Dynamic Component Adaptation

Kai-Uwe Mätzel
Ubilab
Union Bank of Switzerland
Bahnhofstr. 45
CH-8021 Zurich
Switzerland
kai-uwe.maetzel@ubs.com /
maetzel@acm.org

Peter Schnorf
Credit Suisse / Os 1
CH-8070 Zurich
Switzerland
schnorf@acm.org

Ubilab Technical Report 97-6-1

Abstract

Component-based systems are mostly composed out of existing artifacts. Missing pieces are newly developed. Interoperability between the existing artifacts is usually not given. The newly developed artifacts should be well reusable in future developments. The resulting systems should be easily evolvable. In component-based systems, a key to solve the problems and meet the requirements is by providing an infrastructure that makes it easy to bridge the gaps between components and to ensure their interoperability. In this paper, we discuss the issues of dynamic interface adaptation starting with a discussion about design issues along the three dimensions integration, reusability, and evolution. We elaborate on what it takes to perform dynamic interface adaptation and present design and implementation of a framework that covers these issues.

1 Introduction

Today's software business defines strong requirements. At least in commercial settings, there is always the necessity to integrate legacy systems, middleware infrastructures, or third party products with newly developed systems. The user is increasingly demanding. Systems have to be continually accessible, reliable, and, therefore, fault-tolerant. They must be dynamically reconfigurable and evolvable in order to retain their 24-by-7 nature. Versioning of system components becomes a major issue because global software updates are no longer possible. Multiple versions of the same software component can be used in the same system. This all leads to systems of higher complexity.

Furthermore, new systems must have an ever shorter time to market and strong budget limitations. With respect to future developments and their even tighter time schedules, the newly developed cost-intensive system parts have to be better reusable than in former days.

Overall we can say that today's large-scale software design and development is faced with the challenge to develop dynamically changeable and evolvable systems which seamlessly integrate legacy systems and are built out of reusable system components.

Systems meeting these requirements usually cannot follow a single design decision but have to be a sensible mixture of complementing design decisions at various levels of abstraction. The design decisions about system architecture, component design, and inter-component connections have to be coordinated along three closely related but distinct dimensions:

- integration
- reusability
- evolution.

In order to be applicable in a commercial, industrial-strength environment, the resulting implementations of these design decisions have to meet further requirements:

- They must be useful for developing new systems as well as improving existing ones. Design ideas can incrementally be brought into an existing system. This can be done either by using just a few important concepts of a design idea or by using these concepts only in particular parts of the system.
- Applying design ideas has to be inexpensive for the developers. This means that, for example, the developer has to be supported with an easy to learn programming environment such as a small object-oriented framework that implements the necessary design guidelines. The developer should not be bothered with learning complex tools, a new programming language, or a difficult new design methodology.

In this paper, we present a small but powerful framework for dynamic component adaptation. It defines important concepts of the system- and component architecture as well as various inter-component connection mechanisms. The dynamic component adaptation framework has been used as a key technology to implement systems according to Design for Slippage [MB96].

We first discuss design issues along the three major dimensions. We argue that the capability of dynamic component adaptation is a major enabler for improving the system's quality in the three dimensions. Therefore, we focus on dynamic adaptation and discuss how it can be accomplished. We discuss the dynamic component adaptation framework by means of its Smalltalk implementation and present an example how to use the framework. Finally, we discuss related work and summarize the paper's content.

2 Design Issues

In this section, we discuss design issues along the three dimensions evolution, reusability and integration. First, we informally define a few terms by means of a system model that we will refer to in our discussion.

We consider a system to be consisting of a finite set of components. The set can dynamically change. There are no assumptions about the granularity of components. A component is an entity with a persistent identity, interfaces, and state. We distinguish two different kinds of interfaces: *provided* and *required* (Fig. 1). A provided interface is a message protocol that specifies what services the component provides to clients. Components are almost never self-contained; they usually interact with other components. The complete set of components which a particular component requires in order to be operational is called the context of the component. The context of a component is represented inside the component by means of required interfaces. With other words, the set of required interfaces of a component describes its subjective view onto its context. Required interfaces are message protocols as well. Although this does not belong to the literal definition of the term "interface" interfaces usually associate a particular semantics with the defined message protocols. On an informal level interfaces are defining abstract types. In addition, interfaces can inherit from other interfaces. The inheritance hierarchy forms an acyclic graph. Our understanding of inheritance between interfaces is similar to the inheritance known from IDL [OMG96] or Java Interfaces [GYS96].

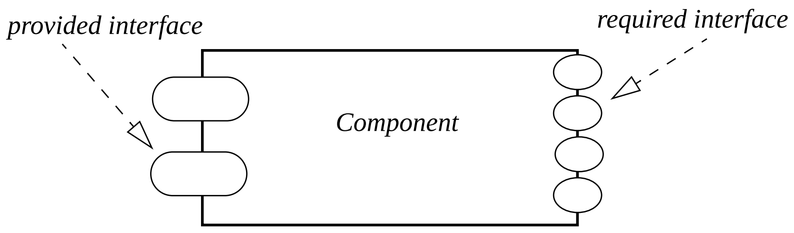

Fig. 1: Component Model

Based on this system model we can now discuss various design issues to improve the quality of software systems according to the mentioned three dimensions.

Integration. In order to use an external system (e.g. existing legacy system, middleware, or third party product) effectively, it has to be successfully integrated with the newly developed system. There are two different facets of integration. Either the external system is extended or modified by means of new code, or a system exploits external systems as particular service providers. In both cases, it is necessary that the external system is causally connected with the component system. There must be a sensible and operational representation of the external system within the component system.

According to the model above, this means that the external system plays the role of a component which is activated using the client's required interfaces. In practice, there is usually a separate component which internally acts as proxy of the external system. Proxies usually bridge the architectural mismatch between external and internal world by being able to interact with both in a semantically correct fashion.

On a design level, the question how does a proxy for a given external system look like is one of the most important ones. This question is not easily to answer. Usually, external systems, especially legacy systems provide a broad range of functionality. This fact actually makes them such a valuable asset. Typically, clients use only a small fraction of the functionality and, therefore, need a limited reflection or understanding of the used system. Each client has its own individual view of a used external system. The proxy can either be universal by almost completely representing the system's functionality or it is specific for the subjective view of a particular set of clients. The latter implies the existence of a set of proxies of the same external system as well as the need to dynamically change this set according to the dynamics of the system under development.

Reusability. Designing proxies of external systems deals with the issue of making the proxies and, at the same time, external systems reusable in a given context. Reusability of components depends on numerous factors. Some of the more important ones are the degree of generality of the provided functionality, the number of implicit assumptions about the calling context, the completeness and soundness of the component's provided interface, the number and complexity of its required interfaces, as well as the capability to influence and change all these factors.

Independent of the actually provided functionality, the less implicit assumptions a component embodies and the smaller and simpler its required interfaces are the better is a component reusable. Therefore, in order to improve reusability, components should explicitly reflect all assumptions about the calling context as well as having an explicit representation of required interfaces. Furthermore, components should be equipped with mechanisms that give them the opportunity to manipulate their provided interfaces from the client's point of view and to adapt their required interfaces to given service providers.

Evolution. Software systems evolve over time because of expected or unexpected requirement changes. The resulting modifications of a system can be classified into structural and functional changes. Structural change comprises all changes of the system configuration. Functional change comprises the construction of new components and the modification of component functionality. According to this classification, we distinguish *compositional* and *component evolution*. Furthermore, we distinguish *anticipated* and *unanticipated evolution*. Anticipated evolution is caused by expected requirement changes and can therefore, be appropriately considered by the system design. Unanticipated evolution is caused by unexpected requirement changes. This means, it is impossible to predict at design time which requirements will change in which way and which system parts will be affected. Both, compositional and component evolution can be either anticipated or unanticipated. Integration and reuse are closely connected to evolution. Thus, the discussion of evolution comprises the issues of integration and reusability as well.

Design for Slippage, as discussed in [MB96], provides a conceptual system model which leads to better evolvable systems. It adapts Steward Brand's idea of "shearing architectural layers" [Bra94] to software architecture. Design for Slippage proposes software systems consisting of various *regions* of components where each layer has its own distinct pace of evolution. Building a system according to Design for Slippage means to determine the appropriate number of layers and to map all system parts to them according to their expected evolution rate. This decision is specific for the particular system at hand and cannot be made generically. For this reason, Design for Slippage consist of only two mandatory regions: the Design Core and the Context. The Design Core is the "slowest" region. It is considered static and dominates the architecture. It defines the major architectural and design decisions. Everything outside the Design Core belongs to the Context and is considered a potential subject of change. The Context can be either a single region or a compound region. Examples of change are: replacing a Context component by a new version of this component, dynamically changing a Context component, or reconfiguring the Context.

The static nature of the Design Core and the dynamics of the Context raise the issue of collaboration mismatch between their components. Collaboration mismatch is caused by syntactical and/or semantic differences between a client component's required interface and the actually provided one of a serving component. Thus, Design for Slippage calls for coupling mechanisms between components with various degrees of flexibility and dynamic behavior. These mechanisms must bridge the existing mismatch by making the provided interface plug-compatible with the required interface. They absorb changes at the edges of the Design Core and between components of the Context by focusing on connection management and communication. This includes the finding of a component providing an interface that matches a particular required interface within a certain tolerance, binding the related components, and performing the message exchange between them.

Fig. 2 schematically depicts a Design for Slippage example consisting of Design Core and two Context regions. A line between two components indicates coupling between these components, the line's width correlates with the required flexibility of the coupling. The stronger the line is, the less flexibility is required.

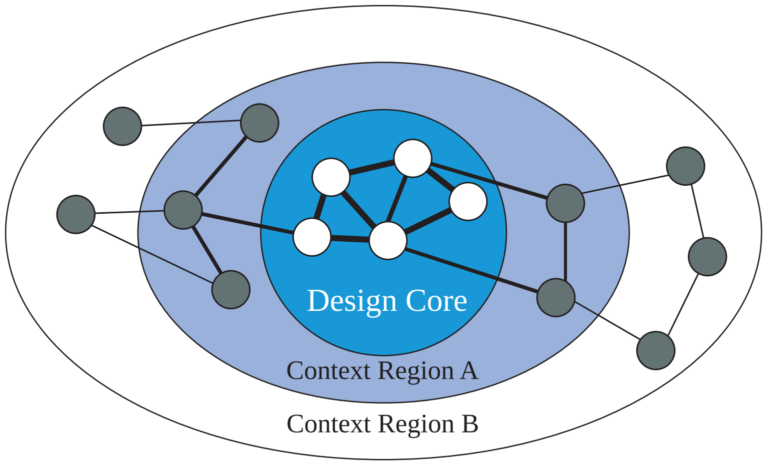

Fig. 2: Design for Slippage

Design for Slippage can be used for a few fine-grained objects, a system of coarse-grained components, or complex client-server installations. It meets the business requirements listed in Section 1.

By advocating this system model, Design for Slippage favors compositional evolution over component evolution. The flexible coupling mechanisms focus on dynamically bridging the collaboration and the architectural mismatches. There are various coupling mechanisms such as self-describing requests, anonymous communication channels, or traders.

Applying Design for Slippage raises various issues. It obviously promotes dynamic systems that inherently suffer from their potential for runtime failures and which are usually slower than static systems. Applying Design for Slippage is always a trade-off between necessary flexibility, runtime efficiency, and decreasing the probability of runtime failures. Efficiency can be consider to be less serious than robustness, because time-critical functionality usually resides inside the Design Core or one component. Therefore, efficiency-critical calls which involve dynamic coupling mechanisms are considered infrequent.

Dynamic adaptation. Design for Slippage advocates utilizing flexible coupling mechanisms between components of different regions. However, there is no generic algorithm which perfectly matches differing required and provided interfaces by considering the underlying semantics of the interfaces [BGR96]. It is necessary to provide dynamically customizable and extensible bridging mechanisms. One such mechanism are adapters as depicted in Fig. 3.

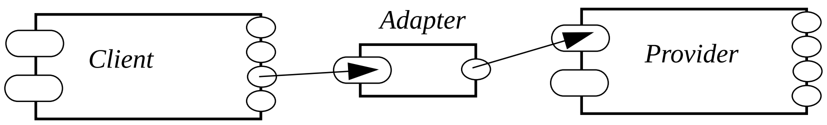

Fig. 3: Dynamic adaptation by adapters

Adapters are components which implement a mapping function between a required and provided interface by taking the necessary semantic transformations into account. Adapters have to be provided by developers. However, a developer should not be bothered with the details of installing an appropriate adapter between two components. It must be possible to install and remove adapters at runtime. Adaptation can be performed either per component or per interface. Considering the requirements of a real life software development process, it seems clear that dynamic adaptation on a per-interface basis is sufficiently powerful for almost all cases.

3 A Component Adaptation Architecture

If dynamic component adaptation is used as a design principle in the production of software systems that are tolerant against collaboration mismatch certain assistance from the environment is necessary. Without assistance the architecture is reduced to just guidelines and recipes and, therefore, remains hard to control. The more of the architecture that can be implemented and offered to programmers for reuse the better. As is typical in object-oriented environments a *framework* will be most helpful in this situation. A framework can pre-define and concretely implement an architecture as a set of interacting classes providing hooks for white-box reuse and components for black-box reuse to developers. Here, we discuss the general requirements and design issues for such a framework. In a later section, we describe a concrete implementation in Smalltalk.

3.1 Requirements

The major constructs for component adaptation must be explicitly expressible in the programming language and in the programming/runtime environment. Since features and expressiveness of languages vary widely the representations and mechanisms used in concrete implementations can be quite different. The purpose of this section is to establish the necessary abstract requirements that are independent of language or implementation choices.

Code written by clients will have to refer to *interfaces*. We therefore need an explicit representation of interfaces. It must be possible to express which interfaces some component provides and which it requires. In essence, it must be possible to establish ‘provides’ and ‘requires’ relationships between a component and a list of interfaces. If interface inheritance is to be supported a parent-child relationship between interfaces must be expressible as well. In addition to these relationships, an interface representation must contain a unique identity and a version number.

The second construct that requires explicit representation are *adapters*. An adapter maps the protocol defined by an interface to one or several other protocols defined by other interfaces or components. It must therefore be possible to specify for every adapter an interface it maps from and a list of interfaces/components it maps to. Every adapter instance wraps an instance of the object it delegates to.

A *component* not only provides the interfaces it defines itself but also to the ones its parents in the class hierarchy defines. Equivalently, a component not only requires the interfaces it defines but also to the ones of its parents. These component inheritance relationships must be represented at runtime because the algorithms that dynamically search for matching adapters need that information.

We finally need two *algorithm implementations.* The first algorithm, the *converter,* converts a given component to particular interface by deterministically matching the given component/interface pair to an adapter on demand. The algorithm will rely on the runtime representations for interfaces, adapters and components, e.g., it will have to find out at runtime which interfaces a given component supports and which interfaces a given adapter maps to. The second algorithm, the *binder*, deterministically binds a given interface to an implementation instance. This implementation instance is either a component or a component wrapped by adaptors. We describe the algorithm issues in the next subsection.

The requirements for interface adaptation support in a framework can now be summarized as follows:

- explicit development and runtime representation of interfaces including versioning information and, optionally, parent-child relationships for interface inheritance
- explicit development and runtime representation of adapters including mapping relationships from an interface to a list of interfaces/components
- possibility to express the provided and required interfaces of a component
- runtime access to the component inheritance relationships
- an algorithm implementation that deterministically finds adapters for a given interface/component pair
- an algorithm implementation that deterministically binds an interface to a component instance

3.2 Algorithm issues

The converter algorithm to find an adapter for a given pair of interface and component must fulfill certain requirements and solve some tricky issues. Most importantly, the algorithm must take inheritance structures into account. Assume, e.g., a component C adheres to an interface I. Then C also adheres to all interfaces that I inherits from. C also adheres to all the interfaces its own superclasses adhere to. This can easily lead to situations where the set of possibly matching adapters contains more than one element. Therefore a strategy for conflict resolution needs to be defined. We found that the following specification is both natural and adequate as a default.

The algorithm has two arguments: an interface and a component object. It returns either the object itself or an adapter instance that understands the given interface protocol and wraps the given object. Two cases are distinguished in the following order:

1. The given interface or one of its sub-interfaces is directly supported by the component or one of its superclasses. In this case, no adapter is needed because the component object already understands the interface's protocol. The component object itself is returned.

2. The component or one of its superclasses is directly supported in one of the given interface's adapters. The returned adapter instance contains a reference to the component object it wraps. The adapter is chosen to be the most closely matching, most specific adapter class according to the following order:

   a) adapters explicitly supporting the component match more closely than those supporting the component's superclass, etc.

   b) among the closest matching adapters, those at deeper levels in the adapter hierarchy take precedence

   c) if there remains more than one adapter choose an arbitrary one

In other words, directly supported interfaces take precedence over adapters. Among adapters, those mapping directly to the component win over those mapping only to its superclasses. From the winners the most specific subclass is chosen.

It must be possible to specify adapters to interfaces, i.e. adapters that map one interface's protocol to another interface's protocol. This introduces a transitive relationship: if an adapter A1 maps an interface I1 to a component C and an adapter A2 maps interface I2 to I1 (or A1) then we can concatenate the two adapters so that A2 maps I2 to C with one level of indirection (via A1). In order for this to be well defined we need to specify one more precedence order:

3. Lower levels of adapter indirection win, e.g. direct adapters take precedence over indirect adapters.

This algorithm serves well as a default implementation. It is desirable however to give clients more control over the search and decision process. The strategies how the set of all matching adapters is generated and how the actual adapter is chosen from that set can be made customizable by delegating the work to strategy objects that are definable by clients. We refer to these objects as adapter-lookup strategies.

The binder algorithm must solve problems similar to the converter algorithm. The difference is that the input consist of only an interface. The default binder interprets the described conflict resolution strategy as instruction how to find a component, either wrapped be adapters or not, which implements the interface. This means, the binder first inspects all component classes. If there is no appropriate one, it takes the inheritance hierarchy of the components into account. The next step is a single indirection with adaptors, followed by the inheritance hierarchy of the adaptors, followed by multiple indirections, and so forth. If there are multiple equally rated implementation instances, the default binder makes an arbitrary choice. Finally, a new component instance is created.

3.3 Guidelines and trade-off

The use of adapters does not come for free, of course. The typical overhead can be summarized as an indirection for every method call to a wrapped component as well as one call to create the adapter in the first place. The cost of adapter creation varies with the complexity of the relationships between the involved interfaces, adapters, and components. In any case, the result of the search for the right adapter type can be cached to reduce the following calls with the same interface/component combination to essentially a dictionary lookup. The intended purpose of the adapter mechanism is for slippage tolerant communication between components, i.e. between relatively high-level building blocks. The incurred overhead is the price for the flexibility but should be negligible at that level.

The proposed component adaptation architecture does not make any further assumptions about the component model than the mentioned ones. Especially, it is independent from whether components can be distributed or not. However, a few additional design issues need to be discussed:
- adapter and component placing,
- adapter-lookup-strategy placing and replication.

Adapter and component placing. Assuming the situation that a component C1 wants to talk to component C2 using interface I1. C2 does not support I1 directly but there is an adapter A1 from I1 to C2's provided interface. Then there are three alternatives to place A1: A1 can be collocated either with C1 or with C2 or with neither. Which one of these alternatives is most recommendable depends on:
- the interaction behavior of C1,
- the interaction behavior of C2,
- the complexity of A1,
- the component system support for component migration.

Finding the right place for an adapter according to these factors is a particular kind of load balancing question. We will not discuss it in detail here but give an example of a placing strategy: If A1 is highly complex then communication overhead can be neglected. A1 has to be placed where it can be executed most efficiently. If A1 is not complex then communication overhead has to be considered. If C1 and C2 are collocated, A1 has to be collocated as well. Otherwise, the placement of A1 depends on whether C1 or C2 is more loaded at the point of the interaction between C1 and C2. A1 has to be collocated with the component that is in less demand. If the component system supports migration the decision about the placement of A1 depends only on the system state at the decision's point of time. Otherwise, the decision depends on the average state of the whole duration of the C1 to C2 interaction.

Assuming the situation that C1 does not already know C2, the binder has to create/find C2 and to announce it to C1. In this case, there are similar issues regarding the placing of C2 as in the first scenario regarding the placing of A1.

Adapter-lookup-strategy placing and replication. Placing the adapter-lookup-strategy is rather similar to placing adapters. In contrast to adapters, replication is a further dimension along designers have to decide in this case. Like load balancing, replication is a well studied problem in the area of distributed systems.

4 The Smalltalk Adapter Framework

We implemented the laid-out adapter architecture as a framework in Smalltalk and C++ as well as for CORBA [OMG96]. In this section we will describe the Smalltalk framework because it emphasizes the relevant aspects most comprehensively. Some characteristics of Smalltalk are noteworthy with respect to such a framework. Smalltalk does not know interfaces as a standard language construct nor does it support multiple inheritance. At runtime however, introspection features are supported that allow clients access to the class hierarchy.

4.1 General requirements mapping

Interface requirements. If a language supports interfaces many requirements like, e.g., specification of provided interfaces are naturally taken care of. Otherwise we must find a – necessarily imperfect – way to express interfaces through other language means. In an object-oriented language, a natural and straight-forward representation for an interface is as a class containing public instance methods only. In such a representation, interface inheritance will be automatically supported by sub-classing. Depending on the language and the amount of desired sophistication, different ways exist to add the capability to express that a component supports or expects a certain interface. One simple possibility is to define a class method in a common component base class that returns the interfaces as a collection of names. Subclasses override that method to return the interfaces they themselves support or expect.

Certain environments provide built-in versioning support. If such support is not available versioning information can be added to interface classes through a class variable and a read-only access method. The burden of maintaining the values of these variables then lies on the developer.

Adapter requirements. Since adapters need to implement the protocol of an interface they have a natural representation as subclass of an interface (or a class implementing an interface). An adapter simply overrides the interface methods and implements them either directly or by delegating to the object it wraps. Again, besides having adapters explicitly represented it is also necessary to give developers the capability to express which interface an adapter maps to which interfaces/components. If we stick to the above adapter representation the interface that gets mapped is the superclass (or the implemented interface). The list of classes the interface gets mapped to can be defined in various ways, e.g. simply – and similarly as above for interface definitions – in a polymorphic method that returns a collection.

Defining adapters as subclasses of interface classes has the advantage that it easily matches a developer's paradigm. Subclassing and overriding methods is a well known process. In addition, the programming language or environment sometimes provides support for checking that an adapter actually implements all methods defined in the interface. It is very well conceivable, however, that adapter and interface hierarchies are kept unrelated in terms of language means and that the actual relations are explicitly maintained in framework data structures.

Runtime requirements. The defined algorithms need access to the inheritance relationships of interfaces, adapters and components. In programming languages that offer runtime introspection a framework can take full advantage of these introspection capabilities if appropriate representations – like adapters as subclasses of interfaces – were chosen. Without introspection features, additional work is needed in the framework to build up the inheritance structures at runtime. In this case, it is also likely that some help from the developers of interfaces, adapters, and components is required, e.g. to explicitly register a class/superclass pair at class initialization time.

4.2 The actual framework

A relatively straight-forward mapping of the above requirements to a Smalltalk framework resulted in the structure outlined in Fig. 4. The framework provides a class IFInterface as the common superclass for all interfaces. Interfaces are subclasses of IFInterface with – by convention – only public instance methods and no state. Adapters are subclasses of interfaces overriding methods to provide concrete implementations of these methods. Adapter classes inherit a private instance creation method from IFInterface that takes the component object to be wrapped as an argument and stores it in a predefined instance variable. This variable can be accessed in methods that need to delegate to the wrapped object (`#component`). Adapters are not meant to be created by clients. Rather the framework creates adapters when needed. An adapter class must explicitly declare which classes it can adapt. IFInterface defines a method `#supportedComponents` that returns an empty collection. Adapters override that method to return a collection of supported classes (as symbols).

Components need a common protocol as well. It is, however, not advisable to expect clients to derive all their components from one single, predefined class. Without multiple inheritance, all that can be done is to keep the required protocol as narrow as feasible and the implementation thereof as simple as possible. In our framework, a class IFBase defines that protocol. Most of its methods simply delegate their work to a registry that is provided by the framework as a black-box component. Through this indirection, clients are isolated from implementation changes even if they have to replicate the protocol in the root class of their own class hierarchy. Because adapters can play the roles of components for other interfaces as well they inherit that behavior from IFBase by making IFInterface a subclass of IFBase. Components declare which provided and which required interfaces they support by overriding the IFBase methods `#supportedInInterfaces` and `#supportedOutInterfaces`.

An instance of a component can be asked via the method `#withInterface:for:` to convert itself into an object that responds to all messages specified in the interface class (symbol) given as first parameter. The second parameter specifies for which component the converted object will be used, i.e. to which target component it will be passed as an argument. This will provide all the necessary information to the matching algorithm: it can not only find an adapter for that interface but for the *correct version* of that interface as expected by the target component and specified in the target component's required interfaces. If a target component is not specified (the second parameter is optional) the latest version of the interface is chosen. Again, that method is defined by IFBase. It triggers the adapter matching converter algorithm and results in an adapter creation if necessary. Repeated identical calls return identical objects.

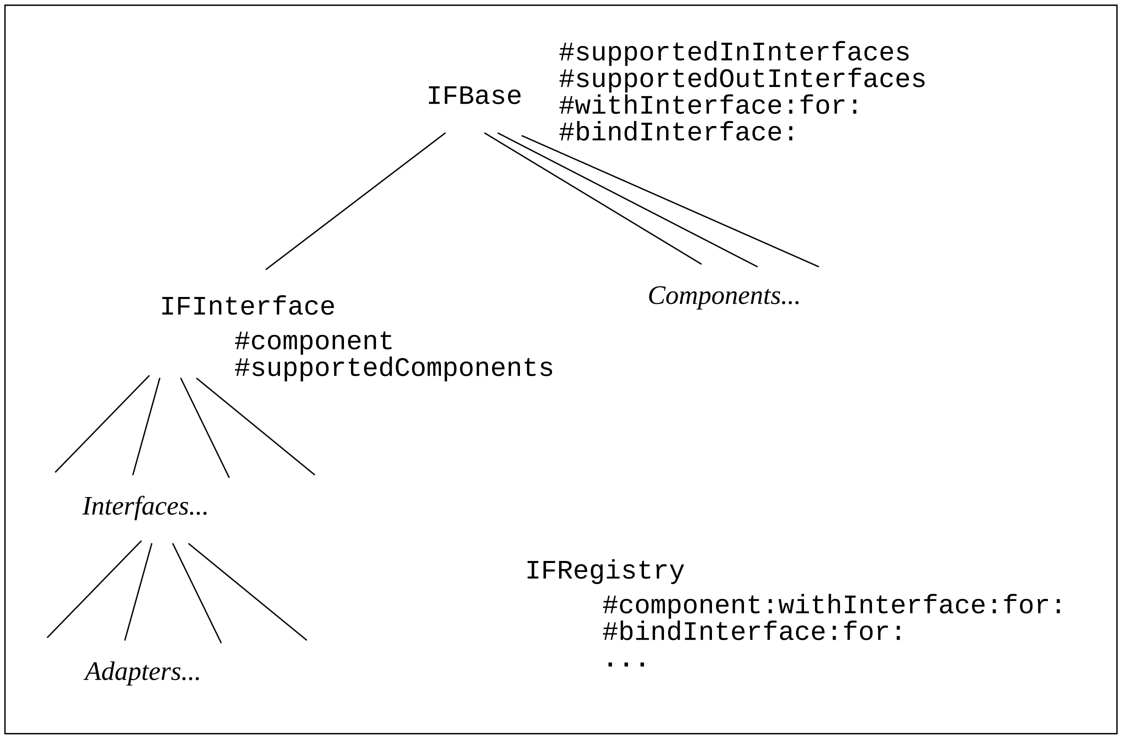

Fig. 4: Inheritance hierarchy

Interfaces are classes as well. Clients can create interface instances and treat them as regular components. Since they do not define any functionality, interfaces have to be bound to implementation instances. This can be done by calling the `#bindInterface:` method with the component which uses the interface as parameter. As in interface conversion, this information is required in order to provide an implementation of the correct version of an interface.

4.3 Example

Assume a situation where a simple error reporting component is written from scratch. The component should be able to report error messages and include a description of the component in which the error occurred. Concretely it provides a method `#reportError:fromComponent:` to report an error, e.g. by displaying a dialog. The method has two parameters: an error specification and the component instance in which the reported problem occurred.

The error reporting component expects the passed-in faulty component to adhere to a certain simple interface that allows a client to get a short description string identifying the object somehow. Typically, not all existing components and not all those loaded in the future will support such a protocol. With the adapter framework, the writer of the error reporting component can specify the expected protocol as an interface (SelfDescribingComponent, see Fig. 5) and declare it as a required interface of the component.

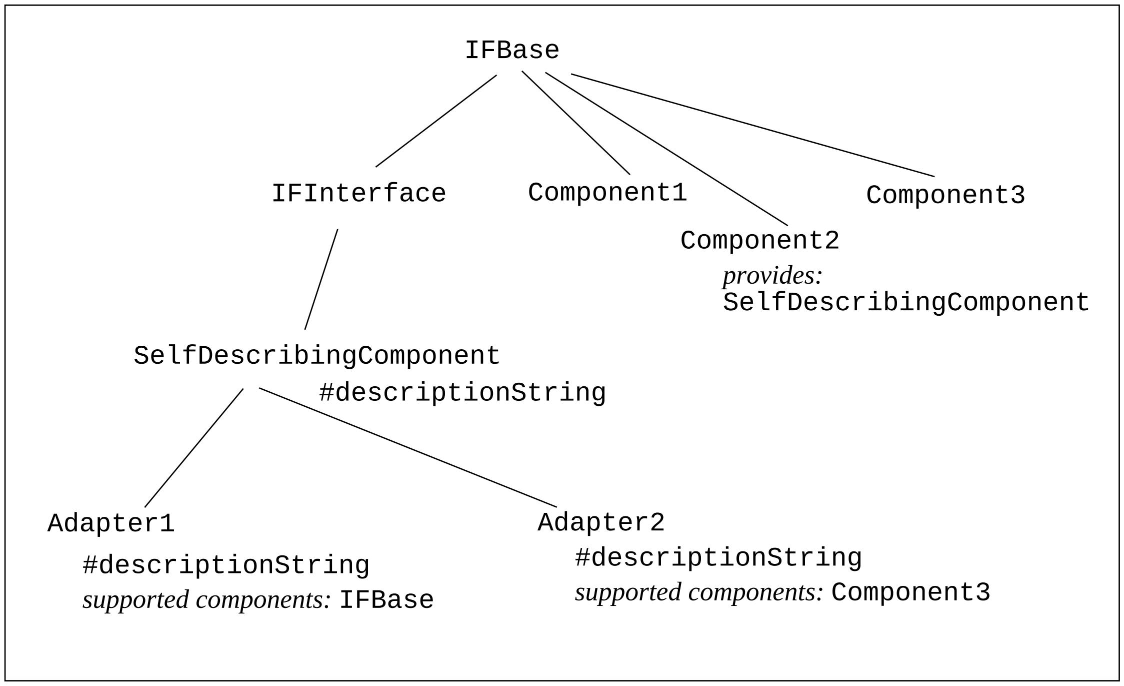

Fig. 5: Example

The SelfDescribingComponent defines one instance method `#descriptionString` that is supposed to return a readable description identifying the receiver. An adapter subclass (Adapter1) can provide a reasonable default for all possible components (get the wrapped object's class name):

```
#descriptionString
        "return a short description of receiver"
        ^'an instance of class', self component class name
```

The adapter Adapter1 announces that fact by returning IFBase from its `#supportedComponents` method. Consequently, every possible component (deriving from IFBase) can be passed to the error component by first converting it through `#withInterface:for:` to an object adhering to the expected protocol. Specific adapters (like Adapter2) for certain components can still be written and components supporting the required `#descriptionString` method can prevent adapter creation by adding the SelfDescribingComponent to their provided interfaces (like Component2).

5 Related Work

Work related to the dynamic component adaptation framework can be divided into the three categories:

- work sharing the same goal
- systems focusing on particular aspects of dynamic component adaptation
- systems that can easily be extended with mechanisms for dynamic component adaptation.

Trading and mapping operators. The Reference Model for Open Distributed Processing (RM-ODP) [ISO93a] describes trading as one of its integrated parts. Trading is the process of matching service offers with service requests. A service offer describes the services a component provides by means of the supported interfaces together with their dynamic properties. Similarly, a service request specifies a service a component requires to be operational. A trader is a component which facilitates dynamic rendezvous between clients and services for the purpose of dynamic binding [ISO93b]. The RM-ODP does not define the trading strategy. Both, the conversion as well as the binding algorithm of our framework can be considered particular kinds of trading.

In [SU96], Senivongse and Utting propose a model for evolution of services in distributed systems based on RM-ODP. They introduce mapping operators that can bridge the difference between two different versions of the same service. Although mapping operators are restricted on service versions they support a wide range of changes to an interface type that makes them rather similar to the adapters of our framework. In contrast to our system, their model does not support inheritance which is essential to our system.

Subject-Oriented Programming. Subject-Oriented Programming (SOP) focuses on the development and the evolution of suites of cooperating applications [HO93]. Applications can show a highly varying granularity. Hence, we can consider such applications components. SOP's goal is that components can be independently developed and evolved, combined to suites, and that these suites can be dynamically changed, for example by adding new components or replacing existing ones. Thus, our goal and that of SOP are identical.

Components are built by using subjects and subject compositions. A subject means "a collection of state and behavior specifications reflecting a particular gestalt, perception of the world at large, such as is seen by a particular application or tool." Subjects define an subjective view onto existing objects or collections of objects. "A subject ... corresponds to a traditional (tough usually incomplete) class hierarchy, describing the interfaces and classes known to this subject" [HO93]. Subjects can be combined in order to form compositions. Each composition defines a composition rule specifying the combination of the subjects' partial class hierarchies.

Regarding component adaptation, SOP is highly powerful. The component and its adapters might be considered subjects. Subject composition then merges the original component as well as one of its adapter using an appropriate composition rule. However, subject composition is an active process whereas in the described system adapters are installed dynamically and transparently. SOP is closely related to role modeling [Ree96]. Adapters could be used as explicit implementations of roles.

Adapter generation. In this paper, we do not explain how the construction of adapters is supported. Although adapters cannot be generated completely automatically, partial adapter generation is the most convenient way to construct them. The prototype implementation of Senivongse's and Utting's service evolution model contains a evolution manager [SU96] that is used to interactively generate mapping operators based on a structural comparison of the interface definitions of the two different versions of a service. Another example of adapter generation is the NIMBLE parameter mapping language [PA91]. NIMBLE allows the developer to declare how the actual parameter list of a procedure call has to be transformed in order to match the formal parameter list of the called procedure. The NIMBLE compiler generates an adapter from this pattern declaration.

Module interconnection languages provide comparable functionality [PN86]. The allow a developer to describe how modules are to be combined and to specify interface mapping.

Other related systems. In Section 3.1, we described the requirements of the dynamic adaptation framework. We argued that besides classes the representation of interfaces and adapters is indispensable. These representations present a sufficient basis to implement the dynamic adaptation. Therefore, any system which provides mechanisms similar to these presentations are well suited for dynamic adaptation. Examples are, Mircrosoft's Component Object Model [MD95], Java [GYS96], or any reflective system [Mae87] such as CodA [McA95], [McA96], SOM [Lau94], or Geo [BGR96], [BR96].

6 Summary

Interoperability between components is a major issue in software development. Interoperability means plug-compatibility between the involved components regarding interfaces as well as semantics [Weg96]. To achieve interoperability it is often necessary to bridge the existing interface and architectural mismatch by means of adapters. In component systems, entirely new, unknown, or unanticipated components may be loaded dynamically to running programs. Therefore, adapters have to be provided dynamically as well. A flexible and powerful adapter system considerably improves the quality of a system along the dimensions integration, reuse, and evolution.

Dynamic interface adaptation exists in two facets. First, a client wants to treat a given component according to a particular interface. Second, a client is looking for a component whose provided interface can be matched to a particular required interface. The dynamic component adaptation framework, presented in this paper, supports both facets. The framework has been developed in the context of our efforts to effectively apply Design for Slippage. Components participating in the dynamic adaptation process provided by the framework must explicitly declare to which interfaces they adhere. On demand, a component can support all interfaces for which proper adapters exist. The adapter-lookup strategy appropriately considers the component and interface inheritance hierarchy and dynamically constructs adapter chains if necessary.

The framework is small, comprehensible, and powerful. Its major parts are:

- explicit notion and runtime representation of interfaces and their hierarchy
- explicit notion and runtime representation of adapters including mapping relationships from an interface to a list of interfaces/components
- provided and required interfaces of components
- runtime access to the component inheritance relationships
- a converter that deterministically finds adapters for a given interface/component pair
- a binder that deterministically binds an interface to a component instance.

The converter and binder are both configurable strategies. They implement the conflict-resolution strategy in the case that there are multiple, equally rated adapters or components matching the requirements. An incarnation of this framework, as for example the presented Smalltalk version, can profit from an infrastructure which already supports some of these mechanisms.

Our experiences in using the dynamic component adaptation framework are very favorable so far. We experienced that in an environment such as Smalltalk where no explicit distinction between objects and components exists, developers tend to make use of the adaptation functionality even for very fine-grained objects. Therefore, it is necessary to have a common understanding of components and component functionality in mind. Design for Slippage serves as guideline here because it restricts the usage of dynamic adaptation to the Context and the edges of the Design Core.

References

[BGR96] Bischofberger W.R., Guttman M., and Riehle D. “Architecture Support for Global Business Objects: Requirements and Solutions”. In *Joint Proceedings of the SIGSOFT’96 Workshops*. ACM/SIGSOFT, New York, 1996.

[Bra94] Brand S. *How buildings learn - What happens after they're built*. Viking, 1994.

[BR96] Bischofberger W.R. and Riehle D. “Global Business Objects - Requirements and Solutions”. In *Proceedings of the Ubilab Conference '96* . Mätzel K.U. and Frei H.-P. (Eds.) *Computer Science Research at Ubilab, Research Projects 1995/96*. Universitätsverlag Konstanz, Konstanz, 1996.

[GYS96] Gosling J., Joy B., and Steele G. *The Java Language Specification*. Addision-Wesley, 1996.

[HO93] Harrison W. and Ossher H. “Subject-Oriented Programming (A Critique of Pure Objects)”. In *OOPSLA'93 Conference Proceedings*. ACM Press, New York, October 1993.

[ISO93a] ISO/ITU. ITU-T X.901 | ISO/IEC 10746-1 ODP Reference Model Part 1.

[ISO93b] ISO/ITU. ITU-TS SG7.Q16. ODP Trader.

[Lau94] Lau C. *Object-Oriented Programming Using SOM and DSOM*. Van Nostrand Reinhold, 1994.

[Mae87] Maes P. “Concepts and Experiments in Computational Reflection”. In *OOPSLA ‘87 Conference Proceedings*. ACM Press, New York, 1987.

[MB96] Mätzel K.-U. and Bischofberger W.R. “Evolution of Object Systems - How to tackle the Slippage Problem on object systems”. In *Proceedings of the Ubilab Conference '96* . Mätzel K.U. and Frei H.-P. (Eds.) *Computer Science Research at Ubilab, Research Projects 1995/96*. Universitätsverlag Konstanz, Konstanz, 1996.

[McA95] McAffer J. “Meta-level programming with CodA”. In *ECOOP ‘95 Conference Proceedings*. Springer Verlag, 1995.

[McA96] McAffer J. “Meta-level architecture support for distributed objects”. In *IWOOOS '95 Conference Proceedings*. USENIX Association, Berkeley, 1995.

[MD95] Microsoft Corporation & Digital Equipment Corporation. *The Component Object Model Specification*, Draft Version 0.9, Oct. 1995.

[OMG96] Object Management Group. CORBA 2.0, Common Object Services and Common Facilities Specification. 1996.

[PA91] Purtilo J. and Atlee J. “Module Reuse By Interface Adaptation”. *Software Practice and Experience*, Vol. 21, No. 1, 1991.

[PN86] Prieto-Diaz R.and Neighbors J. “Module Interconnection Languages”. *Journal of Systems and Software*, Vol. 6, No. 4, 1986.

[Ree96] Reenskaug T., Wold P., and Lehne O.A. *Working with Objects*. Manning, 1996.

[SU96] Senivongse T. and Utting I.A. “A model for evolution of services in distributed systems” In *Proceedings of the IFIP/IEEE Int. Conference on Distributed Platforms*. Schill A., Mittasch C., Spaniol O., and Popien C. (Eds.) *Distributed Platforms*. Chapman & Hall, 1996.

[Weg96] Wegner P. “Interoperability”. *ACM Computing Surveys*, Vol. 28, No. 1, 1996.