Type-Safe Delegation for Run-Time Component Adaptation

Günter Kniesel

Universität Bonn
Institut für Informatik III
Römerstr. 164, D-53117 Bonn, Germany
gk@cs.uni-bonn.de
http://javalab.cs.uni-bonn.de/research/darwin/

Abstract. The aim of component technology is the replacement of large monolithic applications with sets of smaller software components, whose particular functionality and interoperation can be adapted to users' needs. However, the adaptation mechanisms of component software are still limited. Most proposals concentrate on adaptations that can be achieved either at compile time or at link time. Current support for dynamic component adaptation, i.e. unanticipated, incremental modifications of a component system at run-time, is not sufficient.
This paper proposes object-based inheritance (also known as delegation) as a complement to purely forwarding-based object composition. It presents a type-safe integration of delegation into a class-based object model and shows how it overcomes the problems faced by forwarding-based component interaction, how it supports independent extensibility of components and unanticipated, dynamic component adaptation.

1 Introduction

Component-oriented programming aims at the replacement of monolithic applications with sets of smaller software components. Its ... motivation is two-fold. For software engineers, assembly of applications from existing components should increase reuse, thus allowing them to concentrate on value-added tasks and to produce high-quality software within a shorter time. For users, component oriented programming promises tailor-made functionality from the adaptation of ready-made components.

Component adaptation includes the customization of individual components as well as the customization of a whole component-based application by replacing some components with others that are better suited for a specific task.

Known approaches to component adaptation can be classified according to

- their need for preexisting "hooks" in the application as either suitable for *anticipated* or *unanticipated* changes,
- the time when adaptation is performed as either *static*, *load-time* or *run-time* (dynamic)
- their ability to adapt whole component types or individual component instances as either *global* or *selective*. Selective approaches can be further classified as either *replacing* or *preserving* , depending on whether they replace an existing component instance by its adapted version or let both be used simultaneously.

Rachid Guerraoui (Ed.): ECOOP'99, LNCS 1628, pp. 351–366, 1999.

– the applied techniques as either *code-modification-based*, *proxy-based* or *meta-level-based*.

With respect to the above classification this paper presents an approach to unanticipated, dynamic[1], selective, proxy-based, object preserving component adaptation. It leverages proxy-based techniques by introducing typed delegation as a new basic interaction primitive beyond simple message sending.

The need for *unanticipated* dynamic adaptation has been repeatedly pointed out in literature (e.g. in [MS97]) and its practical relevance has been impressively demonstrated by the recent transition from national currencies to the Euro. As opposed to the "year two thousand problem" this change of requirements could not be forseen as the currently deployed software systems had been designed. So the software departments of European banks, insurances and other companies providing 24-hours services to their customers were suddenly faced with the problem to perform unanticipated changes to their systems, without discontinuing operation. Not all of them succeeded, e.g. some banks needed more than three days to get their automatic teller machines operational with the new software. Others succeeded thanks to redundand hardware.

Detailing the ideas sketched in [Kni98b] this paper explores the feasibility of more timely and inexpensive, purely software-based solutions.

When active components can neither be directly modified nor unloaded from a running system, we are faced with the problem to change their behavior solely by adding more components. This paper explores how far delegation can help to solve this apparent contradiction and how component adaptation can be performed without deleting the "old" version of a component. Delegation enables the joint use of different versions of a component and the easy modeling of components that present different interfaces to different clients.

An overview of the state of the art is given in section 2 using the Euro example to illustrate the benefits and limitations of different component adaptation techniques. This section motivates the need for typed delegation as a new component interaction primitive. Section 3 introduces typed delegation in the framework of the DARWIN object model and the LAVA programming language. The use of DARWIN/LAVA for dynamic component adaptation is described in section 4.

2 State of the Art

Currently, the problem of component adaptation is mainly tackled from a programmer's point of view with proposals that aim at easing reuse of existing components in the development of new applications. Most proposals in this category are based on code modification ([Bos98,Har97,ACLN98,Kel97,Hol98]). The paper [KAZ98] proposes programing guidelines for the construction of components that should anticipate and ease the construction of wrappers.

A second line of work concentrates on the problem of adapting running appliations. Proposals in this category are based on wrappers (e.g. [PBJ98]) or metalevel architectures (e.g. [MS97]).

[1] The proposed approach can also be applied statically or at load-time. However, its dynamic use is the most beneficial one with respect to component adaptation

Before proceeding to the introduction of typed delegation it is instructive to review the aforementioned techniques (code modification, adapters, and metaobject protocols) in the light of their suitability for dynamic component adaptation. We do not go into the details of approaches that impose special design guidelines ([KAZ98,MS97]), making them unsuitable for unanticipated adaptation.

Code Modification. Code modification uses two inputs, a class to be modified and a specification of the modification. The result is a modified version of the initial code which is used instead of the old version. Examples in this category are Jan Bosch's superimposition technique ([Bos98]), the class composition proposal of Harrison and Ossher ([Har97]) and the recent work of Keller and Hölzle on binary component adaptation ([Kel97,Hol98]). Superimposition is a language construct within a special object model, LayOM. LayOM programs are translated to C++ and Java, making the modified source code available for further use. Class composition as proposed by Harrison and Ossher is also applied at "compile-time" and requires source-code availability. Binary component adaptation can be seen as a more general form of class composition, which can adapt *binary* components when they are *loaded*. The basic technique behind binary component adaptation can be generalized to other approaches. In particular, it can be used to produce a more dynamic (load-time) version of subject oriented programming ([HO93,HOT98]), with similar component adaptation properties.

Code modification is applicable at compile-time or load-time but has only limited applicability at run-time. Its essence is the replacement of an existing class by a new version of that class. This is very difficult in a running system, where instances of the class to be replaced already exist and are being used. This is the well-known yet still generally unsolved problem of schema evolution in database systems, transferred to the run-time environment of an object-oriented language. Even if the schema evolution problem could be takled in some ad hoc way, dynamic class replacement would be too coarse grained for many applications, globally affecting all existing instances of a component, even if selective adaptation was intended.

Component instance replacement. In the extreme case when each component has only one instance, neither the schema evolution problem, nor the granularity problem would arise. Thus a component and its sole instance could be replaced by a new version. However, component instance replacement has its own shortcomings. First of all, the component to be replaced might not be prepared to hand all its relevant private data over to the new version, if adaptation was not anticipated. Secondly, the application might require joint use of the old and the new component. For instance, for the Euro transition it is required that prices in the national currency be printed in addition to prices in Euro during the first two years.

Wrappers. When active components can neither be directly modified nor unloaded from a running system, we are faced with the problem to change their behavior solely by adding more components. This leads us to the use of wrappers ([GHJV95]). A wrapper is interposed between a component and each of its clients that should perceive some new behaviour. For each operation visible to the client the wrapper either provides an own implementation or forwards corresponding messages to the "wrapped" component. In the sequel a wrapper will also be called a *child* and the "wrapped" component will be called its *parent*.

The wrapper approach enables unanticipated, dynamic, selective adaptation, joint use of different versions of a component and easy modeling of components that present different interfaces to different clients. However, wrappers in traditional class-based object-oriented systems fall short of achieving the desired adaptation functionality. The usefulness of the wrapper approach is limited by the underlying object model, which provides message sending as the only component interaction primitive[2]. Thus forwarding of messages that should be answered by the parent on behalf of the child can only be implemented by sending the message from the child to the parent.

2.1 Adaptation and the *Self* Problem

It has been repeatedly pointed out (e.g. by [Har97,Szy98]) that message sending limits the range of component interaction and component adaptation that can be achieved. This is due to the so called "*self*-problem" [Lie86] that is inherent in message passing. In [Har97] Harrison and Ossher rephrased the *self* problem in component-based terminology:

> Robust solutions [...] require that when components of composite objects invoke operations [...], the operations need to be applied to the composite object, rather than to the component object alone.

With respect to wrappers, this means, that when a message is *sent* from a child to a parent, the value of the *self* pseudovariable is bound to the parent. Thus all subsequent messages to *self* are addressed to the parent. As a consequence, the parent "is not aware" of modified behaviour in the child. If the child provides an own implementation of a method, say print(), the parent will ignore it and continue to use its own print() method, even in the context of forwarded messages.

Consider the wrapper-based modeling of the Euro transition scenario illustrated in listing 1.

The class DM (for the German currency) represents a component that existed before the decision to adopt the Euro was taken. It provides two methods:

- `amount()` computes the curent value of an investment (deposit, stock, assurance), using some private, non-shared data structures
- `foo()` does something else but calls `self.amount()` in its implementation.

The class EuroDMWrapper represents the wrapper. It encapsulates a reference to a DM instance and also provides two methods:

- `amount()` calls amount() on its wrapped DM instance and divides the result by the fixed DM-to-Euro exchange rate.
- `foo()` calls foo() on its wrapped DM instance

[2] For the purpose of this discussion events do not add any new insight and are therefore not mentioned.

```
public class DM{
       ... some private data ...
       ... constructor ...
    int amount() { return ...
    }
    void foo() { ... self.amount() ...
    }
}

public class EuroDMWrapper{
       // the wrapped component:
    DM parent;
       // constructor:
    EuroDMWrapper(DM p) { parent = p
    }
       // redefined method:
    int amount() { return parent.amount() / 1.96
    }
       // forwarding method:
    void foo() { parent.foo()
    }
}
```

Listing 1: The Euro scenario: Traditional wrapper-based adaptation fails

When a foo() message is sent to the wrapper, the same message is sent to its parent and in the course of its evaluation, the `self.amount()` message will be sent to the parent. Thus the adapted definition of `amount()` from the child will be ignored, and a wrong result will be produced in the end.

What is required in such cases is the ability to make one component act as a specialization of another one by sharing its *self*. This is exactly what is provided by delegation or object-based inheritance.

3 Delegation

”Delegation” was originally introduced by Lieberman ([Lie86]) in the framework of a class-free (prototype-based) object model. An object, called the *child*, may have modifiable references to other objects, called its *parents*. Messages for which the message receiver has no matching method are *automatically* forwarded to its parents. When a suitable method is found in a parent object (the *method holder*) it is executed after binding its implicit *self* [3] parameter. This parameter refers to the object on whose behalf the method is executed. Automatic forwarding with binding of *self* to the message receiver is called *delegation* (figure 1). Automatic forwarding with binding of *self* to the method holder is called *consultation*.

In contrast to code modification, delegation does not require source code or abstract binaries – it works equally well on native code and can be employed at run-time rather than at compile- or load time. Operating by addition rather than replacement and at the level of objects rather than classes delegation does not incur the schema-evolution problems of run-time code modification.

[3] The implicit self parameter is also called *this* (in Simula, Java and C++) and *current* (in Eiffel).

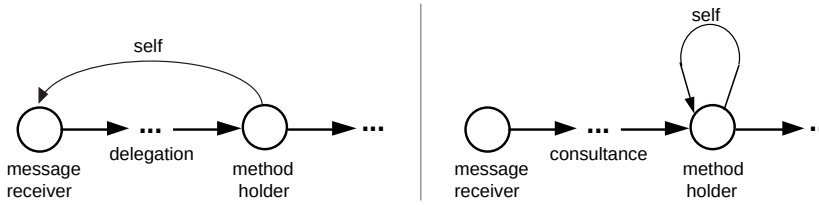


Fig. 1. Different effect of delegation and consultation on *self*

Why simulations of delegation are not enough. In spite of its advantages, no widely-used class-based object oriented language has incorporated delegation yet. Instead, various simulations of delegation have been proposed, either as language specific idioms or general ”design patterns”.The possible simulation techniques and their drawbacks are summarised and evaluated in [Har97] and [Kni98a]. The main disadvantages of the discussed simulations are:

- the need to anticipate the use of a piece of software as part of a larger composite and to build in ”hooks” that allow the correct treatment of self in the context of the composite. Components that do not provide such hooks are not effectively composable ([Szy98]).
- the need to obey rigid coding conventions for implementing the required hooks. On one hand, the absence of a standard convention goes against any attempt to make composable software by simulating delegation. Components made according to different conventions cannot be deployed together. On the other hand, the risk of standardising immature proposals has been demonstrated by JavaSoft’s ”Object Aggregation and Delegation Model”, which was initially contained in the Glasgow Proposal for the new JavaBeans model and has been dropped as result of public criticism of its limitations[4].
- the need to edit (or at least recompile, if a tool for automatic generation of forwarding methods is available) the ”delegating” classes when the interface of the class ”delegated to” changes, in order to propagate the change, e.g. the addition of a method. This introduces another variant of the ”syntactic fragile base class problem” ([MS97,Szy98]).

Each of the individual simulation techniques has additional weaknesses in terms of limited applicability, limited functionality, limited reusability or excessive costs [Kni98a]:

- *Storing a reference to self in parent objects* has a very limited applicability. Sharing of one parent by multiple delegating children cannot be expressed at all and recursive delegation can only be simulated with significant run-time and software mainatenance costs.

[4] Note that the limitations of the Glasgow Proposal’s rejected part have already been hardwired into the design of COM ([Box98])

– *Passing a reference to self as an argument* of forwarded messages requires to extend the interface of methods in parent objects, which might not be possible, if the parent object is part of a ready-made, black-box component.
Furthermore the typing of the explicit *self* argument interacts in subtle ways with the construction of subclasses of parent classes. In the end, the simulation either does not reach the full functionality of delegation or it does so at the price of excessive costs for managing class hierarchy changes, rendering reuse *ad absurdum*. ([Kni98a]).

Why delegation has been (said to be) difficult. Considering the above list of limitations inherent to simulation approaches it is obviously worthwile to rethink the reasons why delegation has been considered unsuitable or unfeasible in the context of mainstream object-oriented languages. A survey of literature would reveal that many authors have acknowledged the modeling power and elegance of dynamic delegation but at the same time called for ways to harness this power and to make it more amenable to a "disciplined use" ([Don92,Szy98,Tai93,Wec97]). This is essentially a critique on the lack of a static type system for delegation-based languages. However, even some well respected authorities ([Aba96]) claimed that delegation cannot be combined with static typing and subtyping without severe restrictions of the way objects are used ([Fis95]). Therefore, the use of simulations seemed to be the only choice.

It is the main achievement of the DARWIN model ([Kni99]) to have shown that type-safe dynamic delegation with subtyping *is* possible and *can* be integrated into a class-based environment, laying the foundations for dynamic component adaptation. The DARWIN model is sketched in the next section to the degree relevant in the context of this paper, pointing out the interesting parallel between type-safety and independent extensibility of components ([Szy96]). The use of DARWIN for dynamic component adaptation is described in section 3.

4 Darwin and Lava: Combining Class-Based and Object-Based Inheritance

We assume that the reader is familiar with the notions of class, instance, and class-based inheritance ([Weg90]). For simplicity of presentation, we shall introduce DARWIN () using the syntax of LAVA ([Cos98,Sch97]), a proof of concept extension of Java that conforms to the DARWIN model [5] .

Parents and Declared Parents. In DARWIN / LAVA, objects may delegate to other objects referenced by their *delegation attributes*. Delegation attributes have to be declared in an object's class by adding the keyword `mandatory delegatee` to an instance variable declaration. Delegation attributes are *mandatory*, i.e. they must always have a non-nil value. This is automatically enforced by the compiler and suitable run-time checks. If class *C* declares a delegation attribute of type *T* we say that *C* is a *declared child class* of *T* (and of *T*'s subtypes), and *T* is a *declared parent type* of *C* (and of *C*'s subclasses).

[5] The only deviation of Lava from Darwin is the restriction to single inheritance following the design of Java. This is not a real limitation because Lava offers multiple delegation.

Dynamic Delegation. Since a delegation attribute can reference any value that conforms to its declared type, assignment to a delegation attribute can be used to change the behavior of an object at run-time by changing its parent object(s). This is called *dynamic* delegation. If desired, we can restrict delegation to be static by adding the Java keyword `final` to the delegation attribute's declaration.

Mandatory and Optional Atrributes. An attribute is called *mandatory* if it must always have a non-nil value, *optional* otherwise. This is specified by corresponding keywords that can be added to any attribute declaration (if nothing is specified *optional* is the default). The *mandatory* keyword is most relevant in connection with delegation attributes because it influences the typing relation between a child class and its parent type(s).

Types. In purely inheritance-based models, the type of an instance corresponds to the signature of the methods defined by its class (and its superclasses). Delegation has the effect of extending this interface by the interfaces defined for the *declared* types of *mandatory* delegation attributes. Thus, in LAVA delegation and inheritance are two orthogonal ways to create subtypes of a base type. Delegating objects may be used in any place where an object of their declared parent type is expected.

An Example. In LAVA, a class of text formatting objects (`Formatting`) that may use and dynamically switch between different line breaking strategies (type `LineBreaking`) can be written as shown in Listing 2

```
public class Formatting {
       // delegate line breaking requests to the object referred to by lb;
   mandatory delegatee LineBreaking lb;
       // create object with default strategy
   public Formatting () { lb = new SimpleLineBreaking();
   }
       // switch strategy
   public setLBStrategy (LineBreaking _lb) { lb = _lb;
   }
       // By how many pixels can individual text components be stretched
       // Overrides method from parent type LineBreaking.
   public int[] getStretchability() { ...
   }
}
```

Listing 2: The strategy pattern in Lava

The `Formatting` class may use all methods of the `LineBreaking` type as if they where locally defined or inherited from a superclass - with the essential difference that it may dynamically switch to a different set of method implementations simply by assigning an object of a different `LineBreaking` subtype to the variable `lb`. Like in the case of inheritance, the `Formatting` class can fine tune the "inherited" behavior via overriding. For instance, in the above example it was assumed that the `LineBreaking` type specifies that the line breaking algorithm calls the method `getStretchability()` to determine by how many pixels individual text elements can be stretched. Then providing a specialized version of this method (as shown above) might be all that is needed to adapt the "inherited" behavior to the delegator's needs. Note especially that the designer and the implementors of the `LineBreaking` type do not have to be aware of its use as a parent class and hard-code any hooks to enable this use.

4.1 Type Safety, Independent Extensibility and Overriding

Let us consider the scenario illustrated in figure 2 (Gray arrows represent delegation at instance and class level, arrows with hollow heads instantiation resp. inheritance, and solid arrows "normal" object references.) : `c`, an instance of class `Child`, delegates to `p`, an instance of class `Parent`; `Parent` is a subtype of the declared parent class of `Child`.

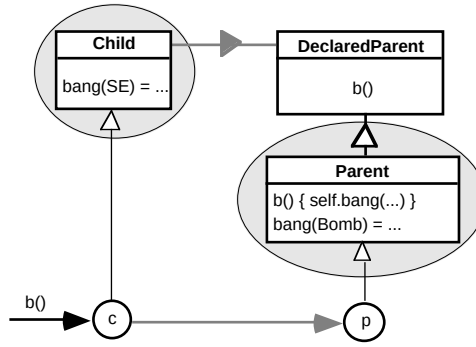


Fig. 2. What happens during evaluation of the message `c.b()`?

Typing Problem. In figure 2 the two `bang` methods have different argument types: `Child` expects an argument of type `StockExchange` whereas `Parent` provides an argument of type `Bomb`. Therefore, Fisher ([Fis95]) argues that during the evaluation of the message `c.b()` delegated from `c` to `p` the message `self.bang(aBomb)` sent back to `c` would be unsafe, because its argument would not have the expected type. However, arguing about type-safety in the above example is misleading, because the essence of the problem is not typing.

Independent Extensibility Problem. The astute reader might have noted that the classes `Child` and `Parent` might have been developed and compiled independently, knowing only `DeclaredParent` but not each other. Therefore, even if the two independently introduced `bang` methods had the same signature it would still be very unlikely that they have the same semantics. Overriding of `Parent::bang` by `Child::bang` would therefore be undesirable anyway, because it would silently change the semantics relied upon by methods from `Parent`, leading to obscure, hard to locate errors.

The importance of independent extensibility for component programming has already been described by Szyperski ([Szy96,Szy98]) in a delegation-free environment. Whereas Szyperski focused on the joint use of two independent extensions of a base type by a third party, our discussion relates to the delegation-based composition of one independent extension with another one.

Overriding. The point of the above discussion of type-safety and independent extensibility is that both seemingly unrelated problems have a common cause: the implicit

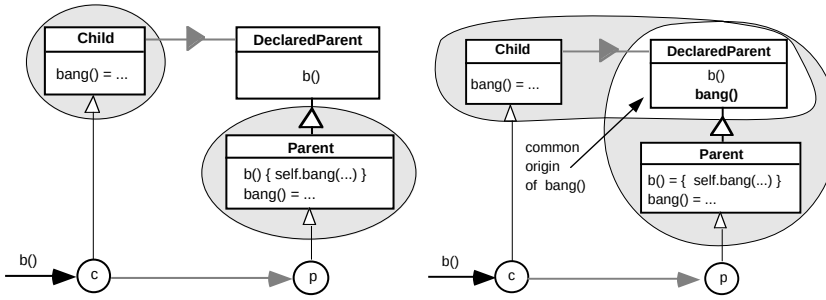
assumption that methods with the same name (and possible same signature) may override each other.

This assumption can be safely made only if it is guaranteed that the author of the overriding method is aware of the effect. This is always the case with class-based inheritance: a method implementation in a class can only override one in a known superclass. Assuming a sensible documentation of the superclass and no intentional fraud, authors of subclasses will not create semantically incompatible overriding methods. The assumption is unsafe if independent extensibility is possible.

The solution to both problems discussed above is the following adapted rule for method overriding:

For a message *recv.n(args)* a method with signature σ from type *T overrides* the matching [6] method from the static type of *recv*, T_{stat}, if there is some common *declared* supertype of *T* and T_{stat} that contains σ.

Thus a method from type *T* will only override methods from a declared parent type of *T*. This rule reflects the fact that the common declared supertype (`DeclaredParent` in figure 2 and figure 3) is the common semantic base on which implementors of independent extensions can rely.



(a) Independently introduced methods: *No* overriding.

(b) Methods with common origin: Overriding enabled.

Fig. 3. Overriding

The above definition of overriding influences the operational semantics of the system. A method of an object is applicable to a message only if it may override the matching method from the static type of the receiver expression. Messages with no applicable local method are delegated further to a parent object.

In the case of static delegation the type-safety of the approach follows from the fact that the sender of the message to *self* is itself among the parent objects.

For instance, in figure 3a, the message `self.bang()` sent from `p` to `c` will not find an applicable method in `c` and be delegated further up the object hierarchy, back

[6] A method with signature n(T1, ..., Tn):T matches a message recv.n(expr1, ..., exprn) if for all i = 1...n, the static type of expri is a subtype of Ti.

to p (where the search will succeed). In figure 3b the message will find an applicable method in c.

The type-safe treatment of dynamic delegation is more involved and its discussion is beyond the scope of this paper. Static delegation already sufices for most wrapper-based component adaptation tasks. A complete presentation of the DARWIN model including type-safe dynamic delegation is contained in [Kni99].

To recap, the integration of delegation into statically typed object-oriented languages offers, among others, an easy way

- to make an object appear to be part of and act on behalf of various other ones, and
- to extend existing objects in unanticipated ways, without fear of semantic conflicts.

As a general object-oriented language mechanism delegation has a multitude of possible uses. For instance, many well-known behavioural patterns ([GHJV95]) turn out to be simple applications of delegation (e.g. chain of responsibility, proxy, decorator, strategy, state, visitor). In the sequel we shall concentrate on the use of delegation for dynamic component adaptation.

5 DCA: Dynamic Component Adaptation

Dynamic component adaptation is a modification of a component's functionality at runtime that can be achieved by

- adding further components to a system and
- transferring part of the existing component's "wiring" to the new components.

The technical prerequisites for this functionality are language support for delegation and support for component "re-wiring" by the underlying component architecture. This section discusses both aspects. For simplicity components are considered to be JavaBeans. The described considerations equally apply to any other component model (e.g. COM) provided that in a future version it will support type-safe delegation according to DARWIN.

5.1 Incremental Component Assembly and Adaptation with Delegation

Delegation enables extension and modification (overriding) of a parent component's behavior. Child components can be transparently used in any place where parent components are expected. Unlike other approaches, which irrecoverably destroy the old version of a component, delegation enables two types of component modifications. Additive modifications are the product of a series of modifications, each applied to the result of a previous one. Disjunctive modifications are applied independently to the same original component.

Additive modifications are enabled by the recursive nature of delegation. They meet the requirement that the result of compositions / adaptations should itself be composable / adaptable ([Bos98,Szy98]). In the user view (figure 4a) additive composition is depicted by stacking components one on top of the other. The system view in figure 4b

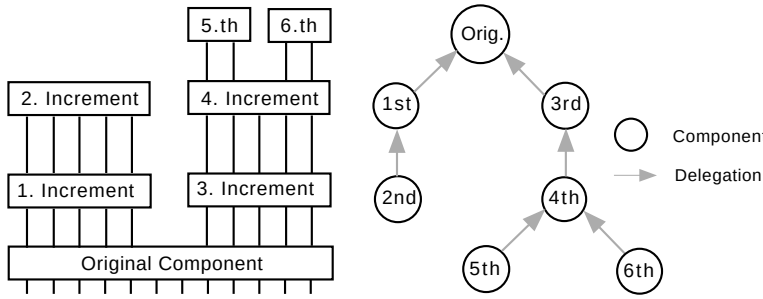


Fig. 4. Additive and disjunctive composition

shows the implementation by building chains of delegating components. E.g. the first and second increment of the original component together form an additive modification.

Disjunctive extensions are enabled by the fact that each extension is encapsulated in a separate component instance that can be addressed and reused independently. Disjunctive extensions are most useful in modeling components that need to present themselves differently to different clients. In the system view (figure 4a), disjunctive extension are visualized sitting on top of the jointly extended component. For instance component 5 and 6 represent different extensions of component 4, which itself is part of a disjunctive extension branch of the original component. At the implementation level (figure 4b), disjunctive extensions delegate to the same parent component.

5.2 Dynamic Component Assembly

The effects described so far only take effect if the most specialized increment components along each disjunctive modification branch are used as the receiver of messages or events instead of the original component. Therefore, dynamic component adaptation requires dynamic component (re)assembly, i.e.

- rerouting of all ”input connections” of a component to its increment (or to different disjunctive increments) and
- routing of the ”output connections” of all increments to the same destinations as the corresponding outputs of the original component.

In the JavaBeans model input connections correspond to the registration of a component as an event listener of other components and output connections correspond to the set of own registered event listeners.

The complete schema for dynamic component adaptation, including component reassembly (”re-wiring”), is illustrated in figure 5. Part a) shows the implementation view of the original component configuration. Part b) shows the re-wired configuration after addition of three increment components, one of which represents a disjunctive modification.

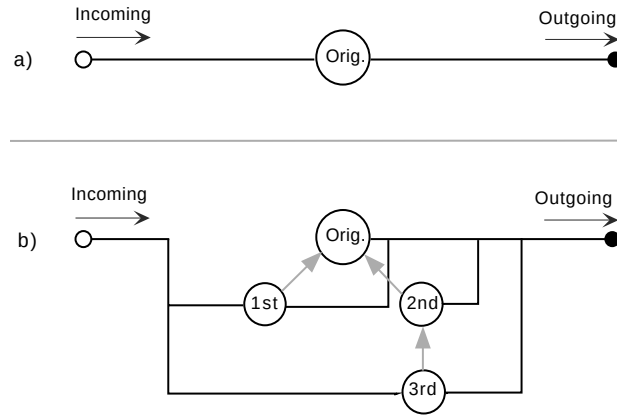


Fig. 5. Component (re)wiring: before extension (a) and after extension (b)

A component architecture that supports dynamic component (re)assembly must provide

- a run-time component directory,
- the ability to ask every component for its current input and output connections,
- the ability to ask every component to abandon all its input connections in favor of one or more other components. This must happen as an atomic operation in order to guarantee that the system is not left in an inconsistent state.

These requirements are not met by current component architectures, although dynamic component rewiring is an essential infrastructure which would benefit also simple forwarding based dynamic composition techniques, not just delegation. E.g. the ”Extensible Runtime Containment and Services Protocol” of the Glasgow model for JavaBeans

- provides no run-time component directory; every BeanContext functions as a directory of nested components but there is no directory of top-level BeanContexts,
- a JavaBean is only required to maintain a list of registered Listener objects but no list of objects to which it listens itself; thus it only knows about its outgoing connections but not about incoming connections,
- there is no atomic operation for handing a bean’s incoming connections over to another bean.

We are currently exploring suitable extensions of the JavaBeans model. Similar dynamic rewiring infrastructures have also been proposed and already implemented by other researchers, e.g. [PBJ98].

5.3 Example: Delegation-Based Euro Transition

Comming back to the example from Listing 1 this section shows how a delegation-based modelling of the Euro scenario would look like. Listing 3 shows the delegation-based variant of the EuroDMWrapper class. Note that an explicit definition of the `foo` method is not necessary. Messages for `foo` are implicitly delegated to `parent`. The `<-` operator in the amount method denotes explicit delegation to the object reffered to by `parent`. This is analogous to a `super` call in class-based inheritance.

```
public class EuroDMWrapper{
       // delegate to the object referred to by parent:
    mandatory delegatee DM parent
       // constructor:
     EuroDMWrapper(DM p) {  parent = p
     }
       // redefined method:
     int amount() { return parent<-amount() / 1.96
     }
}
```

Listing 3: The Euro scenario: Delegation-based adaptation

Now a `foo` message to the wrapper will produce correct results because the `self.amount()` message from DM (cf Listing 1) will correctly be addressed to the wrapper object thus using the adapted definition of `amount()`.

6 Conclusions

The crucial role of dynamic, object-based inheritance (delegation) as a basis of component interaction and the need for an integration of delegation into the statically typed class-based model has been repeatedly pointed out by many researchers. In this context the contributions of this paper are two-fold:

- In the first place, it introduced a general model for dynamic, type-safe delegation, DARWIN, and an implemented language, LAVA, that provide the required language support for component-oriented programming.
- Secondly, the problem of *dynamic* component adaptation was discussed and a solution based on delegation and dynamic component reassembly was presented.

There are, nevertheless, still many open questions. For instance, a high-performance implementation of LAVA is required, to help the current proposal make its way into mainstream commercial languages like Java or C++, thus providing broad language support for delegation-based component interaction. Also, components to be composed by delegation depend on the “specialization interface” of their parent components. Therefore, advanced component interface specifications and approaches to the “semantic fragile base class problem” are required ([Lam93,Ste96,MS97,Szy98,Wec97]).

References

Aba96. Abadi, Martin and Cardelli, Luca. *A Theory of Objects*. Springer, 1996.

ACLN98. P.S.C. Alencar, D. D. Cowan, C.J.P. Lucena, and L.C.M. Nova. A model for gluing components. In Wolgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proc. of Third Intern. Workshop on Component Oriented Programming (WCOP 98)*, volume 10, pages 101–108. Turku Center for Computer Science, 1998.

Bos98. Bosch, Jan. Superimposition - A Component Adaptation Technique, 1998.

Box98. Don Box. *Essential COM*. Addison Wesley, 1998.

Cos98. Costanza, Pascal. *Lava: Delegation in a Strongly Typed Programming Language – Language Design and Compiler (In German: Lava:Delegation in einer streng typisierten Programmiersprache – Sprachdesign und Compiler)*. Masters thesis, University of Bonn, 1998.

Don92. Dony, Christophe and Malenfant, Jacques and Cointe, Pierre. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, 27(10):201–217, 1992.

Fis95. Fisher, Kathleen and Mitchell, John C. A Delegation-based Object Calculus with Subtyping. In *Proceedings of 10th International Conference on Fundamentals of Computation Theory (FCT '95)*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer, 1995.

GHJV95. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.

Har97. Harrison, William and Ossher, Harold and Tarr, Peter. Using Delegation for Software and Subject Composition. Research Report RC 20946 (922722), IBM Research Division, T.J. Watson Research Center, Aug 1997.

HO93. William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, 28(10):411–428, 1993.

Hol98. Holzle, Urs and Keller, Ralf. Binary Component Adaptation. In *Proceedings of ECOOP 98*, pages 307–329. Springer Verlag, 1998.

HOT98. William Harrison, Harold Ossher, and Peter Tarr. Load-time subject-oriented programming, June 1998 1998.

KAZ98. Bülent Kücük, M. Nedim Alpdemir, and Richard N. Zobel. Customizable adapters for black-box components. In Wolgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proc. of Third Intern. Workshop on Component Oriented Programming (WCOP 98)*, volume 10, pages 53–60. Turku Center for Computer Science, 1998.

Kel97. Keller, Ralph and Hölzle, Urs. Supporting the Integration and Evolution of Components Through Binary Component Adaptation. Technical Report TRCS97-15, University of California at Santa Barbara, September 1997.

Kni98a. Günter Kniesel. Delegation for java api or language extension? Technical Report IAI-TR-98-4, ISSN 0944-8535, University of Bonn, May 1998 1998.

Kni98b. Günter Kniesel. Type-safe delegation for dynamic component adaptation. In Wolgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proc. of Third Intern. Workshop on Component Oriented Programming (WCOP 98)*, volume 10, pages 9–18. Turku Center for Computer Science, 1998.

Kni99. Günter Kniesel. *Darwin - A Unified Model of Sharing for Object-Oriented Programming*. Ph.d. thesis (forthcoming), University of Bonn, 1999. Knie98.

Lam93. Lamping, John. Typing the Specialization Interface. *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, 28(10):201–214, 1993.

Lie86. Lieberman, Henry. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, 21(11):214–223, 1986.

MS97. Kai-Uwe Mätzel and Peter Schnorf. Dynamic component adaptation. Technical report 97-6-1, Union Bank of Swizerland, June 1997 1997.

PBJ98. F. Plasil, D. Balek, and R. Janecek. Sofa/dcup:architecture for component trading and dynamic updating. In *ICCDS 98*, Annapolis, Maryland, USA, 1998. IEEE CS Press.

Sch97. Schickel, Matthias. *Lava: Design and Implementation of Delegation Mechanisms in the Java Runtime Environment (In German: Lava:Konzeptionierung und Implementierung von Delegationsmechanismen in der Java Laufzeitumgebung).* Masters thesis, University of Bonn, 1997.

Ste96. Steyaert, Patrick and Lucas, Carine and Mens, Kim and D'Hondt, Theo. Reuse Contracts: Managing the Evolution of Reusable Assets. *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 268–285, 1996.

Szy96. Szyperski, Clemens. Independently Extensible Systems Software Engineering Potential and Challenges . In *Proceedings of the 19th Australian Computer Science Conference, Melbourne, Australia*. Computer Science Association, 1996.

Szy98. Szyperski, Clemens. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

Tai93. Taivalsaari, Antero. Object-Oriented Programming with Modes. *Journal of Object-Oriented Programming (JOOP)*, 6(3):25–32, 1993.

Wec97. Weck, Wolfgang. Inheritance Using Contracts & Object Composition. In Weck, Wolfgang and Bosch, Jan and Szyperski, Clemens, editor, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP '97)*, pages 105–112. Turku Center for Computer Science, Turku, Finland, 1997. ISSN 1239-1905.

Weg90. Peter Wegner. Concepts and paradigms of object-oriented programming (expansion of oopsla 89 keynote talk). *ACM OOPS Messenger*, 1(1):7–87, 1990.