) without having to type a single line of code. The final application allows users to enter a number and test whether it is a prime number or not.

Type-based adaptation is also supported on a per-method basis. In this case, the developer has to select a specific source event instead of a source event set. This combination allowed us to implement a simple drawing application, again without having to write a line of code. The implementation of the application is based on a set of standard components and standard adapters. We only had to select the components and the adapters. All the adaptation necessary between the components was performed by the adapters found in the adapter repository.

6.3 Mobile Agent Systems

Before describing the experiments we performed in this application domain, we will give a short introduction to mobile agents and the terminology used. After this introduction, we will explain how agents can be constructed and how the construction process can benefit from type-based adaptation. Agents can benefit from type-based adaptation not only during their construction but also during their execution. Since agents are repeatedly executed on different agent execution platforms it is unlikely that each platform will provide exactly the same services to agents, hence requiring each agent to adapt to local services.

6.3.1 Mobile Agent Definitions

Throughout this case study we will adopt the definitions of the OMG's Mobile Agent System Interoperability Framework (MASIF) [MBB+98]:

- An *agent* is defined as a computer program that acts autonomously on behalf of a person or an organization.
- An *agent system* is a platform that can create, interpret, execute, transfer and terminate a mobile agent. An agent system is associated with an authority for whom the agent system acts.
- A *place* is a context within the agent system that provides a uniform environment in which an agent can execute. It is associated with a particular agent system. A place provides the means for managing mobile agents, enforcing security policies and accessing local resources.
- *Mobile agents* are agents that move from place to place to perform a given task. Usually, the itinerary of the mobile agent is determined by the mobile agent itself. Since this case study focuses on mobile agents, we will refer to *mobile agents* simply as *agents*.

In the following we will only consider *weak mobility* [FPV98], which defines the ability to transfer code and initialization data, but not the execution state. The agent, however, may explicitly decide to store its execution state within its attributes.

6.3.2 The AgentBean Development Kit

For the construction of mobile agents, we have defined a component-based agent model that allows us to simplify their design and implementation. For many network and systems management tasks, we can identify categories of components and patterns for interconnecting them in order to perform the task. For the AgentBean Development Kit (ADK) [GFP99], we used the following three categories of interrelated components:

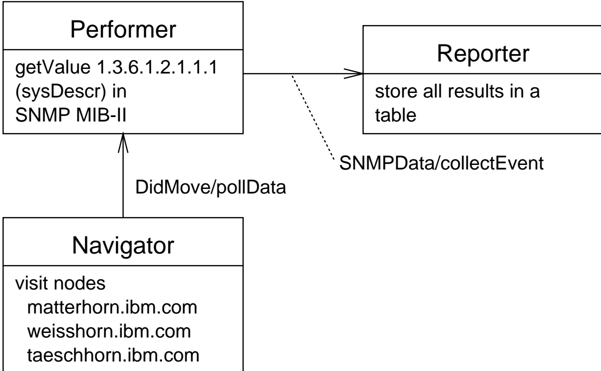


Figure 6.5: Example Illustrating the Three Component Categories

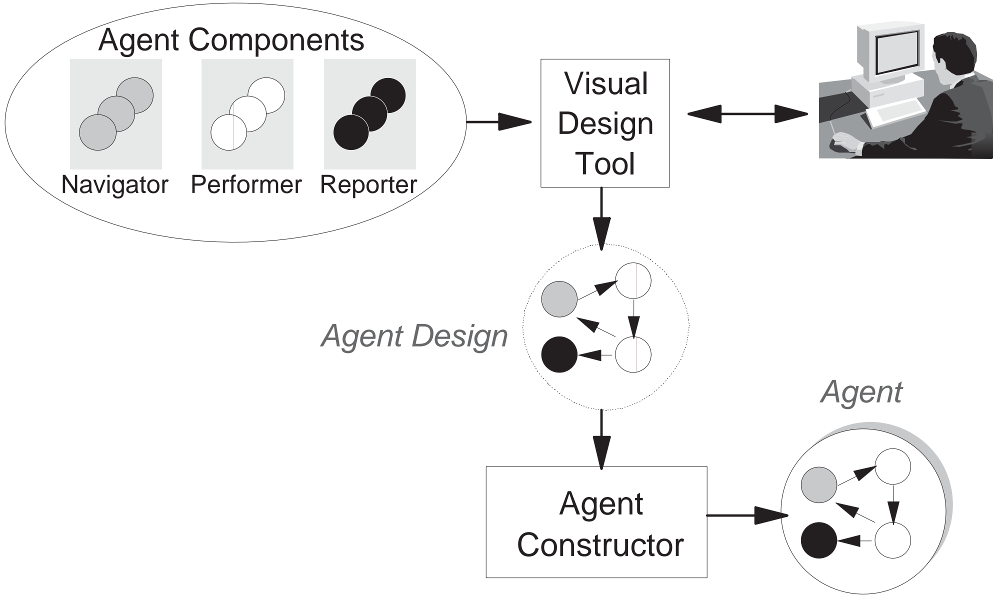


Figure 6.6: Based the ADK the Developer Combines Components to Build the Agent

Navigator: The components in this category are responsible for determining and managing the itinerary of the agent. The itinerary may be static, i.e., consisting of a list of nodes to visit which was determined at the time the agent was designed or instantiated, or dynamically based on the agent's previous computations and the current environment.

Performer: The components of the performer category carry out the management tasks that should be executed at the host of the currently visited place. An agent may contain one or more components linked together to perform a task.

Reporter: The components of this category manage the delivery of the agent's results to the designated destinations. A reporter component may, for instance, send a message to some display tool or aggregate the results and deliver them upon its return to the agent authority.

Although we have evaluated this agent model in the domain of network and systems management [GFP99], it can also be applied to the design of agents for handling tasks in other disciplines and is not restricted to network and systems management. The concept of components allows a modular definition, facilitates creation of agents and encourages further reuse. Instead of creating an agent from scratch, the designer of an agent can glue the required components together. This is compliant with the fact that the creator frequently wants to focus on the business logic of the agent and does not want to bother with the technical details of agent creation.

The interaction between components is based on event/action-based communication. An event generated by one component may trigger an action by another component. Figure 6.5 shows an example of the model, namely a simple mobile agent for collecting configuration information. The navigator component provides a list of nodes to be visited sequentially. Upon arrival at a place the navigator generates a `DidMove` event which triggers the performer's polling action. The performer reads the value of the system description from a local Management Information Base (MIB) [Sta93] and generates an `SNMPData` event that is recorded by the reporter. After the events have been processed by all the components, the navigator continues with its itinerary.

This component model supports a construction process as outlined in Figure 6.6. To support this process, we built the AgentBean Development Kit (ADK), a visual design tool that allows the selection of components from a list. The agent creator uses these components to compose the agent and specifies their interaction. Finally, the tool generates the source code for the agent. The advantage of this approach is that it enables IDEs to be used for the construction of agents. Our initial implementation of the ADK is based on Sun's BeanBox [Sun98a] model. With the ADK, an agent can be created in a straightforward manner on a business logic level. A system administrator who wants to create an agent does not have to be concerned with writing the code himself.

Use of Type-Based Adaptation

One limitation we encountered with this model [GFP99], however, was that the events that a component can generate have to match the event-receptors of the target component; otherwise, visual composition is not possible. Because JavaBeans map events onto interfaces, this problem can easily be solved using type-based adaptation. Since the CWB uses an event/action-based interaction style between the component and already supports type-based adaptation, we propose extending the CWB to support the creation of mobile agents as well.

6.3.3 The Calendar Agent

In another application, we have implemented two different adaptation approaches for the calendar agent [Jak02]. The calendar agent is a mobile agent available for the AgentBean Development Kit [GFP99] that can be used to schedule appointments. The calendar agent moves from host to host and queries whether a certain user has an appointment at a given time. Since different users use different calendar managers to handle their appointments, the calendar agent must be able to adapt to the different calendar managers. For the calendar manager, we have implemented two adaptation approaches: the bridge pattern [GHJV95] and type-based adaptation.

For the bridge pattern approach, we have defined an abstract calendar interface that is used by the calendar agent. To be able to use a specific calendar manager in combination with the calendar agent, we had to implement an adapter that interacted with the calendar manager and provided the abstract calendar interface. Currently, we have implemented adapters for the `ical`, `cm`, and the `KOrganizer` calendar managers. Since these calendars do not provide a Java interface natively, we also implemented a Java component wrapping each of these calendars and providing a native Java interface.

Using the bridge pattern design required us to implement only a single adapter for each calendar manager, hence limiting the number of adapters to the number of calendar managers. While implementing this approach, we identified several limitations [Jak02]:

- The interface of a calendar manager may undergo changes from one version of the program to another. In some cases, it may be easier to write an adapter that transforms one interface into another rather than into the abstract calendar interface.
- Sometimes an adapter transforming one interface into another will already be available. With the bridge design pattern, however, a new adapter would have to be implemented.
- The adapters translating the interfaces of the calendar manager program are rarely reusable by different programs. This is due to the fact that the programs are unlikely to use the abstract interface defined for the calendar agent.

After we implemented the calendar agent using the bridge pattern, we reimplemented it using type-based adaptation. Although both versions of the agent worked perfectly fine and were able to adapt to the individual execution environments, we discovered that type-based adaptation can be used in combination with the bridge pattern approach. Since type-based adaptation is not dependent on the interfaces involved, we can put the adapters that implement the bridge pattern into the adapter repository, hence providing the same functionality. If additional adapters are available which translate one calendar manager interface into another, they can be added to the adapter repository as well. Type-based adaptation is then able to make use of these adapters by chaining them with one of the adapters providing the abstract calendar interface.

6.4 Summary

Type-based adaptation allows the intercommunication of components providing semantically the same functionality but using different types. As we have shown in the previous sections, the adaptation can be performed automatically and support for server-side component models can be added transparently. Such an adaptation technique is important for Web services using server-side components for their back-end processing. On the one hand, Web service providers can offer a wider range of back-ends to be used in combination with their Web service and on the other hand, their back-ends can also be used in combination with other Web services. The advantage for users is that they can pick the back-end components that best fit their needs instead of being locked in to the components used by a single Web service. An example would be to use mapquest.com with the address book from yahoo.com.

In another experiment we showed that type-based adaptation can be easily integrated into IDEs. Although type-based adaptation cannot support a fully automated composition process for IDEs, it guides the developer when composing the individual components. Type-based adaptation shows the developer a set of possible adaptations for the components he wishes to compose. If one of these adaptations is suitable, the developer only has to select it. Otherwise, a new adapter has to be implemented.

In our last experiment we used type-based adaptation in the context of agent systems. In this experiment we also compared it to the bridge pattern. This comparison showed that type-based adaptation is more powerful, since its adapters do not have to adhere to a single interface definition.

Other experiments that might be interesting are the evaluation of type-based adaptation in the context of software maintenance or bridging between different component technologies. In the context of software maintenance, type-based adaptation might also be ideal for maintaining compatibility to older versions of a system when the interface of a component changes. In the Java Programming Language, this problem has been solved by allowing developers to declare a method as deprecated [AG97, LY99]. When

the developer uses a deprecated method, the compiler emits a warning message. While this approach allows the deprecated method to access the component's internal state, the legacy code necessary to maintain compatibility to older systems is still part of the component.

Type-based adaptation allows the decoupling of the component's legacy code from its implementation by providing a separate adapter implementing the legacy code. Now, the old and new versions of the interface can be used simultaneously without the user's noticing it. After all the users have upgraded to the new interface, the adapter can be easily thrown away without the need to change the component again. This allows the developer to focus on the new version of the system without having to worry about legacy code that only provides compatibility to older versions of the system.

Another application domain for type-based adaptation is the integration of a component into different component environments. To identify interfaces across different components, however, a uniform type identifier consisting of the component model as well as the interface has to be introduced. In this case, the adapters not only provide the code for translating between two different interfaces but also the bridge code necessary to translate one protocol into another. Before we are able to attack this problem, however, it is necessary to gain more experience with type-based adaptation.

Chapter 7

Related Work

Most of the work relating to type-based adaptation has already been presented in Chapter 3. We therefore limit the following discussion to the differences between that work and type-based adaptation.

The NIMBLE [PA90] language which is part of the Polylith [PA91] system also tries to attack the adaptation problem. The NIMBLE language allows developers to describe how the parameters in a procedure call need to be coerced to match the callee's signature without changing the source code of the modules involved.

NIMBLE, however, does not support the chaining of the interface translation. Hence, adapters defined by NIMBLE cannot be reused and the developer of the system has to specify adapters for all the different and valid compositions. Additionally, NIMBLE operates only on a per-procedure basis and does not take into account the relationships between procedures as defined by an interface.

An approach similar to NIMBLE is conciliation [SGS98], which also supports the adaptation of independently developed components. The advantage of conciliation is that it takes the components' object-oriented view into account. Conciliation operates on three layers: the component layer, the method layer, and the data layer. The data layer specifies how the individual methods are supplied to the client, the method and component layers define how the data layer is mapped onto the component specification, and the component layer specifies the possible groupings of class instances that must be available at run-time.

Hence, conciliation's data and method layer represent the information provided by the adapters used by type-based adaptation. They also take into account the interface layer which would be located between the method and component layer. Depending on the application domain, however, the information provided by the adapters can be limited to the method layer, which can be useful for IDEs, where lower-level composition mechanisms are necessary. The information provided by conciliation's component layer are provided by the adapter's meta-description.

Type-based adaptation supports the chaining of adapters, which means that more complex adapters can be generated from smaller ones. The advantage of this approach

is that not all possible combinations of adapters have to be provided, although all possible adaptations of the existing adapters and combinations thereof can be performed. Additionally, adapters can be reused for different components providing the same interface. Conciliation, on the other hand, does not support the chaining of conciliators. This is due to the fact that conciliators are generated from the description of all three layers (data, method, and component), hence operating on the components themselves rather than on the interfaces they provide.

Conciliation registers the conciliators as components in the Microsoft Windows registry and sets the component's *treatAs* key. This instructs Windows to instantiate the conciliator instead of the component. In the Windows registry, however, conciliators are statically hardcoded into the system and hence always the same conciliators are used. In contrast, the adapter chains generated by type-based adaptation allow different adapters to be used in different situations. Since the adapters are chosen at composition time, adapters can be implemented as pieces of mobile code to be downloaded on demand. This is especially of interest if the system has to be deployed on a large scale. Thus, type-based adaptation gives the user more flexibility.

Another interesting adaptation approach is that of Jini which uses proxies responsible for interacting with a given device. Whenever a client wants to interact with a device, it downloads the device's Java proxy and interacts with it. The proxy provides a Java interface and uses the appropriate wire protocol to interact with the device, hence shielding the application from having to deal with the device's wire protocol. This allows Jini to access devices from different environments as long as an appropriate proxy exists. An architectural overview of Jini can be found in [Sun99b].

Similar to type-based adaptation, Jini [FPV98] uses mobile code for its proxies that shield the application from the wire protocol. Jini's lookup service serves as a repository of services. Entries in the lookup service are objects to be executed in a Java Virtual Machine environment. These objects can be downloaded as part of the lookup operation and act as local proxies to the services that placed the code into the lookup service. Apart from shielding devices from its wire-protocol, however, Jini does not address the adaptation of software components.

The composite adapter design pattern presented in [SML99] also uses adapters. While type based adaptation focuses on the adaptation of the intercommunication between server-side components, this pattern focuses more on the software engineering side by trying to make possible the independent development of an application and the framework model the application uses. The idea of the composite adapter is to implement all wrapper classes adapting the application classes to the framework as an inner class of a composite adapter class. While the composite adapter class takes care of the instantiation of the wrapper classes, the inner classes only implement the adaptation code. To simplify the implementation of the adapters, a new scoping construct (`adapter`) is proposed. While the composite adapters operate on a lower level than

type-based adaptation and do not support automated adaptation, it might be possible to employ composite adapters for the adapters used by type-based adaptation.

Composition filters [ABV92] support the specification of filters that can be compared to the adapters used by type-based adaptation. The composition filter model is similar to Java's object model but distinguishes between internal and external objects and adds states and filters to an object. While internal objects are owned by the object, external ones are not and can be shared with other objects. Message invocations of objects are first evaluated by the filters controlled by the states and then dispatched to an appropriate method. The selected method can be one of the object's methods, or a method of one of its internal or external objects. Depending on the state of an object, different filters are active and hence different aspects are provided to the object's client. Composition filters thus operate on a much lower level than type-based adaptation.

DAML [DAM02] is a meta description language that can be used for semantic descriptions. While such a language is not enough to describe how different interfaces need to be translated, it can be used for providing the meta description of the adapters themselves. Such meta description can provide information about the interfaces the adapters are translating, whether their transformation leads to any deterioration of quality such as a conversion from a `gif`-image to a `jpeg`-image, or whether they also provide additional behavior.

Chapter 8

Conclusions

We have described type-based adaptation, a novel adaptation technique that supports the automated adaptation and composition of software components. This adaptation technique is important in the area of distributed systems such as the Internet. It allows users to specify only the components that need to interact with each other and adapts and composes them transparently. So far, type-based adaptation is the only adaptation technique that achieves this criterion.

Type-based adaptation is based on a repository that stores prebuilt adapters plus a meta-description of the transformations that the adapters perform, thus providing all the information necessary for the adaptation process. We have shown in Chapters 5 and 6, that this information is sufficient for the automated adaptation of software components. Although to some extent the adapter repository can be compared to semantic description frameworks [DAM02], such frameworks only provide a richer description of the components to identify whether two components provide the same functionality semantically and fall short in providing the necessary translation.

An advantage of type-based adaptation is that compared to a more traditional wrapping approach, it does not require adapters for all the different combinations of interfaces; to do so requires $O(n^2)$ (with n being the number of components) adapters, which would be prohibitively expensive. Type-based adaptation is able to chain the adapters stored in the adapter repository, hence requiring only $O(n)$ adapters. If additional adapters are available, however, type-based adaptation can take advantage of these adapters and can adapt the components more efficiently.

We have shown state-of-the-art component models and adaptation techniques. We have shown that component models can be classified as desktop (or local component models) and as server-side component models and that adaptation techniques address three objectives: adaptation for composition, the separation of concerns, and performance adaptations. While many adaptation techniques have already been developed, none of these techniques supports automated adaptation for the composition of software components. Additionally, we have presented our classification criteria for the

evaluation of adaptation and composition techniques. These criteria allow researchers to put different adaptation techniques into relation to each other.

8.1 Contributions

We have demonstrated that type-based adaptation increases the flexibility of dynamic distributed systems. Type-based adaptation allows users to select the combination of components to be used without having to deal with compatibility issues between them because it enables the component interfaces to be adapted transparently. The presented technique has a wide area of applicability, including Web services using server-side component technologies as back-ends.

Type-based adaptation simplifies the construction of applications using Integrated Development Environments (IDEs). Since it stores the adapters that have been written by the developer, the adapters can be reused more easily. With the meta-description stored about the adapters, IDEs that support type-based adaptation can guide the developer in the composition of the components, hence requiring less glue code to be implemented.

In a related project which was part of the WebMon project [GEGW01, GEG+02], type-based adaptation also served as the basis for the dynamic interface adaptation technique [GG01] developed at Hewlett-Packard Laboratories. WebMon is a tool that gathers performance data about clients, Web servers, and back-end application servers and supports the correlation of this data. To make it possible to correlate the data gathered we had to adapt the interfaces provided by the server-side components. Since we did not have access to the client's source code, we used a modified version of type-based adaptation to pass additional correlation information while maintaining compatibility between the clients and the adapted components.

We have also shown several criteria that can be used for the classification of adaptation techniques. These criteria were identified on the basis of a variety of case studies we have performed and systems we have implemented [GH99, GFP99, Gsc00b, Gsc01b, GEGW01, Gsc01a]. On the basis of our classification criteria, a developer can decide which adaptation technique should be used to solve a problem at hand.

8.2 Future Research Directions

Type-based adaptation has been evaluated using a set of controlled experiments. Since these experiments look promising, we plan to evaluate how type-based adaptation performs in a globally used system, such as an Internet Superstore, with many different users and several components to choose from. This would allow us to correct any remaining weaknesses in our reference implementation and demonstrate the strengths of our adaptation technique more clearly.

An interesting experiment is the evaluation of type-based adaptation in the context of software maintenance or bridging between different component technologies. In the context of software maintenance, type-based adaptation can be used for maintaining compatibility with older versions of a system when the interface of a component changes.

Type-based adaptation could also be used for the integration of components implemented for different component models (bridging). To identify interfaces across component model boundaries a uniform type identifier consisting of the component model as well as the interface has to be introduced. Then the adapter would not only provide the code for translating between two different interfaces but also the bridge-code necessary to translate one protocol into another.

Currently, type-based adaptation is used for the automated adaptation of software components. We are in the process of evaluating its potential to adapt components that do not provide all of the functionality requested by a client. It is indeed a powerful functionality beyond what we have shown in this dissertation if we can use adapters to provide the missing functionality by combining several different software components. Our initial experiments in this directions show promising results.

Appendix A

Glossary

Currently, there is no consistent terminology that describes software components and their different kinds of reuse. The following glossary presents the definition used within this thesis of terms that are used ambiguously within the literature.

Adaptation: This is the process of modifying a component to allow it to interact with another component. This can either be performed by modifying the component itself or by enclosing the component within a wrapper that provides the required interface. Typically, a component is adapted to be composed with another component.

Binary Adaptation: Binary adaptation refers to the process of modifying the binary form of a component. Typically, binary adaptation is used if the source code of a component is not available.

Black Box Adaptation: This refers to an adaptation technique that does not require knowledge of the component's implementation. Wrapping is a traditional black box adaptation technique.

Component: A *software component* is a piece of software that exhibits well-defined interfaces, does not require knowledge of its implementation, is easier to reuse than to reimplement and can be used for the implementation of other software systems.

Component Model: A component model defines the basic architecture of a component, specifying the structure of its interfaces and the mechanism by which it interacts with its container and with other components. The component model provides guidelines for creating and implementing components that can work together to form a larger application.

Composition: Composition refers to the process of generating larger software systems from components. Since software components are developed independently, their

interfaces are rarely plug-compatible. Hence, components frequently need to be adapted to be able to interact with each other.

Contravariance: Given an expression *expr* with a type parameter T, T is said to be contravariant with regards to $expr[T]$ if $expr[T] <: expr[T'] \Leftrightarrow T' <: T$.

Covariance: Given an expression *expr* with a type parameter T, T is said to be contravariant with regards to $expr[T]$ if $expr[T] <: expr[T'] \Leftrightarrow T <: T'$.

Gray Box Adaptation: This term refers to an adaptation technique that requires the availability of the component's source code for the adaptation to be performed. However, the developer himself is not required to know anything about the component's implementation. Examples of gray-box adaptation techniques are C++ templates or performance optimizers such as Simplicissimus.

Invariance: Given an expression *expr* with a type parameter T, T is said to be invariant with regards to $expr[T]$ if $expr[T] <: expr[T'] \Leftrightarrow T = T'$.

White Box Adaptation: In contrast to black-box and gray-box adaptation techniques, white box adaptation techniques require the component's source code to be available and the developer to understand the component's implementation. This is the case for most aspect-oriented programming languages available today because specifying how an aspect has to be integrated into a component requires detailed knowledge of the component's implementation.

Appendix B

Examples

B.1 JavaBeans

B.1.1 Bean Class

```
1   package at.ac.tuwien.infosys.examples.jb;
2
3   class DrawingArea implements Serializable {
4     private Color _color;
5     private boolean _drawing;
6
7     /* color property */
8     public void setColor(Color c) { _color=c; }
9     public Color getColor() { return _color; }
10
11    /* pen is drawing */
12    public void setDrawing(boolean b) { _drawing=b; }
13    public boolean isDrawing() { return _drawing; }
14
15    /* mouse listener */
16    public void addMouseListener(MouseListener l) { /* ... */ }
17    public void removeMouseListener(MouseListener l) { /* ... */ }
18  }
```

B.1.2 BeanInfo Class

```
1   package at.ac.tuwien.infosys.examples.jb;
2   import java.beans.*;
3
4   public class DrawingBeanInfo extends SimpleBeanInfo {
5     public PropertyDescriptor[] getPropertyDescriptors() {
6       try {
7         PropertyDescriptor rv[]={
8           new PropertyDescriptor("color",DrawingArea.class),
```

```
9          new PropertyDescriptor("drawing",DrawingArea.class)
10       };
11       return rv;
12     } catch (IntrospectionException e) { throw new Error(e.toString()); }
13   }
14
15   public EventSetDescriptor[] getEventSetDescriptors() {
16     try {
17       EventSetDescriptor mouse=new EventSetDescriptor(DrawingArea.class,
18           "mouseEvent",
19           java.awt.event.MouseListener.class,
20           "mouseEvent");
21       mouse.setDisplayName("mouse movement");
22       EventSetDescriptor[] rv={mouse};
23       return rv;
24     } catch (IntrospectionException e) { throw new Error(e.toString()); }
25   }
26
27   public BeanDescriptor getBeanDescriptor() {
28     BeanDescriptor back = new BeanDescriptor(DrawingArea.class);
29     back.setValue("hidden-state", Boolean.TRUE);
30     return back;
31   }
32
33   public java.awt.Image getIcon(int iconKind) { return null; }
34 }
```

B.2 Enterprise JavaBeans

The following two examples only show the most interesting excerpts of an address book implementation. The complete source code can be downloaded from http://www.infosys.tuwien.ac.at/Staff/tom/addressbook/.

B.2.1 Session Bean Example

Home Interface

```
1  package at.ac.tuwien.infosys.ejb;
2
3  public interface AddressBookHome extends javax.ejb.EJBHome {
4    AddressBook create()
5      throws java.rmi.RemoteException, javax.ejb.CreateException,
6             AddressBookException;
7  }
```

Remote Interface

```
1  package at.ac.tuwien.infosys.ejb;
```

```
2
3   public interface AddressBook extends javax.ejb.EJBObject {
4     public int createAddress(AddressDetails address)
5       throws java.rmi.RemoteException, AddressBookException;
6
7     public java.util.ArrayList getAddressesOfLastName(String lastname)
8       throws java.rmi.RemoteException, AddressBookException;
9
10    public AddressDetails getAddress(int id)
11      throws java.rmi.RemoteException, AddressBookException;
12
13    // ... additional address manipulation methods ...
14  }
```

Bean Class

```
1   package at.ac.tuwien.infosys.ejb;
2
3   public class AddressBookBean implements javax.ejb.SessionBean {
4     /* constant declaration */
5     public static final String ADDRESS_DATABASE="java:/DefaultDS";
6     public static final String ADDRESS_EJBHOME="java:/comp/env/ejb/address";
7
8     /* attributes */
9     private AddressHome addressHome=null;
10
11    /* container interface */
12    public AddressBookBean() {}
13
14    public void ejbCreate() throws AddressBookException {
15      try {
16        InitialContext initial=new InitialContext();
17        Object obj=initial.lookup(ADDRESS_EJBHOME);
18        addressHome=(AddressHome)
19                    PortableRemoteObject.narrow(obj, AddressHome.class);
20      } catch (Exception e) {
21        throw new AddressBookException("lookup of address home failed");
22      }
23    }
24
25    public void setSessionContext(SessionContext sc) {}
26    public void ejbRemove() {}
27
28    public void ejbActivate() {}
29    public void ejbPassivate() {}
30
31    /* remote interface */
32    public int createAddress(AddressDetails address)
33    throws AddressBookException {
34      // ... error handling ...
```

```
35
36      try {
37        int id=nextAddressID();
38        addressHome.create(new Integer(id),
39                           address.getFirstName(), address.getLastName(),
40                           address.getStreet(),    address.getCity(),
41                           address.getState(),     address.getZIP(),
42                           address.getCountry(),   address.getTelephone());
43        return id;
44      } catch(Exception e) {
45        e.printStackTrace();
46        throw new AddressBookException("cannot create address");
47      }
48    }
49
50    public ArrayList getAddressesOfLastName(String lastname)
51    throws AddressBookException {
52      // ... error handling ...
53
54      try {
55        Collection addresses=addressHome.findByLastName(lastname);
56        ArrayList details=new ArrayList();
57
58        for(Iterator i=addresses.iterator();i.hasNext();) {
59          details.add(((Address)i.next()).getDetails());
60        }
61
62        return details;
63      } catch(Exception e) {
64        throw new AddressBookException("cannot retrieve address ids");
65      }
66    }
67
68    public AddressDetails getAddress(int id)
69    throws AddressBookException {
70      try {
71        return fetchAddress(id).getDetails();
72      } catch(Exception e) {
73        throw new AddressBookException("cannot retrieve address");
74      }
75    }
76
77    // ... additional address manipulation methods ...
78  }
```

Deployment Descriptor

The deployment descriptor for the address book session bean is shown as part of the deployment descriptor for the address entity bean.

B.2.2 Entity Bean Example

Home Interface

```
1  package at.ac.tuwien.infosys.ejb;
2
3  public interface AddressHome extends javax.ejb.EJBHome {
4    public Address create(Integer id, String firstname, String lastname,
5                          String street, String city, String state,
6                          String zip, String country, String telephone)
7      throws java.rmi.RemoteException, javax.ejb.CreateException,
8             AddressBookException;
9
10   public Address findByPrimaryKey(Integer id)
11     throws java.rmi.RemoteException, javax.ejb.FinderException;
12
13   public Collection findByLastName(String lastname)
14     throws java.rmi.RemoteException, javax.ejb.FinderException;
15 }
```

Remote Interface

```
1  package at.ac.tuwien.infosys.ejb;
2
3  import java.rmi.RemoteException;
4
5  public interface Address extends javax.ejb.EJBObject {
6
7    public String getFirstName() throws RemoteException;
8    public void setFirstName(String firstname) throws RemoteException;
9
10   public String getLastName() throws RemoteException;
11   public void setLastName(String lastname) throws RemoteException;
12
13   // ... other attributes ...
14
15   public AddressDetails getDetails() throws RemoteException;
16   public void setDetails(AddressDetails address) throws RemoteException;
17
18 }
```

Bean Class

```
1  package at.ac.tuwien.infosys.ejb;
2
3  public class AddressBean implements javax.ejb.EntityBean {
4    public Integer id;
5    public String firstName, lastName,  street,    city;
6    public String state,     zIP        country,   telephone;
7
8    private javax.ejb.EntityContext ctx;
```

```
 9
10     /* container interface */
11     public void setEntityContext(javax.ejb.EntityContext ctx) { this.ctx=ctx; }
12     public void unsetEntityContext() { this.ctx=null; }
13
14     public void ejbLoad() {}
15     public void ejbStore() {}
16
17     public void ejbActivate() {}
18     public void ejbPassivate() {}
19
20     /* home interface */
21     public Integer ejbCreate(Integer id, String firstname, String lastname,
22                              String street, String city, String state,
23                              String zip, String country, String telephone)
24     throws AddressBookException {
25       if(id.intValue()<=0) throw new AddressBookException("illegal primary key");
26       this.id=id;
27       this.firstName=firstname; this.lastName=lastname;
28       this.street=street;       this.city=city;
29       this.state=state;         this.zIP=zip;
30       this.country=country;     this.telephone=telephone;
31       return id;
32     }
33
34     public void ejbPostCreate(Integer id, String firstname, String lastname,
35                               String street, String city, String state,
36                               String zip, String country, String telephone) {}
37
38     public void ejbRemove() {}
39
40     /* business methods */
41     public String getFirstName() { return firstName; }
42     public void setFirstName(String firstname) { this.firstName=firstname; }
43
44     public String getLastName() { return lastName; }
45     public void setLastName(String lastname) { this.lastName=lastname; }
46
47     // ... other attributes ...
48
49     public AddressDetails getDetails() {
50       return new AddressDetails(id.intValue(), firstName, lastName, street,
51                                 city, state, zIP, country, telephone);
52     }
53
54     public void setDetails(AddressDetails address) {
55       firstName=address.getFirstName(); lastName=address.getLastName();
56       street=address.getStreet();       city=address.getCity();
57       state=address.getState();         zIP=address.getZIP();
58       country=address.getCountry();     telephone=address.getTelephone();
```

```
59     }
60   }
```

Deployment Descriptor

```
1   <?xml version="1.0"?>
2
3   <!DOCTYPE ejb-jar PUBLIC
4     '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
5     'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
6
7   <ejb-jar>
8     <description>The Ultimate Adress Book Application</description>
9     <display-name>AddressBook</display-name>
10    <enterprise-beans>
11      <session>
12        <description>Address Book</description>
13        <display-name>AddressBookEJB</display-name>
14        <ejb-name>AddressBookEJB</ejb-name>
15        <home>at.ac.tuwien.infosys.ejb.AddressBookHome</home>
16        <remote>at.ac.tuwien.infosys.ejb.AddressBook</remote>
17        <ejb-class>at.ac.tuwien.infosys.ejb.AddressBookBean</ejb-class>
18        <session-type>Stateful</session-type>
19        <transaction-type>Container</transaction-type>
20        <ejb-ref>
21          <ejb-ref-name>ejb/address</ejb-ref-name>
22          <ejb-ref-type>Entity</ejb-ref-type>
23          <ejb-link>AddressEJB</ejb-link>
24          <home>at.ac.tuwien.infosys.ejb.AddressHome</home>
25          <remote>at.ac.tuwien.infosys.ejb.Address</remote>
26        </ejb-ref>
27        <resource-ref>
28          <res-ref-name>jdbc/AddressDB</res-ref-name>
29          <res-type>javax.sql.DataSource</res-type>
30          <res-auth>Container</res-auth>
31          <res-sharing-scope>Shareable</res-sharing-scope>
32        </resource-ref>
33      </session>
34      <entity>
35        <description>Address Management</description>
36        <display-name>AddressEJB</display-name>
37        <ejb-name>AddressEJB</ejb-name>
38        <home>at.ac.tuwien.infosys.ejb.AddressHome</home>
39        <remote>at.ac.tuwien.infosys.ejb.Address</remote>
40        <ejb-class>at.ac.tuwien.infosys.ejb.AddressBean</ejb-class>
41        <persistence-type>Container</persistence-type>
42        <prim-key-class>java.lang.Integer</prim-key-class>
43        <reentrant>False</reentrant>
44        <cmp-version>2.x</cmp-version>
45        <abstract-schema-name>address</abstract-schema-name>
```

```
46        <cmp-field><field-name>id</field-name></cmp-field>
47        <cmp-field><field-name>firstName</field-name></cmp-field>
48        <cmp-field><field-name>lastName</field-name></cmp-field>
49        <cmp-field><field-name>street</field-name></cmp-field>
50        <cmp-field><field-name>city</field-name></cmp-field>
51        <cmp-field><field-name>state</field-name></cmp-field>
52        <cmp-field><field-name>zIP</field-name></cmp-field>
53        <cmp-field><field-name>country</field-name></cmp-field>
54        <cmp-field><field-name>telephone</field-name></cmp-field>
55        <primkey-field>id</primkey-field>
56        <query>
57          <query-method>
58            <method-name>findByLastName</method-name>
59            <method-params>
60              <method-param>java.lang.String</method-param>
61            </method-params>
62          </query-method>
63          <ejb-ql>
64            <![CDATA[WHERE lastName = ?1]]>
65          </ejb-ql>
66        </query>
67      </entity>
68    </enterprise-beans>
69    <assembly-descriptor>
70      <container-transaction>
71        <method>
72          <ejb-name>AddressBookEJB</ejb-name>
73          <method-intf>Remote</method-intf>
74          <method-name>*</method-name>
75        </method>
76        <trans-attribute>Required</trans-attribute>
77      </container-transaction>
78      <container-transaction>
79        <method>
80          <ejb-name>AddressEJB</ejb-name>
81          <method-intf>Remote</method-intf>
82          <method-name>*</method-name>
83        </method>
84        <trans-attribute>Required</trans-attribute>
85      </container-transaction>
86    </assembly-descriptor>
87  </ejb-jar>
```

B.2.3 Message Driven Bean Example

Bean Class

```
1  package org.jboss.docs.jms.mdb.bean;
2
```

```
3   public class HelloMDB
4   implements javax.ejb.MessageDrivenBean, javax.jms.MessageListener {
5     private javax.ejb.MessageDrivenContext ctx = null;
6
7     public HelloMDB() {}
8
9     public void setMessageDrivenContext(javax.ejb.MessageDrivenContext ctx)
10    throws javax.ejb.EJBException {
11      this.ctx = ctx;
12    }
13
14    public void ejbCreate() {}
15    public void ejbRemove() { ctx=null; }
16
17    public void onMessage(javax.jms.Message message) {
18      System.err.println("Bean got message" + message.toString());
19    }
20  }
```

Deployment Descriptor

```
1   <?xml version="1.0"?>
2   <!DOCTYPE ejb-jar>
3   <ejb-jar>
4     <enterprise-beans>  <message-driven>
5         <ejb-name>HelloTopicMDB</ejb-name>
6         <ejb-class>org.jboss.docs.jms.mdb.bean.HelloMDB</ejb-class>
7         <message-selector></message-selector>
8         <transaction-type>Container</transaction-type>
9         <message-driven-destination>
10          <destination-type>javax.jms.Topic</destination-type>
11          <subscription-durability>NonDurable</subscription-durability>
12        </message-driven-destination>
13    </message-driven>  </enterprise-beans>
14    <assembly-descriptor>
15      <container-transaction>
16        <method>
17          <ejb-name>HelloTopicMDB</ejb-name>
18          <method-name>*</method-name>
19        </method>
20        <trans-attribute>NotSupported</trans-attribute>
21      </container-transaction>
22    </assembly-descriptor>
23  </ejb-jar>
```

Bibliography

[ABV92] Mehmet Akşit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *ECOOP'92 European Conference on Object-Oriented Programming*, pages 372–395. Springer-Verlag, 1992.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AG97] Ken Arnold and James Gosling. *The Java Programming Language (Java Series)*. Addison-Wesley, 2nd edition, December 1997.

[ALNH01] Uwe Assmann, Andreas Ludwig, Rainer Neumann, and Dirk Heuzeroth. Compost, September 2001. http://i44www.info.uni-karlsruhe.de/~compost/.

[ALSN01] Franz Achermann, Markus Lumpe, Jean-Guy Schneider, and Oscar Nierstrasz. Piccola — a small composition language. In Howard Bowman and John Derrick, editors, *Formal Methods for Distributed Processing, an Object Oriented Approach*. Cambridge University Press, 2001. to appear.

[BCRW00] Don Batory, Gang Chen, Eric Robertson, and Tao Wang. Design wizards and visual programming environments for genvoca generators. *IEEE Transactions on Software Engineering*, 26(5):441–452, May 2000.

[BDS00] Dirk Balfanz, Drew Dean, and Mike Spreitzer. A security infrastructure for distributed java applications. In *Proceedings of 2000 IEEE Symposium on Security and Privacy*, pages 15–26. IEEE, May 2000.

[Bea92] Brian W. Beach. Connecting software components with declarative glue. In *Proceedings of the 14th International Conference on Software Engineering*, pages 120–137. ACM, 1992.

[Bea96] David M. Beazley. Swig: An easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the Fourth USENIX Tcl/Tk Workshop*. USENIX, July 1996.

[BEA01] BEA. *BEA Weblogic Server and Weblogic Express: Introduction to BEA Weblogic Server*, June 2001.

[BH89] Fred Buckley and Frank Harary. *Distance in Graphs.* Addison-Wesley, 1989.

[BPSM98] Tim Bray, Jean Paoli, and C. Michael Sperberg-McQueen. Extensible markup language (XML) 1.0. Technical Report REC-xml-19980210, W3C, February 1998.

[BSST93] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. In *Proceedings of the ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering*, pages 191–199, December 1993.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[Cox90] B. J. Cox. Planning the software industrial revolution. *IEEE Software*, 7(6):25–33, November 1990.

[DAM02] The darpa agent markup language homepage (daml), January 2002. http://www.daml.org/.

[DGT96] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation.* Lecture Notes in Computer Science. Springer-Verlag, 1996.

[Dij76] Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[DYK01] Linda G. DeMichiel, L. Ümit Yal cinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2.0.* Sun Microsystems, April 2001. Proposed Final Draft 2.

[EE98] Guy Eddon and Henry Eddon. *Inside Distributed COM.* Microsoft Press, 1998.

[FPV98] Alfonso Fuggetta, Gian P. Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[FS98] Martin Fowler and Kendall Scott. *UML Distilled.* Addison-Wesley, June 1998.

[Gal96] Mark Galassi. *Guile Programmer's Manual*, July 1996. For use with Cygnus Guile 1.0.

[GEG+02] Pankaj K. Garg, Kave Eshghi, Thomas Gschwind, Boudewijn Haverkort, and Katinka Wolter. Enabling network caching of dynamic web objects. In *Proceedings of the Performance TOOLS2002 Conference*. Springer-Verlag, 2002. To be published.

[GEGW01] Thomas Gschwind, Kave Eshghi, Pankaj K. Garg, and Klaus Wurster. Web transaction monitoring. Technical Report HPL-2001-62, Hewlett-Packard Laboratories Palo Alto, March 2001.

[GFP99] Thomas Gschwind, Metin Feridun, and Stefan Pleisch. Adk—building mobile agents for network and systems management from reusable components. In *Proceedings of the Agent Systems Application and Mobile Agents '99 Conference*, pages 13–21. IEEE Computer Society, October 1999.

[GG01] Thomas Gschwind and Pankaj K. Garg. Dynamic interface adaptation, October 2001. Invention disclosure.

[GH99] Thomas Gschwind and Manfred Hauswirth. Newscache—a high-performance cache implementation for usenet news. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 213–224. USENIX, June 1999.

[GHH+] Bill Griswold, Erik Hilsdale, Jim Hugunin, Mik Kersten, Gregor Kiczales, and Jeff Palm. The AspectJ homepage. http://www.aspectj.org/.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, January 1995.

[Gon98] Li Gong. Secure Java Class Loading. *IEEE Internet Computing*, 2(6):56–61, November/December 1998.

[Gon99] Li Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 1999.

[Goo01] Gerhard Goos. Optimizations with COMPOST, February 2001. Personal communication.

[Grü00] Andreas Grünbacher. Dynamic distributed systems. Master's thesis, Technische Universität Wien, June 2000.

[Gsc00a] Thomas Gschwind. The agentbean development kit, 2000. http://www.infosys.tuwien.ac.at/ADK/.

[Gsc00b] Thomas Gschwind. Comparing object-oriented mobile agent systems. Technical report, Technische Universität Wien, May 2000. Presented at the 14th European Conference on Object-Oriented Programming (ECOOP2000), Workshop on Mobile Object Systems: Operating System Support, Security and Programming Models.

[Gsc01a] Thomas Gschwind. Aop and software maintenance. Technical Report TUV-1841-01-07, Technische Universität Wien, May 2001.

[Gsc01b] Thomas Gschwind. PSTL—the persistent standard template library for c++. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technology Systems (COOTS 2001)*, pages 147–158. USENIX, January 2001.

[Ham97] Graham Hamilton, editor. *JavaBeans*. Sun Microsystems, http://-java.sun.com/beans/, July 1997.

[Har01] Elliotte Rusty Harold. *XML Bible*. Hungry Minds, Inc, 2nd edition, 2001.

[HBS99] Mark Hapner, Rich Burridge, and Rahul Sharma. *Java Message Service*. Sun Microsystems, November 1999.

[HKK00] Manfred Hauswirth, Clemens Kerer, and Roman Kurmanowytsch. A secure framework for java. In *Proceedings of the 7th ACM Conference on Computer and communications security*, pages 43–52. ACM, 2000.

[HO99] George T. Heineman and Helgo M. Ohlenbusch. An evaluation of component adaptation techniques. Technical Report WPI-CS-TR-98-20, Worcester Polytechnic Institute, Computer Science Department, March 1999.

[ISO98] ISO/IEC. *ISO/IEC14882: Programming Languages—C++*, 1st edition, July 1998.

[Jak02] Stefan Jakl. The calendar agent. Master's thesis, Technische Universität Wien, 2002. To be published.

[Jaz95] Mehdi Jazayeri. Component programming—a fresh look at software components. In Wilhelm Schäfer and Pere Botella, editors, *Proceedings of the 5th European Software Engineering Conference*, pages 457–478. Springer-Verlag, 1995.

[JBo] JBoss Group. The jboss homepage. http://www.jboss.org/.

[JLM00] Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser, editors. *Generic Programming*. Lecture Notes in Computer Science. Springer-Verlag, 2000.

[Kic96] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11, January 1996.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object Oriented Programming (ECOOP'97)*, pages 220–242. Springer-Verlag, 1997.

[LH00] Andreas Ludwig and Dirk Heuzeroth. Metaprogramming in the large. In *Net.Objectdays 2000 Tagungsband, 2nd International Conference on Generative and Component-Based Software ENgineering*, pages 443–452, October 2000.

[Lis87] Barbara H. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1987.

[LSNA97] Markus Lumpe, Jean-Guy Schneider, Oscar Nierstrasz, and Franz Achermann. Towards a formal composition language. In Gary T. Leavens and Murali Sitaraman, editors, *Proceedings of the ESEC'97 Workshop on Foundations of Component-Based Systems*, pages 178–187, September 1997.

[Lud01] Andreas Ludwig. *RECODER Technical Manual*, April 2001. http://recoder.sourceforge.net/doc/manual.html.

[LW93] Barbara H. Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications '93*, September 1993.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, April 1999.

[MBB+98] Dejan Milojicic, Markus Breugst, Ingo Busse, John Campbell, Stefan Covaci, Barry Friedman, Kazu Kosaka, Danny Lange, Kouichi Ono, Misuru Oshima, Cynthia Tham, Sankar Virdhagriswaran, and Jim White. MASIF, The OMG Mobile Agent System Interoperability Facility. In Kurt Rothermel and Fritz Hohl, editors, *Proceedings of the Mobile Agents'98*. Springer-Verlag, September 1998.

[Mey88] Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall, 1988.

[Mey92] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[MH00] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, 2nd edition, March 2000.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.

[MSL00] Mira Mezini, L. Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. In Mehmet Aksit, editor, *2000 Symposium on Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Publishers, 2000.

[NM95] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In *Proceedings of the ECOOP '94 workshop on Models and Languages for Coordination of Parallelism and Distribution*, pages 147–161. Springer-Verlag, 1995.

[NT95] Oscar Nierstrasz and Dennis Tsichritzis. *Object-Oriented Software Composition*. Prentice Hall, 1995.

[Obe01] Johann Oberleitner. The component workbench: A flexible component composition environment. Master's thesis, Technische Universität Wien, October 2001.

[Obe02] Johann Oberleitner. The component workbench. http://-www.infosys.tuwien.ac.at/cwb/, February 2002.

[OMG99a] Object Management Group. *CORBA Components—Volume I*, August 1999. OMG TC Document `orbos/99-07-01`.

[OMG99b] Object Management Group. *CORBA Components—Volume II: MOF-based Metamodels*, August 1999. OMG TC Document `orbos/99-07-02`.

[OMG99c] Object Management Group. *CORBA Components—Volume III: Interface Repository*, August 1999. OMG TC Document `orbos/99-07-03`.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Ous98] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[PA90] James M. Purtilo and Joanne M. Atlee. Improving module reuse by interface adaptation. In *Proceedings of the International Conference on Computer Languages*, pages 208–217, March 1990.

[PA91] James M. Purtilo and Joanne M. Atlee. Module reuse by interface adaptation. *Software — Practice and Experience*, 21(6):539–556, June 1991.

[Par76] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(16):1–9, March 1976.

[Pur94] James M. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.

[Ros99] Guido Van Rossum. *Python Reference Manual*, February 1999.

[Sam97] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, 1997.

[SGML01] Sibylle Schupp, Douglas P. Gregor, David R. Musser, and Shin-Ming Liu. User-extensible simplification—type-based optimizer generators. In Reinhard Wilhelm, editor, *International Conference on Compiler Construction*, Lecture Notes in Computer Science, pages 86–101. Springer-Verlag, 2001.

[SGS98] Glenn Smith, John Gough, and Clemens Szyperski. Conciliation: The adaptation of independently developed components. In Gopal Gupta and Hong Shen, editors, *Proceedings of the 2nd International Conference on Parallel and Distributed Computing and Networks*. IASTED, 1998.

[SL96] Jean-Guy Schneider and Markus Lumpe. Modelling objects in PICT. Technical Report IAM-96-004, University of Berne, Software Composition Group, 1996.

[SLL02] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[SML99] Linda Seiter, Mira Mezini, and Karl Lieberherr. Dynamic component gluing. In Ulrich Eisenecker and Krzysztof Czarnecki, editors, *First International Symposium on Generative and Component-Based Software Engineering*, Lecture Notes in Computer Science, pages 134–164. Springer-Verlag, 1999.

[Sta93] William Stallings. *SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards*. Addison-Wesley, 1993.

[Sta99] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, 13th edition, February 1999. Updated for Emacs Version 20.7.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.

[Sun98a] Sun Microsystems. The bean development kit, July 1998. http://-java.sun.com/beans/software/bdk_download.html.

[Sun98b] Sun Microsystems. Reflection, 1998. http://java.sun.com/j2se/1.3/docs/-guide/reflection/.

[Sun99a] Sun Microsystems. *Java Remote Method Invocation Specification*, December 1999. http://java.sun.com/products/jdk/1.3/docs/guide/rmi/.

[Sun99b] Sun Microsystems. *Jini Architectural Overview*, 1999. Technical White Paper.

[Sun01] Sun Microsystems. *JSR-000057 Long-term Persistence for JavaBeans^{TM} Specification*, November 2001. http://jcp.org/jsr/detail/57.jsp.

[Szy97] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, January 1997.

[WCD+01] Sanjiva Weerawarana, Francisco Curbera, Matthew J. Duftler, David A. Epstein, and Joseph Kesselman. Bean markup language: A composition language for javabeans components. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technology Systems (COOTS 2001)*, pages 173–187. USENIX, January 2001.

[WCO00] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly & Associates, 3rd edition, July 2000.

[Weg90] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

[Weg93] Peter Wegner. Towards component-based software technology. Technical Report CS-93-11, Brown University, 1993.

[WS91] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, 1st edition, January 1991.

Curriculum Vitae

Thomas Gschwind <tom@infosys.tuwien.ac.at>

Address

Institut für Informationssysteme
Technische Universität Wien
Argentinierstraße 8/E1841
A-1040 Wien, Austria
Tel: +43(1)58801-18412

Date of Birth

December 26, 1973

Education

PhD Student, Technische Universitaet Wien, since 1997. Working on the adaptation and composition of software components with focus on distributed systems.

Master of Science, Technische Universitaet Wien, June 1997, with highest distinction.
The Thesis (Grade A) 'A Cache Server for News' describes how cache servers can contribute to a new and optimized architecture for USENET News. It presents a prototype cache implementation (NewsCache) and the implementation of a persistent object library for this purpose.

Graduated from Gymnasium Wien III, Wien, June 1992, with distinction (Subjects: Mathematics, German, English, Physics, Computer Science).

More Information

Upon request.