

Type-Safe Delegation for Run-Time Component Adaptation

ECOOP 99 conference paper (Springer, LNCS series)

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 twofold. 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 *dynamic* (run-time)

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

With respect to the above classification this paper presents an approach to unanticipated, dynamic¹, selective, wrapper-based, object preserving component adaptation. It advances wrapper-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 European currencies to the Euro. As opposed to the “year two thousand problem” this change of requirements could not have been foreseen at the time that the currently deployed software systems were 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 redundant 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. This section also illustrates the limitations of pure message sending, which prevent adaptations of the type required by the Euro transition example. It thus motivates the need for a more powerful component interaction primitive. Section 3 points out that delegation could be a perfect solution if it could be integrated into statically typed object models. Section 4 introduces typed delegation in the framework of the DARWIN object model and the LAVA programming language, an extension of Java. The use of DARWIN/LAVA for dynamic component adaptation is described in section 5.

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

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

development of new applications. Most proposals in this category are based on code modification ([Bos98,HOT97,ACLN98,KH97,HK98]). The paper [KAZ98] proposes programming guidelines for the construction of components, which anticipate and ease the construction of wrappers.

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

In the following we review these techniques in the light of their suitability for dynamic component adaptation.

Metalevel Architectures. Metalevel architectures enable extensive manipulation of a system at run-time. In principle, every object-level approach can be mimicked at the metaobject-level. The price for this generality is that metalevel architectures still suffer from two main limitations: they defeat static type systems and are notoriously inefficient.

One application of metalevel architectures to the problem of dynamic component adaptation is described in [MS97]. It supports only anticipated adaptation because it relies on special design guidelines, which have to be obeyed in order to construct adaptable systems.

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 ([HOT97]) and the recent work of Keller and Hölzle on binary component adaptation ([KH97,HK98]). 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 tackled 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 primitive².

2.1 Adaptation and the *self* problem

It has been repeatedly pointed out (e.g. by [HOT97,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 [HOT97] 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 forwarded from a child to a parent via message sending, the value of the *self*³ pseudovvariable is bound to the parent. Thus all messages to *self* in the context of the forwarded message are addressed to the parent. So, even 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. As a consequence, the parent “is not aware” of modified behaviour in the child.

This effect is illustrated by the traditional wrapper-based modeling of the Euro transition scenario shown 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:

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

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

- `amount()` computes the current 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 `DMtoEURO` 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.

```
public class DM{
    // ... private data, constructor ...

    int amount() { return ... }

    void foo() { ... self.amount() ... }
}

public class DMtoEURO{
    // the wrapped component:
    DM parent;

    // constructor:
    DMtoEURO(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 `DMtoEuro` wrapper, the same message is sent to its `DM` 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 (in `DM` instead of `Euro`) will be produced in the end.

What is required in such cases is the ability to forward messages from a child component to a parent component in a way that guarantees that the child and the parent will have a common *self* during the evaluation of the forwarded message. In the above example this would allow the `self.amount()` message issued by the parent to be evaluated in the context of the child, thus accessing the intended version of the `amount()` method. 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* 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). In contrast, automatic forwarding with binding of *self* to the method holder is called *consultation* ([KRC91]). Delegation is object-based inheritance whereas consultation is just an automatic form of message sending. To avoid confusion, it is worthwhile to note that many authors use the term delegation to denote consultation or simple message sending.

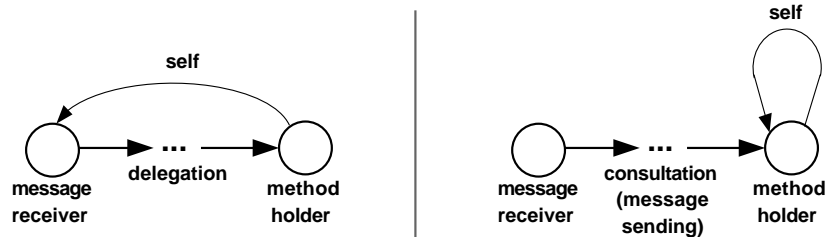


Fig. 1. Different effect of delegation and message sending on *self*

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.

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 ([Cop92]) or general "design patterns" ([GHJV95]). The possible simulation techniques and their drawbacks are summarised and evaluated in [HOT97] 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 limitations⁴.

- 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:

- *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 maintenance costs.
- *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 ([Kni98a]). 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*.

Why delegation has been (said to be) difficult. Considering the above list of limitations inherent to simulation approaches it is obviously worthwhile 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 ([AC96]) claimed that delegation cannot be combined with static typing and subtyping without severe restrictions of the way objects are used ([FM95]). 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 5.

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

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 model⁵.

Parents and Declared Parents. In DARWIN / LAVA, objects may delegate to other objects referenced by their *delegation attributes*. Delegation attributes are declared in an object's class by adding the keyword `delegatee` to an instance variable declaration. 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).

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 Attributes. An attribute is called *mandatory* if it must always have a non-nil value, *optional* otherwise. This distinction can be specified in LAVA by corresponding keywords, which may be added to any attribute declaration. If nothing is specified *optional* is the default. If an attribute is declared as being mandatory, this invariant is automatically enforced by the compiler and by suitable run-time checks. Mandatory attributes are most relevant in connection with delegation because they influences the typing relation between a child class and its parent type(s). A mandatory delegation attribute ensures that a parent object of the declared type always exists (cf example in Listing 2).

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 Figure 2.

The `Formatting` class may use all methods of the `LineBreaking` type as if they were 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

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.

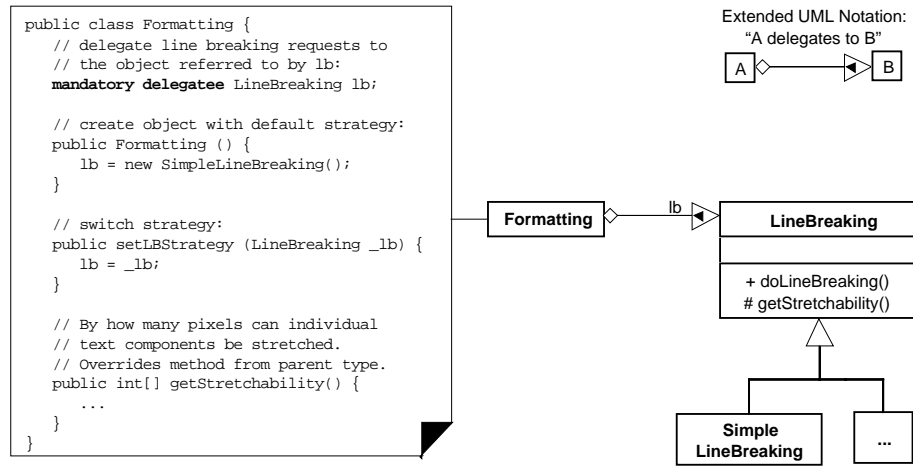


Fig. 2. The strategy pattern in LAVA

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. It is instructive, if not even revealing, to compare the Figure 2 with the recommendations for the implementation of the original strategy pattern example presented in [GHJV95].

4.1 Type Safety, Independent Extensibility and Overriding

So far, we have seen how delegation works, how it overcomes the *self* problem and how it eases the construction and maintenance of dynamic systems.

One main promise and problem of dynamic systems is dynamic extension. The promise of dynamic extensibility is that components with required behaviour may be composed at run-time from smaller components with well-defined partial functionality. For instance, application components could dynamically switch parts of their behaviour by selecting suitable strategy components as their parents. The problem of such scenarios is to guarantee that two concrete components that have been independently developed as extensions of a common base component (e.g. of the abstract strategy type) will work together without undesired side-effects. In this section we shall focus on the problem of independent extensibility in conjunction with delegation and on its notable similarity to the typing problem that has often been cited as an argument against delegation.

Let us consider the scenario illustrated in Figure 3⁶: 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`.

⁶ The notation is standard UML extended with a symbol for delegation, depicted as an aggregation relation with an inheritance-style arrow at the opposite end. The outer (white) arrowhead

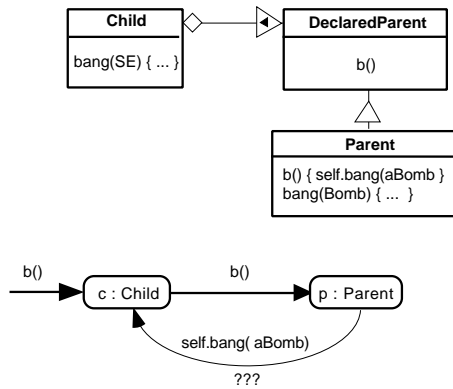


Fig. 3. What should happen during evaluation of the message `c.b()`?

Typing Problem. In Figure 3 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 ([FM95]) 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 the `bang` method from `Parent` by the `bang` method from `Child` 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 which other method it will override. This is always the case with class-based inheritance: a method implementation in a class can only override

indicates the direction of forwarding, the inner (black) arrowhead indicates the binding of *self* to the child object.

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.

Solution. The solution to both problems discussed above is to extend method overriding by a criterion for semantic compatibility. We propose the following adapted rule for method overriding:

Definition 1. A method defined in type T_1 may only override a method defined in type T_2 if the method is contained in some common declared supertype of T_1 and T_2 .

Thus in LAVA a method from a class will only override methods that had already been defined in a superclass or a *declared* parent type. This rule reflects the fact that the common declared supertype (e.g. DeclaredParent in Figure 3 and Figure 4) is the common semantic base on which implementors of independent extensions can rely.

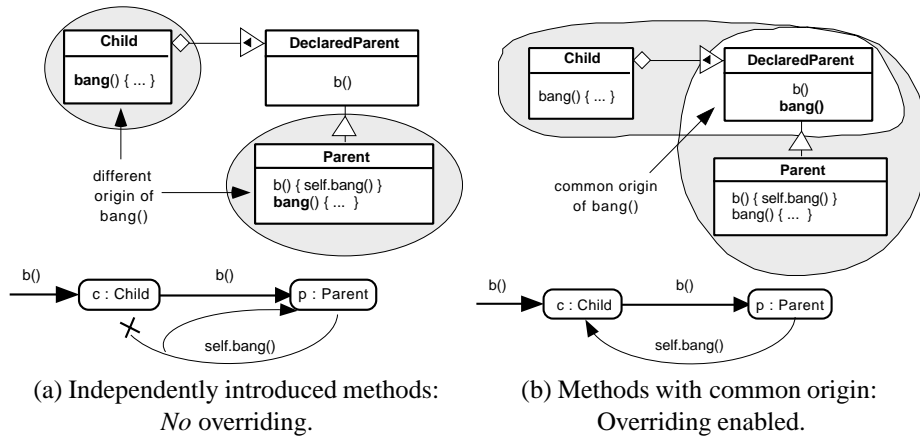


Fig. 4. Overriding

The above definition of overriding is used as a criterion for method applicability thus influencing the operational semantics of the system.

Definition 2. A method of an object is applicable to a message if and only if it may override the matching⁷ method from the static type of the receiver expression. Messages with no applicable local method are delegated further to the next parent object.

For instance, in Figure 4a, the message `self.bang()` sent from `p` to `c` will not find an applicable method in `c` and will therefore be delegated further up the object hierarchy, back to `p` (where the search will succeed). In Figure 4b the message will find an applicable method in `c`.

⁷ A method with signature $m(T_1, \dots, T_n) : T_0$ matches a message $recv.m(expr_1, \dots, expr_n)$ if for all $i = 1 \dots n$ the static type of $expr_i$ is a subtype of T_i .

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. The type-safe treatment of *dynamic* delegation is more involved and its discussion is beyond the scope of this paper. Static delegation already suffices 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 patterns ([GHJV95]) turn out to be simple applications of delegation (e.g. chain of responsibility, wrapper, 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 run-time 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 5a) additive composition is depicted by stacking components one on top of the other. The system view in Figure 5b

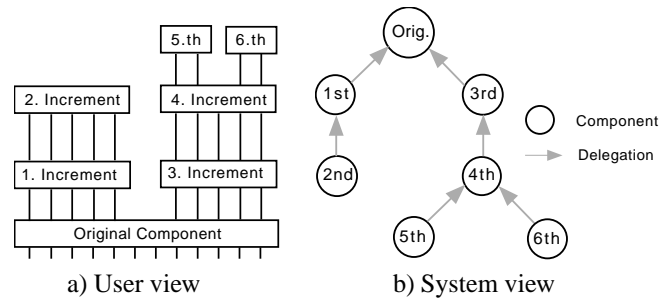


Fig. 5. Additive and disjunctive composition

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 user view (Figure 5a), 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 5b), disjunctive extensions delegate to the same parent component.

5.2 Dynamic Component Re-Wiring

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 instead of the original component. Therefore, dynamic component adaptation requires dynamic component “re-wiring”, i.e. rerouting of all “input connections” of a component to its most specialized increments. In the JavaBeans model input connections correspond to the registration of a component as an event listener of other components.

The complete schema for dynamic component adaptation, including component re-wiring, is illustrated in Figure 6. 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.

A component architecture that supports dynamic re-wiring must provide

- a run-time component directory, including information about the “increment of” relationship between components and
- 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 re-wiring is an essential infrastructure which would benefit also sim-

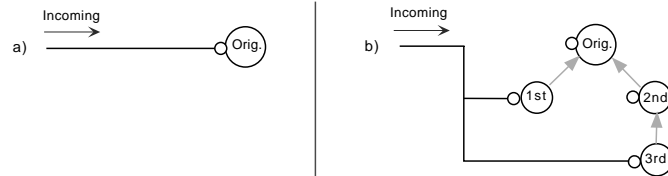


Fig. 6. Component wiring before extension (a) and after extension (b)

ple forwarding based dynamic composition techniques, not just delegation. For instance, 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.

Dynamic re-wiring infrastructures have already been proposed and implemented by other researchers, e.g. [PBJ98]. However, it is still unclear how they could be applied to arbitrary JavaBeans. We are currently exploring suitable extensions of the JavaBeans model.

5.3 Example: Delegation-Based Euro Transition

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

```
public class DMtoEURO{
    // delegate to the object referred to by parent:
    mandatory delegatee DM parent ;

    // constructor:
    DMtoEURO(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 a `DMtoEURO` instance will produce correct results because the subsequent `self.amount()` message sent by its DM parent (cf Listing 1) will correctly be addressed to the `DMtoEURO` wrapper object thus using the adapted definition of `amount()`.

Figure 7 shows two different clients that use the old DM component and its adapted `DMtoEuro` version simultaneously. The stock overview component calculates in Euro, complying with the new regulations, whereas the internal accounting component continues to use DM as long as possible because calculations in DM are more precise.

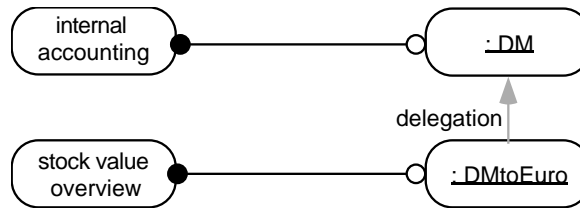


Fig. 7. Simultaneous use of the original DM component and of its adapted `DMtoEuro` version by different clients

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 twofold:

- 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,SLMD96,MS97,Szy98,Wec97]).

7 Acknowledgements

This paper benefited from many discussions with Pascal Costanza and Oliver Stiemering and from many helpful comments of the anonymous ECOOP reviewers.

References

- [AC96] Martin Abadi and Luca Cardelli. *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 Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proc. of Third Intern. Workshop on Component Oriented Programming (WCOP 98)*, pages 101–108. Turku Center for Computer Science, 1998. ISBN 952-12-0284-X.
- [Bos98] Jan Bosch. Superimposition - A Component Adaptation Technique, 1998. <http://www.ide.hk-r.se/bosch/articles.html>.
- [Box98] Don Box. *Essential COM*. Addison Wesley, 1998.
- [Cop92] James O. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1992.
- [Cos98] Pascal Costanza. *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.
- [FM95] Kathleen Fisher and John C. Mitchell. 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.
- [HK98] Urs Hölzle and Ralf Keller. Binary Component Adaptation. In *Proceedings of ECOOP 98*, pages 307–329. Springer Verlag, 1998.
- [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.
- [HOT97] William Harrison, Harold Ossher, and Perri Tarr. Using Delegation for Software and Subject Composition. Research Report RC 20946 (922722), IBM Research Division, T.J. Watson Research Center, Aug 1997.
- [HOT98] William Harrison, Harold Ossher, and Peter Tarr. Load-time subject-oriented programming (personal communication), June 1998.
- [KAZ98] Bülent Küçük, M. Nedim Alpdemir, and Richard N. Zobel. Customizable Adapters for Black-Box Components. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proc. of Third Intern. Workshop on Component Oriented Programming (WCOP 98)*, pages 53–60. Turku Center for Computer Science, 1998. ISBN 952-12-0284-X.
- [KH97] Ralph Keller and Urs Hölzle. 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.
- [Kni98b] Günter Kniesel. Type-safe Delegation for Dynamic Component Adaptation. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proc. of Third Intern. Workshop on Component Oriented Programming (WCOP 98)*, pages 9–18. Turku Center for Computer Science, 1998. ISBN 952-12-0284-X.

- [Kni99] Günter Kniesel. *Darwin – Dynamic Object-Based Inheritance with Subtyping*. Ph.d. thesis (forthcoming), University of Bonn, 1999.
- [KRC91] Günter Kniesel, Mechthild Rohen, and A.B. Cremers. A Management System for Distributed Knowledge Base Applications. In W. Brauer and D. Hernandez, editors, *Verteilte Künstliche Intelligenz und Kooperatives Arbeiten (Distributed Artificial Intelligence and Cooperative Work)*, pages 65–76. Springer-Verlag, 1991.
- [Lam93] John Lamping. Typing the Specialization Interface. *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, 28(10):201–214, 1993.
- [Lie86] Henry Lieberman. 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 Switzerland, 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] Matthias Schickel. *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.
- [SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. *Proceedings OOPSLA '96, ACM SIGPLAN Notices*, pages 268–285, 1996.
- [Szy96] Clemens Szyperski. Independently Extensible Systems Software Engineering Potential and Challenges. In *Proceedings of the 19th Australian Computer Science Conference, Melbourne, Australia*. Computer Science Association, 1996.
- [Szy98] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
- [Tai93] Antero Taivalsaari. Object-Oriented Programming with Modes. *Journal of Object-Oriented Programming (JOOP)*, 6(3):25–32, 1993.
- [Wec97] Wolfgang Weck. Inheritance Using Contracts & Object Composition. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *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.