Type Based Adaptation: An Adaptation Approach for Dynamic Distributed Systems

Thomas Gschwind

Technische Universität Wien
Institut für Informationssysteme
Argentinierstraße 8/E184-1
A-1040 Wien, Austria
tom@infosys.tuwien.ac.at
http://www.infosys.tuwien.ac.at/Staff/tom/

Abstract. Recently, component models have received much attention from the Software Engineering research community. The goal of each of these models is to increase reuse and to simplify the implementation and composition of new software. While all these models focus on the specification and packaging of components, however, they provide almost no support for their adaptation and composition. This work still has to be done programmatically. In this paper we present Type Based Adaptation, a novel adaptation technique that uses the type information available about a component. We also describe the design and implementation of our reference implementation thereby verifying the feasibility of this approach.

1 Introduction

Today's component models can be distinguished between *server side* component models and *local* component models. Local component models describe pieces of software similar to libraries that can be used by builder tools to create an application. Server side component models, however, describe components similar to services that can be accessed by other components. In this article, we will focus on server side component models.

Most component models available today, such as the CORBA Component Model (CCM) [21–23], the JavaBeans component model [13], the Enterprise JavaBeans (EJB) component model [5, 20], COM [6], or .NET, rely on black box components with well-defined and publicly available interfaces. Based on the knowledge of these interfaces, the components can be composed to interact with each other. One component, for instance, might request a weather service from a naming or trading service. If the interface provided by the weather service matches the interface expected by the requesting component interaction between them is possible. Otherwise, there is currently no means for the two components

to interact with each other. We propose to address this problem using *type-based adaptation*. In particular, we are interested in *Dynamic Distributed Systems* [12] where the communication patterns have not been defined *a priori* and thus the interfaces provided by a service are not known to the client until run-time.

Type based adaptation builds on top of the interface a component *expects* and the interface *provided* by a component, information provided by modern component models. In case of server side component models such as the CORBA component model (CCM) or the Enterprise JavaBeans component model (EJB) the interfaces provided by a component is the only type information available. We will show, how to extract information about the expected interfaces.

Type-based adaptation does not rely on any additional description of the semantics of the component as could be provided by a semantic description framework such as the DARPA Agent Markup Language (DAML) [4]. Semantic description frameworks are not yet readily available and rely heavily on standardization.

The outline of this paper is as follows. In Section 2 we present a running example of a sample application that can use type-based adaptation. Section 3 shows the model underlying our adaptation approach. Based on this, we discuss design issues in Section 4 and describe an implementation in Section 5 along with its evaluation in Section 6. Possible extension and future work are considered in Section 7. In Section 8 we present related work and we draw our conclusions in Section 9.

2 Motivation

In a dynamic distributed system the communication patterns between its components are not determined *a priori*. This allows the system to adapt more easily to the users' needs. For the implementation of such a system a middleware layer such as provided by CORBA or Enterprise JavaBeans (EJBs) is used. While both CORBA and EJBs provide server side components they lack the possibility to adapt the components available. Thus, component a can only interact with component b if b provides the interface expected by a. As shown by the following example, the flexibility of a component model can be improved by the provision of a transparent adaptation mechanism such as type based adaptation.

The use case [7] in Figure 1 shows an *Internet Superstore* where the *User* browses and selects products available. After the *User* has selected the products he wishes to buy, he selects the shipping address from an *Addressbook Service* with which the *User* 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 addressbook service. This, however, is inconvenient because the *User* has to manage a different addressbook for each store he wishes to buy goods from. He would prefer to use an external addressbook service to manage his addresses and simply instruct the *Internet Superstore* to use the addressbook provided by that service.

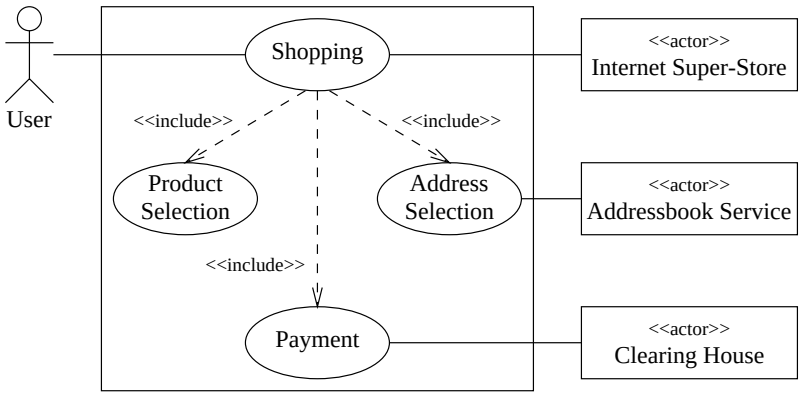

Fig. 1. Internet Shopping Application

As long as the *Addressbook Service* exactly implements 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 is lost. With multiple service providers, however, 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 Addressbook service) to match the interface required by another (e.g., the *Internet Superstore*).

3 Adaptation Model

Server-side components consist of multiple interfaces, each represented by a different object [21]. For a component using another component only the interfaces provided by the other component are important. Since an interface defines a contract that specifies the behavior [18], type-based adaptation uses this type information as the basis of whether a component can be substituted with another. The mapping information that define how one interface has to be mapped into another, however, has to be provided by a human capable of understanding the different interfaces that need to be mapped. A human has to decide whether two interfaces can be translated into each other, and, if so, to specify their translation.

Only the presence of this information makes automated adaptation possible. To provide mapping information type based adaptation uses *adapters*. These adapters algorithmically describe how to translate different interfaces into each other. The interfaces provided and expected by the components are the only type information required. The adapters themselves are written in typical programming languages such as C++ or Java. Finally, the adapters are stored in a repository with some meta-information about the adapters such as the interfaces they translate.

Our approach can be seen as an extension of the adapter pattern [9, 19]. The difference, however, is that the adapters are first-class objects described on their own and that an adapter repository exists having full knowledge about the adapters available and the transformations they describe. The repository stores information such as the interfaces the adapters translate, and optionally their performance characteristics or whether their translation is lossless. Based on this information the adaptation-process can be automated.

In case of 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. Thus, the implementation of an adapter A that translates from an interface I_{from} to an interface I_{to} is straightforward.

In fact, code as it needs to be written for these adapters exists already 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 especially when subclassing is being used, the adapter is part of a bigger component and thus cannot exist on its own. To be used in combination with the adapter repository this 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 automatically in other systems where similar interface transformations are necessary.

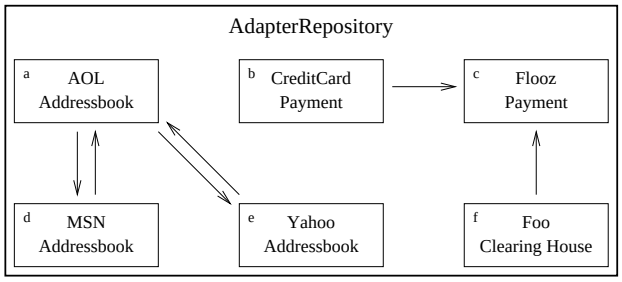

Fig. 2. A Sample Adapter Repository

As shown by the sample adapter-repository in Figure 2, an adapter repository forms a directed graph G where the interfaces are represented by the set of vertices $V(G) = \{a, b, c, d, e, f\}$ and the adapters by the set of edges $E(G) = \{ad, da, ae, ea, bc, fc\}$ between the interfaces they can translate. In case an adapter required is missing adapters can be combined. For instance da and ae can be combined to simulate an adapter de. To find a suitable combination of adapters a simple shortest path algorithm is sufficient. Additionally, the algorithm can be tuned to prefer adapters of specific characteristics by applying different weights to each edge (adapter). For instance, if the user were interested in a lossless adaptation, the weight of lossy adapters should be set to ∞.

We propose to implement this algorithm in a separate adaptation component that can be used by a client or naming service. 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. This gives the user the impression of having automatic composition at hand. Only if no such adapter exists, the composition of the components cannot be performed automatically and the user has to 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.

A different approach to the adaptation problem is the use of a semantic description framework that allows to describe the interface, its methods and data structures using a common ontology. Such an approach might be implemented using DAML [4]. We claim, however, that this approach only adds another level of indirection since no commonly accepted ontology for the semantic description exists. Different ontologies, however, will be unavoidable since companies are unwilling to release any information about their products in their early development states. Such information, however, is crucial for the standardization of a common ontology.

4 Design Issues

In a dynamic distributed system, each component implements a different service and can be executed by a different computer as shown in Figure 3. 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 4. Now, the client can interact with the service returned by the naming service. If the service requested by the client, however, does not match the interface expected by the client an exception will be thrown and the client will be unable to interact with the service.

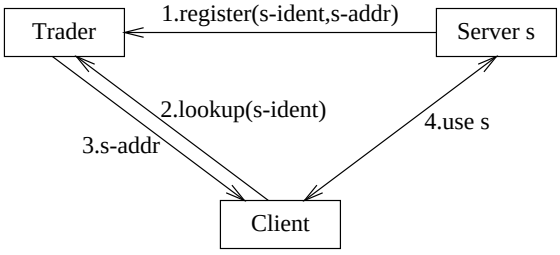

Fig. 3. Accessing a Service in a Distributed Component System

```
try {
  Context ctx=getInitialContext();
  Object dobj=ctx.lookup("CookieServer");
  CookieHome ch=(CookieHome)
      PortableRemoteObject.narrow(dobj,
      CookieHome.class);
  Cookie c=ch.create();
  System.out.println(c.getCookie());
} catch(Exception e) {
  e.printStackTrace();
}
```

Fig. 4. Typical Client Code in a Distributed Component System

Instead of throwing an exception it is better to use an adaptation component responsible for the transparent instantiation of the adapters when necessary. An ideal location for this is the client since typically the naming service or the service provider do not know the interface expected by the client. Still, it might make sense to add adaptation support to a trading service if the query language allows clients to request a component by its interface.

Since an adaptation component has to be available to the client and probably to the naming service, a straightforward approach would be to locate an adapter repository at both sites. While placing an adaptation component at each site is fine we recommend to use a central adapter repository or better to link the various adapter repositories. Hence, the adapters have to be pieces of mobile code [8] loaded on demand by the adaptation component.

While this approach increases the number of adapters available and the number of interfaces that can be translated into each other it has to be used in combination with mobile code. The security risks posed by mobile code can be solved by the following two approaches. First, the adapters should be executed within a safe sandbox environment. The Java Virtual Machine [17], for instance, can provide such an environment that does not allow the downloaded code to execute arbitrary instructions. Additionally, much work exists in the context of Java and Mobile Agents that can be reused for our approach [1, 10, 11, 15].

Additionally, we have to determine where the adapters should be executed. We recommend the adapters to be executed by the party that wants the two components to interact with each other, typically, the client. This pushes the security risk and the adapter's resource consumption to the party benefiting from the composition. Thus, the service providers do not suffer any disadvantage.

5 Implementation

To evaluate the feasability of our approach, we have implemented type based adaptation for the Enterprise JavaBeans (EJB) component model [5, 20]. Most of the implementation decisions, however, can be applied to other component

models as well. EJB is a server side component model that provides support for security, persistence, and transaction management allowing the developer to focus on the application logic [5].

A typical Enterprise JavaBean consists of a home interface serving as the component's factory, a remote interface specifying the component's functionality and the component's implementation. After an EJB has been deployed in an EJB server, the EJB server provides a container for each component that takes care of the component's lifecycle and the interaction with its clients. For a client, however, an EJB component looks similar to an RMI service. Figure 4 shows the typical client code to obtain a reference to the home interface, to create a new instance of an EJB component and to invoke one of its method.

Since our reference implementation is based on Java which simulates a homogenous environment little additional support has to be provided to download the adapters on demand from a central repository. As discussed in the previous sections the following items must be considered for an implementation of type based adaptation:

Adapter Repository This component is responsible to store the adapters and to provide lookup operations based on their meta-information.

Adaptation Component This component is responsible to compute an optimal chain of adapters.

Adapter Description Each adapter comes with a description of the interfaces it can translate.

Packaging Each adapter is stored in its own package along with the adapters description.

5.1 The Adapter Repository

The adapter repository itself is straightforward since it only has to store the adapters and their descriptions. Any data structure that can store a graph with parallel edges (multiple edges connecting the same two nodes) is sufficient. Since the graph, however, will consist of a large number of blocks (independent components within the graph) of equivalent interfaces, we have chosen to use an adjacency list that stores all the out-edges for a given vertex v. For the lookup of adapters we chose to use Dijkstra's shortest path algorithm which has a complexity of $O(|E(G)|)$ which typically is much smaller than $O(|V(G)|^2)$ [3].

5.2 The Adaptation Component

The code in Figure 4 illustrates that the narrow operation (a system independent cast operation) is the perfect place for the adaptation component. 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 transparently plug type based adaptation into an existing system by upgrading the middleware layer only.

Table 1. Methods Provided by the Adaptation Component

`Object getAdapter(Object from, Class to)` Instantiates an adapter that provides the interface `to` to the client and interacts with the service represented by `from`. The object returned is the adapted object.
`Adapter getAdapter(Class from, Class to)` This method looks up an adapter or combination of adapters that provide the interface `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.

The adaptation component only has to provide lookup operations as shown in Table 1 to be used by the middleware layer's narrow operation. While the first method returns an already instantiated adapter, the second method returns an object describing a combination of adapters. This object provides methods that wrap a service with the according adapter.

Frequently, a trading service also allows the client to lookup a service based on some properties. For instance, a travel agent might request any component that implements the `at.ac.tuwien.Weather` interface to display weather information on its web site. If the naming service supports type based adaptation, it can even 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 is registered at the naming service.

Table 2. Methods provided by the Naming Service

`Object lookup(Class to)` Lookup a service that provides the interface `to` to its clients or can be translated to that interface.

Since the JBoss EJB Server [16] we used for the reference implementation does not provide such a naming service we have implemented our own supporting the lookup of a component based on the interface it provides (Table 2). If multiple servers with the same interface are registered they are returned in a round robin manner.

5.3 Adapter Description

The description of an adapter has to specify the interface the adapter translates from and the interface the adapter maps to. Additionally, it should be possible to supply additional information about the adapter such as whether it performs a lossless translation, the adapter's performance complexity or other properties.

This description could be maintained by the adapter class itself and could be be accessed using introspection as it is used by the JavaBeans component model [13]. Alternatively, it could be stored externally using a configuration file.

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

<adapter name="SampleAdapter">
<mapsfrom>
<interface>
  com.yahoo.AddressDatabase
</interface>
</mapsfrom>
<mapsto>
<interface>
  com.amazon.AddressBook
</interface>
</mapsto>

<implementation type="classname">
  at.ac.tuwien.infosys.tba.SampleAdapter
</implementation>

<lossless>false</lossless>
</adapter>
```

Fig. 5. Sample Adapter

We have decided to support both choices. This allows us to keep the adapters simple as well as to extend type based adaptation to other application domains where the use of introspection might not be possible. For the configuration file we have chosen to use XML [2, 14]. Using XML has the advantage that existing tools can be used to parse the adapter specification. Only a document type definition or XML Schema has to be provided that describes the syntax of the adapter's specification.

A sample adapter description is shown in Figure 5. The `SampleAdapter` converts from a fictitious `com.yahoo.AddressDatabase` interface to a fictitious `com.amazon.AddressBook` interface. Additionally, the specification indicates that the transformation is not lossy and that the adapter's implementation is provided by the class `at.ac.tuwien.infosys.tba.SampleAdapter`. Thus, when the adapter is applied some information about an address might be lost.

5.4 Packaging

To simplify the adapter's installation and transfer to another site we have decided to put all the class and resource files required for the adapter into a single archive. In general, we recommend to use a format similar to the format already exploited by one of the existing component models. Since our implementation is based on Java, we have chosen to use a *.jar*-archive for the reference implementation. In case of the CORBA Component Model, we recommend to use a *.zip*-file containing the adapters and their specification files.

6 Evaluation

We implemented a weather service and an addressbook service that we used for experimentations with our system. Both systems were implemented using Enterprise JavaBeans [5]. For the weather service we implemented a set of different weather services providing access to weather information along with a set of adapters able to convert between them. Finally, we implemented a client to see whether our implementation was able to provide for a transparent adaptation of the different weather services.

Our system was able to transparently adapt the different weather components. In some cases, however, the adapters were unable to provide some information expected by the client. This is due to the fact that the adapter cannot generate information that is not provided by the component it adapts. This problem, however, could be solved by allowing the adapter to contact several different services to collect the required information.

For the addressbook service we downloaded and installed the petstore and ebank web applications from Sun's website. Both applications were extended to allow the customer to specify our external addressbook service instead of having to type in the shipping and mailing addresses over and over again. This extension allows the user to specify a location of an external addressbook service to be used. After the user has specified the external addressbook, the web application contacts the user's addressbook service and presents a list of the mailing and shipping addresses to the user.

In this scenario, however, the petstore application acts as a client of the addressbook service and thus executes the adapters. While the adaptation was performed as expected, for security reasons, however, we think that the adapter should be executed by the web browser. Otherwise, the web browser's user might be able to inject malicious code on the petstore server provided it has access to the adapter repository. Additionally, executing the adapter within the web browser would allow the user to control what address data is transferred to the web application and thus increases the user's privacy. This issue, however, will be attacked in future versions of our implementation.

7 Future Work

While we have only presented type-based adaptation in the context of dynamic distributed systems, it can be extended to other application domains as well. For instance, it could be used in combination with more traditional software development tools such as *Integrated Development Environments* (*IDEs*). In a typical IDE components are composed by selecting an event a component can generate and specifying the method or connector to be executed. Usually, this connector has to be implemented by the *Developer*. Otherwise, the interaction between the components would be limited to the execution of existing methods matching the event's signature [13].

Type based adaptation eases this problem by allowing the user 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 can be reused for each newly created connection. For instance, a connector toggling a certain property could be reused fairly frequently. For instance in a drawing application to indicate whether the pen is drawing or not.

It might also be possible to use type based adaptation to integrate a component in different component environments. To identify interfaces across different components, however, a uniform type identifier consisting of the component model as well as the interface would have to be introduced. In this case, the adapter would not only provide the code for translating between the interfaces but also the bridge-code necessary to translate one protocol into the other. Before we will be able to attack this problem, however, it is necessary to gain more experience with type based adaptation.

One issue we have not resolved yet is the performance degredation if multiple adapters need to be combined. We assume, however, that in a typical situation most of the computation takes place within the components. Additionally, we expect that typically no more than two or three adapters will have to be combined. Thus, the intercommunication between the components is less important to be optimized.

Even though performance is less important it should not be ignored entirely. If several adapters need to be combined partial evaluation could be used to fuse the adapters together. This inlines a large number of method calls and thus lowers the performance penalty if a large number of adapters are combined. Additionally, constants passed from one adapter to another can result in the elimination of a considerable number of redundant computations.

8 Related Work

The interworking problem between different components has already been identified by the NIMBLE [25] language which is part of the Polylith [24] system and by the conciliation approach presented in [27]. Unlike conciliation, NIMBLE does not take the object-oriented view into account and solely operates on a procedural level. Compared to type-based adaptation, however, their adaptation is

static and their adaptations cannot be chained. In dynamic distributed systems, however, it is important that the adaptation is performed at run-time since the components that need to interoperate with each other cannot be known a-priori.

Another approach using adapters is the composite adapter design pattern presented in [26]. 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 enable 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 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 (`adapter`) construct is proposed.

Another interesting adaptation approach is used by Jini. Jini uses proxies responsible to interact 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 is responsible to shield the application from having to deal with the device's wire protocol. This allows Jini to access devices from different component environments as long as an according proxy exists. An architectural overview of Jini can be found in [28].

9 Conclusions

The contribution of this paper is a simple but flexible and powerful adaptation technique that enables the automated adaptation of software components. Since the adapatation can be performed during run-time, as we have shown, it is the ideal adaptation approach for server-side component models.

Type based adaptation, the adaptation technique presented in this paper, uses an adapter repository that stores prebuilt adapters. These adapters specify how one interface can be translated into another. Hence, the repository contains all the information necessary for an automatic adaptation process. To some extent the repository can be compared to semantic description frameworks [4] but using an algorithmic approach to specify the interface's translation instead. One advantage is that only the adapter repository needs to be standardized whereas for semantic description languages the terminology to describe the semantic of each application domain needs to be standardized. Another advantage is that multiple adapters can be combined to build a more powerful one. Hence, it is not necessary to provide adapters for all the different combinations.

We have demonstrated the feasibility of type based adaptation in combination with dynamic distributed component systems. Our results, however, indicate that it can be used in a variety of other application domains as well. We have implemented a prototype for dynamic distributed component systems that operates on a per-interface level and enables automatic adaptation. As we have shown with our examples automatic adaptation is important for dynamic dis-

tributed systems since it allows services as already available on the Internet to give more flexibility to their customers by allowing them to decide what services should interact with each other.

Acknowledgements

I would like to thank Mehdi Jazayeri for his continuous support and Pankaj Garg, my supervisor during my internship with HP labs, for his help and support of this project.

This work was supported in part by an IBM University Partnership Award from IBM Research Division, Zurich Research Laboratories and the European Union as part of the EASYCOMP project (IST-1999-14191). This work was in part done during an internship with Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA-94304.

References

1. Dirk Balfanz, Drew Dean, and Mike Spreitzer. A security infrastructure for distributed java applications. In *Proceedings of 2000 IEEE Symposium on Security and Privacy*. IEEE, 2000.
2. Tim Bray, Jean Paoli, and C. Michael Sperberg-McQueen. Extensible markup language (xml) 1.0. Technical Report REC-xml-19980210, W3C, February 1998.
3. Fred Buckley and Frank Harary. *Distance in Graphs*. Addison-Wesley, 1989.
4. The darpa agent markup language homepage (daml). http://www.daml.org/.
5. Linda G. DeMichiel, L. Ümit Yalcinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2.0*. Sun Microsystems, April 2001. Proposed Final Draft 2.
6. Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Press, 1998.
7. Martin Fowler and Kendall Scott. *UML Distilled*. Addison-Wesley, June 1998.
8. Alfonso Fuggetta, Gian P. Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5), 1998.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, January 1995.
10. Li Gong. Secure Java Class Loading. *IEEE Internet Computing*, 2(6):56–61, November/December 1998.
11. Li Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 1999.
12. Andreas Grünbacher. Dynamic distributed systems. Master's thesis, Technische Universität Wien, June 2000.
13. Graham Hamilton, editor. *JavaBeans*. Sun Microsystems, http://java.sun.com/-beans/, July 1997.
14. Elliotte Rusty Harold. *XML Bible*. Hungry Minds, Inc, 2nd edition, 2001.
15. 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.
16. JBoss Group. The jboss homepage. http://www.jboss.org/.

17. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, April 1999.
18. Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.
19. 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.
20. Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, 2nd edition, March 2000.
21. OMG. *CORBA Components—Volume I*, August 1999. OMG TC Document `orbos/99-07-01`.
22. OMG. *CORBA Components—Volume II: MOF-based Metamodels*, August 1999. OMG TC Document `orbos/99-07-02`.
23. OMG. *CORBA Components—Volume III: Interface Repository*, August 1999. OMG TC Document `orbos/99-07-03`.
24. James M. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.
25. 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.
26. 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. Springer, 1999.
27. 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.
28. Sun Microsystems. *Jini Architectural Overview*, 1999. Technical White Paper.