Delegation for Java: API or Language Extension?

- Technical Report IAI-TR-98-5, May 1998 -

(Extension of a letter to JavaSoft commenting on its
"Proposal for an Object Aggregation / Delegation API"
May 26th, 1997)

Günter Kniesel
Universität Bonn
Institut für Informatik III
Römerstr. 164
D-53117 Bonn

fax: +49-228-73-4382
phone: +49-228-73-4511
e-mail: gk@cs.uni-bonn.de
http://javalab.cs.uni-bonn.de/research/darwin

Abstract

The failed attempt of JavaSoft to incorporate an "Object Aggregation / Delegation API" into its newest JavaBeans model has demonstrated impressively the high necessity and also the notorious difficulty of incorporating delegation into typed class-based languages. Although JavaSoft's proposal has been withdrawn due to public criticism of its limitations, the general issue is still relevant: is it possible to define a "one size fits all" standard API for delegation, and if not, is there any real alternative to such an API?

This paper explores both questions. It shows that on one hand, all API-level solutions have serious drawbacks related to functional limitations, simulation costs and sensitiveness to change. On the other hand, recent work has demonstrated that integration of dynamic delegation into a class-based, statically typed language with subtyping is feasible in theory and practice in spite of contradictory claims in literature. However, since an efficient implementation is still missing we can just recommend an API-level compromise and invite researchers worldwide to join efforts for a high-performance implementation of dynamic delegation.

Keywords

design patterns, dynamic delegation

1 Introduction

Traditionally, object-oriented programming is centered around the notions of classes, instantiation, inheritance and encapsulation which date back to Simula 67 ([4]) and Smalltalk 80 ([8]). Twenty years after the creation of Simula, Liberman's language DELEGATION ([20]) marked the beginning of a new paradigm of object-oriented programming, *prototype-based programming*. Prototype-based languages focus on working with concrete objects instead of abstract classes. They give up the notion of class and replace static, class-based inheritance by dynamic, object-based inheritance, also known as *delegation*.

Although the advantages of delegation-based programming over class-based programming in terms of flexibility and unanticipated reuse were widely recognized from the beginning, work on the relationship of class- and delegation-based systems has mainly been centered around the question whether particular features of one paradigm can or cannot be simulated in the other one. Especially regarding the *simulation* of dynamic delegation in traditional class-based systems, the last ten years have seen a proliferation of various language specific idioms ([2]) and general design patterns [7], [12], [10]). Although these proposals shed light on some interesting technical aspects, they often neglected the cost of simulations in terms of program clarity, good design, reusability, error-safety and time spent by programmers on simulations rather then on the essence of an application. Also some of the proposed simulations are not faithful, they only approximate the full functionality of delegation.

The limitations of simulation approaches suggest that the *integration* of delegation into typed, class-based object models might be a more promising way to increase the flexibility and expressive power of object-oriented languages. However, a series of recent theoretical papers claim that a type-safe combination of delegation with subtyping, let alone typed *dynamic* delegation, is impossible ([1], [5], [6]))

This paper explores both alternatives. On one hand, it reviews, compares and extends existing simulation techniques, trying to achieve a simulation that is as faithful as possible. On the other hand, it sketches the DARWIN model ([17]), which achieves the “impossible” type-safe integration of dynamic delegation into a class-based model with unrestricted subtyping. Finally, it compares the extended model with the discussed simulations and gives some recommendations for current “best practice” and future research.

2 Delegation

The language concept called “delegation” was originally introduced by Lieberman ([20]) 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*. In this paper delegation parents are also called *delegatees* and delegation children are called *delegators*, to avoid confusion that might arise from the multiply overloaded meaning of *child* and *parent*. 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, which refers to the object on whose behalf the method is executed. Automatic forwarding with binding of `self` to the initial message receiver is called *delegation* (figure 1). Automatic forwarding with binding of `self` to the *method holder* is called *consultation* ([19]).The keyword `this` always refers to the method holder.

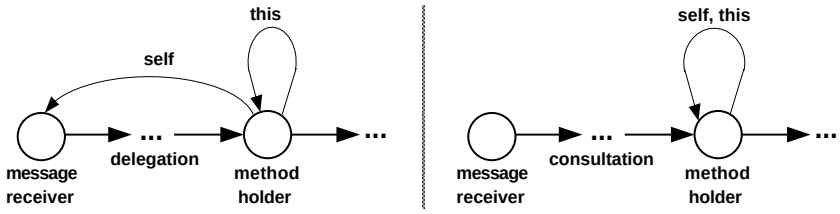


FIGURE 1 Different effect of delegation and consultation on `self`

Intuitively, delegating a message means asking another object to do something *on behalf of the message receiver*, i.e. as the message receiver would do it, whereas consultation means asking another object to do something as *it* knows how. Technically, delegation is a variant of inheritance whereas consultation is just an automatic form of message sending.

Delegation is a language concept that should be well distinguished from an implementation technique which is also called delegation by many authors: writing a method that only sends one message to an object referenced by an instance variable – this message typically has the same signature as the method that contains it. To avoid confusion, we call this *explicit resending*. This is the technique underlying all API-level simulations of delegation.

3 API-level Simulations

In this section we list the requirements for a faithful simulation of delegation, discuss two alternative classes of simulation approaches and evaluate them with regard to the stated requirements.

3.1 Requirements for a Simulation

A faithful simulation of delegation in a strongly-typed, class-based language must meet the following technical (1-3) and usability (4-5) requirements:

(1) The information about `self`, the initial receiver of a delegated message, must be propagated to delegatee objects.

(2) All messages otherwise sent to `this` in delegatees have to be redirected to `self`, in order to give delegator objects a chance to override methods of delegatee objects.

(3) Static type safety must be guaranteed.

(4) The simulation must not require modifications to delegator classes and their subclasses when methods are added to delegatee classes.

(5) The simulation must enable unanticipated reuse. In particular, it must not depend on the assumption of a certain structure of the class- or object-level delegation hierarchy, e.g.

- that a delegatee object will be shared only by a fixed number of delegators,
- that only certain classes will ever be used as delegatees, or
- that delegation will not be recursive.

Depending on the technique for meeting the first requirement, simulation approaches can broadly be classified into two categories:

- storing of references to `self` in the delegatee and
- passing of `self` as an additional argument of delegated messages.

In [10] these two categories are called the *stored pointer* and the *passed pointer* model. In the remainder of this chapter the essence of these techniques will be summarized and additional aspects beyond those treated in [10] will be discussed. Such additional aspects are the functional limitation of the stored pointer model with respect to multiple delegators and recursive delegation, as well as the limitation of both models with respect to static typing and evolution of the class hierarchy. The latter aspect is discussed in detail on the example of the passed pointer model.

3.2 Storing references to delegators

In the Glasgow proposal[1], the method for simulating the treatment of `self` that characterizes delegation was to *explicitly store a reference to the delegator* in the delegatee. This works well only in the most restricted application scenario

- when there is just one delegator per delegatee (delegatees are not shared),
- *and* the delegator is the initial message receiver (no recursive delegation).

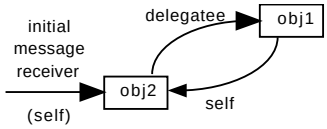


Fig. 1 One single delegator: Only scenario in which storing of `self` reference works well

[1] For the Glasgow Specification see [14]. Other approaches in the stored pointer category can be found e.g. in [10, 11, 12].

With **multiple delegators** (fig. 2) the delegatee would not know which delegator is `self`, if corresponding information were not passed dynamically as a message argument. But if explicit passing as an argument is required anyway, storing delegators makes no sense at all. Without dynamic information about the current `self` one can only hardwire a fixed interaction protocol in the code of the parent object by predetermining which messages are sent to *obj2* and which to *obj3*. But that is not delegation

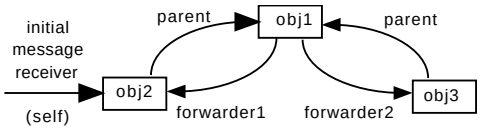


FIGURE 2 Multiple delegators / shared delegatee: Which delegator sent me this message? Which one is `self`?

Recursive delegation can be modeled by storing in each delegatee a direct reference to `self` (fig. 3). Then delegatees cannot be used on their own as message receivers: When messages were *sent* (not forwarded) *to delegatees*, the stored `self` reference would point to the wrong object (fig. 3b).

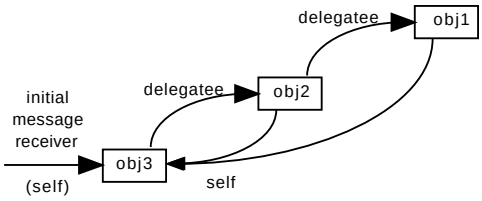
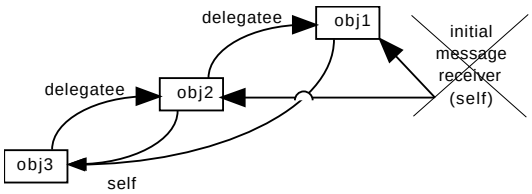


FIGURE 3a) Recursive delegation implemented by storing a direct reference to `self` in every delegatee



b) The drawback: messages sent to delegatees are addressed to the wrong `self`

One could avoid this effect by including a *setDelegator()* method in the public interface of delegatees. However, this would complicate the clients of delegatee objects, requiring to duplicate the number of messages sent: if *d* is the delegatee object then, instead of *d.msg()*,

- "normal" clients of *d* have to send the messages *d.setDelegator(d)*; *d.msg()*;
- "delegating" clients of *d* that are not themselves delegatees have to send the messages *d.setDelegator(*`this`*)*; *d.msg()*;
- assuming that *mySimulatedSelf* is the name of the instance variable referring to their simulated `self`, "delegating" clients of *d* that are themselves delegatees have to send the messages *d.setDelegator(mySimulatedSelf)*; *d.msg()*;

just to ensure that (simulated) `self` is set correctly. This "solution" is clumsy, inefficient, error-prone and incompatible with multi-threaded execution: during execution of any delegatee method, concurrent calls to *setDelegator()* have to be disabled, thus disabling *any* concurrent execution of delegatee methods (which all have to be preceded by a call to *setDelegator()*).

Summarizing, storing references to delegators in delegatees has a very limited applicability. Sharing of one delegatee by multiple delegators cannot be expressed at all and recursive delegation can only be modeled with significant run-time costs and an amount of simulation code that makes the approach highly error-prone and sensitive to change.

3.3 Passing of self as a message argument

When delegatees do not store delegator references, their methods[1] have to be extended by an additional, explicit "`sSelf`" parameter with the convention that

- "normal message sending" passes the receiver object as the `sSelf` argument, i.e. every message `object.msg(arg)` is replaced by `object.msg(object,arg)`,
- messages to `self` / `this` are redirected to `sSelf`, i.e. every message `msg(arg)` or `this.msg(arg)` is replaced by `sSelf.msg(sSelf,arg)`), and
- simulation of delegation passes the current value of `sSelf` further up the delegation hierarchy, i.e. delegation is simulated by the forwarding message `delegatee.msg(sSelf,arg)`.

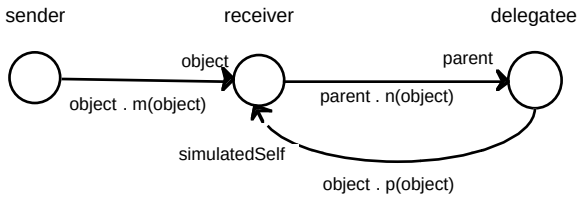


FIGURE 4 Simulation of normal message sending and delegation by passing `self` as message argument

The interaction between a message sender, a message receiver (`self`), and a delegatee in this approach is illustrated in figure 4. The message `object.msg(object)` results in the "delegated" message `parent.n(object)` and in the subsequent message `object.p(object)` sent back to the actual value of `sSelf`. The *actual* value of the `sSelf` argument is shown with each message. The names of formal arguments are shown as role names.

This approach works for shared delegatees as well as for recursive delegation. Moreover, due to the first rule of the above message sending convention, instances of delegatee classes can be used on their own as message receivers, not just as delegatees. Of the technical requirements stated above (3.1) the only one that is still to be met is

---

1 Only the methods for messages that can be delegated to the object. In an API-level simulation these are only the public methods. In a language extension also protected methods could be included, making delegator classes equivalent to subclasses with respect to their inheriting and overriding capabilities.

type-safety. The remainder of this section explores the wide-ranging implications of typing for the simulation and its usability.

3.3.1 The type of self

If the declared type of `sSelf` were one fixed delegator type (as suggested in the Glasgow proposal), then delegation from any other delegator type would be excluded by the type checker. resulting again in a very restricted model (non-shared delegatees, non-recursive, non-extensible). Moreover, delegatees which expect `sSelf` to be of a *delegator* type could not be used on their own, because the application of the above "normal message sending" rule would be vetoed by the type-checker (because the delegatee would not substitutable for a `sSelf` parameter of delegator type).

A better alternative is to declare `sSelf` to be of an interface type[1] that corresponds to the interface of the delegatee class[2] containing the method:

```
interface DelegateeInterface {
    public aMethod(DelegateeInterface sSelf, ...);
}

class Delegatee implements DelegateeInterface {
    public aMethod(DelegateeInterface sSelf, ...) {...}
}
```

Delegator classes implement at least the `DelegateeInterface`. Therefore any instance of a delegator class can be passed in the `sSelf` argument, making delegatee classes independent of their potential delegators and hence *reusable in non-anticipated contexts*.

In Java the substitutability of delegators for delegatees has to be made explicit by declaring delegator interfaces as extensions of delegatee interfaces:

```
interface DelegatorInterface extends DelegateeInterface {
... };
```

The basic approach to typing `sSelf` using Java interfaces (or purely abstract classes in other languages) is illustrated in figure 5. `I1` is the `DelegateeInterface`, I2 is the `DelegatorInterface`, `C1` is a delegatee class, and `C2` is a delegator class.

---

1 In C++ Java interfaces translate to purely abstract classes, *interface extension* is derivation of a purely abstract subclass and *interface implementation* is derivation of a concrete subclass.

2 It might appear as a drawback of this solution that delegatees cannot send messages to *sSelf* that are specific to a particular delegator type. If really needed, one can work around this restriction with run-time type checks. However, one should use this technique sparingly, since it again makes the delegatee dependent on certain delegator types.

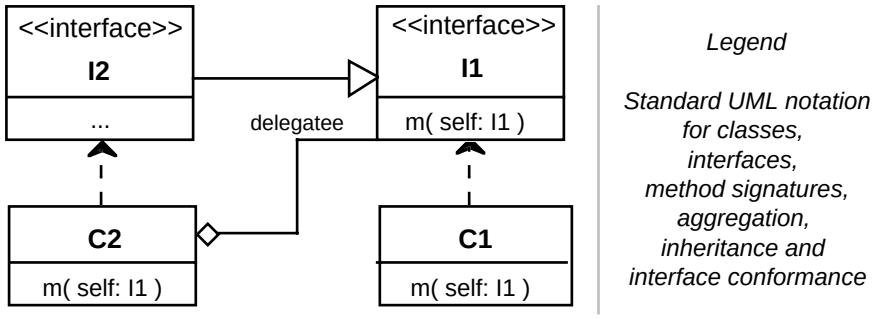


FIGURE 5 Typing of the `self` parameter (first approximation)

A delegator class must contain one method for every signature defined in the delegatee interface. This method will either implement local behavior or forward the received message to the delegatee object (*forwarding method*). If class *C2* does *not* provide local behavior for `aMethod(...)` it will appear as follows:

```
class C2 implements I2 {
    // "parent" is the object to which messages are forwarded:
    I1 parent;

    // A forwarding method:
    public aMethod(I1 sSelf, ...) { parent.aMethod(sSelf,...);
    }
}
```

3.3.2 Method overriding in delegator classes and their subclasses

Whereas message forwarding can be trivially implemented as shown above, method overriding is more complicated, due to strong typing. A local method in class `Delegator` will more often than not need to call other methods that are specific to `Delegator`, respectively `DelegatorInterface`. However, the type-checker will veto any such attempt on the premise that the declared type of `sSelf` is `DelegateeInterface`. Therefore, any local implementation will need a dynamic type check (resp. a checked type cast) in the first place to verify whether the current value of `sSelf` is an object of type `DelegatorInterface`. For instance, a delegator class `C2` that provides own behavior for method `m()` from the delegatee interface `I1` will appear as follows:

```
class C2 implements I2 {
    I1 parent;

    public m(I1 sSelf) {
      I2 selfI2 = (I2) sSelf;   // this cast always succeeds
      ...                       // local implementation
      selfI2.n(...);            // ... that uses selfI2

}
```

Methods that perform a type cast in order to adjust the type of `sSelf` are called *casting methods* and are depicted in diagrams by a down-arrow (↓) next to the method signature. It is worth noting that *in a delegator class* the type cast will always succeed:

(1) If the local method was activated by a message to `sSelf` from a delegatee then the receiver object *is* `sSelf`. The cast will succeed because *an object can always be cast to its own type*. The cast merely recovers type information that was lost while forwarding `sSelf` up the delegation hierarchy.

(2) If the local method was activated by a forwarding message from a delegator, then the cast will also succeed, since by the construction of the simulation (figure 5) *every delegator has a type which is a subtype of all its (immediate and recursive) declared delegatee types*.

Similarly, the type cast will always succeed in casting methods of *subtypes* of declared delegator types (provided that the delegator is not simultaneously a delegatee): consideration (1) still holds for delegator subtype instances (the scenario of consideration (2) does not apply).

3.3.3 Method overriding in subtypes of declared delegatee types

The situation is subtly different if we consider method overriding in nontrivial *subtypes of declared delegatee types*. Then the scenario of consideration (2) is applicable but the consideration does not hold:the dynamic type of `sSelf` is no subtype of the type of the current delegatee object because the interface of the delegatee object contains messages that are not in the dynamic type of `sSelf`. Thus the cast of `sSelf` to the delegatee type will fail.

Consider for instance the scenario illustrated in figure 6. The delegatee class `C1` has a subclass (`C11`) whose type (`I11`) is a subtype of the declared delegatee type (`I1`). An instance of the delegator class `C2` delegates to an instance of `C11`. Obviously, the type of the child object (`I2`) is not a subtype of the parent object's type (`I11`) because the method `n()` is not contained in `I2`.

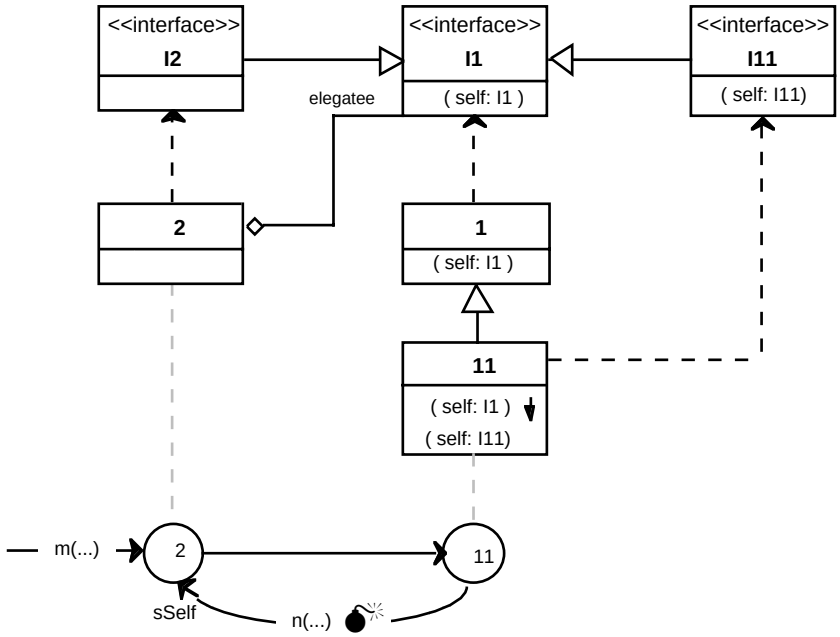


FIGURE 6 Overriding in subclasses of delegatee classes / subtypes of delegatee types

Thus a casting method `m()` in `C11` will always produce a run-time error due to the inadmissible cast at the beginning of the method:

```
class C11 implements I11 {
    public m(I1 sSelf) {
      I11 selfI11 = (I11) sSelf; // This cast will fail!!!
      ...                        // local implementation
   }
}
```

Obviously, messages specific to `C11` (resp. `I11`) can only be sent to `this`, not to `sSelf`:

```
class C11 implements I11 {
    public m(I1 sSelf) {
      sSelf.m(...);      // message from I1 is sent to sSelf
      ...
      this.n(...);       // message from I11-I1 is sent to this
   }
}
```

In general, messages defined in the static type of `this` but not in the static type of `sSelf` (e.g. in `I11-I1`) can only be sent to `this`. The distinction between these two message receivers has to be *hard-coded* into overriding methods in subclasses of delegatee classes.

Class hierarchy evolution. If programs were static, unchangeable entities, this situation would be acceptable. However, programs typically evolve. A type like I11, which is a subtype of a declared delegatee type might become itself a declared delegatee type when new types and classes are added to the program (cf. figure 7). We have to be prepared for such changes in the structure of the delegation hierarchy if our simulation is not to break in the next release of the program. In general, all modifications of other

parts of a program that do not affect the interface of a given class should not require changes in the implementation of that class ([22]).

In the light of these aims, the possible addition of new delegator classes poses serious problems to the simulation. Changes of the program may lead to situations where the same delegatee object may have delegators of different, unrelated types. Then the set of messages that can be sent to `sSelf` varies depending on the type of delegator object, which is known only at run-time. Thus, when we write an overriding method within a subclass like `C11`, we cannot statically determine which messages may be sent to `sSelf` and which may only be sent to `this`.

For instance figure 7, shows the program from figure 6 extended by a delegator type (`I3`) and a delegator class (`C3`) whose declared delegatee type is `I11`. Now the object `c11` can also have `c3` as delegator and in the context of messages delegated from `c3` it would have to send the message `n()` back to `sSelf=c3`, which contains a more specialized method definition than `c11`.

It is therefore necessary to find a way of determining dynamically the *most specific common supertype (MSCS)* of the dynamic type of `this` and of `sSelf`: all messages defined in the MSCS are safe for `sSelf`, whereas all other messages in the type of `this` are safe only for `this`. E.g. in the case of `sSelf=c2` and `this=c11` the MSCS is `I1` whereas for `sSelf=c3` and `this=c11` it is `I11`.

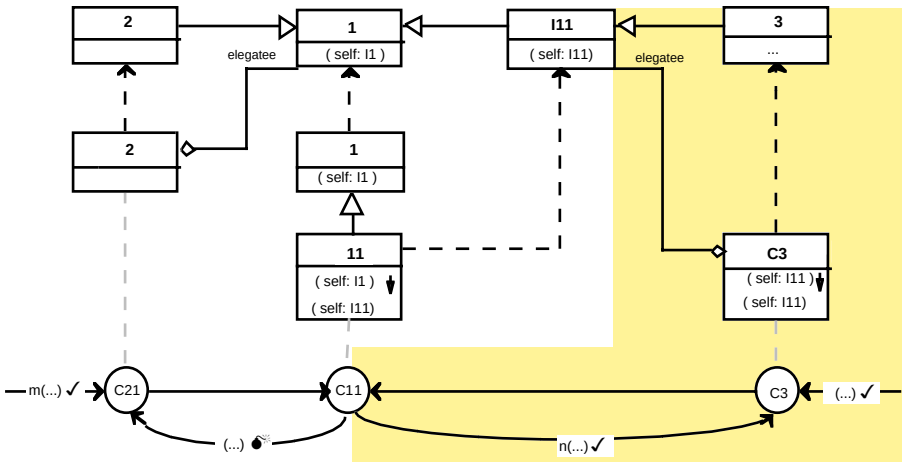


FIGURE 7 Example of program structure evolution that should not require changes in existing classes

Java offers two mechanisms to determine the MSCS of `this` and `sSelf`:

- dynamic type checks and
- method overloading that preserves dynamic binding.

Dynamic type checking. Let the class in which to include the overriding method be the *N*-th one in its inheritance hierarchy (counting the root class as number 1). Then dynamic checking would amount to an *N* times nested “if-then-else”, iterating from the current class up the inheritance hierarchy until a check succeeds. In the case of success a

local method implementation that uses a specific distribution of messages to `sSelf` and to `this` would be called:

```
class C1n implements I1n {
    public m(I11 sSelf, args) {
        if ( sSelf instanceOf I1n ) {
             implementation sending all messages to sSelf, none
             to this
             } else {
        if ( sSelf instanceOf I1n-1) {
             implementation sending messages from I1n-1 to
             sSelf, others to this
             } else { ...
        if ( sSelf instanceOf I11) {
             implementation sending messages from I11 to sSelf,
             others to this
        } ... }
    }
}
```

Overloading[1]. In the overloading approach each of the above dynamic checks for a certain type would be replaced by a method whose `sSelf` argument has the corresponding type. Then dynamic binding of messages forwarded from delegators automatically selects the method with the most specific type ([9], §15.11.2).

```
class C1n implements I1n {
    public m(I11 self, args) {
         // implementation sending messages from I11 to self, others to
         this
    }

    public m(I12 self, args) {
         // implementation sending messages from I12 to self, others to
         this
    }
    ...
    public m(I1n self, args) {
         // implementation sending messages from I1n to self, none to this
    }
}
```

What's gained? The dynamic type checking approach involves worst case run-time costs that are linear in the depth of the inheritance hierarchy. In contrast, method overloading has zero run-time cost. However, neither approach can be recommended, since the main expense of both approaches is hidden in the number of additional overloaded definitions of each single method. Their number is linear in each class, quadratic along an inheritance path and exponential within an inheritance hierarchy:

- The number of overloaded versions of each locally overridden method in a class at depth *N* in the inheritance hierarchy is *N*-1 (the class contains *N*

---

[1] This part of the simulation would work well in Java but not in C++ or other languages that statically bind overloaded methods. In such languages only the dynamic checking approach could be used.

method definitions of which one has the original signature and the others are overloaded versions).

- The worst case sum of overloaded definitions of each method along an inheritance path of depth *N* is $\sum_{i=1..N} i - 1$.
- The worst case sum of overloaded definitions of each method within an inheritance hierarchy of depth *N* with *b* subclasses per class is $\sum_{i=1..N} (i-1) * b^i$.

The most serious drawback is not the sheer number of methods in itself but the fact that each overloaded version is textually identical to the other *N-1* local versions, up to the message receivers. The programmer needs to manually "patch" *N-1* copies of a method in order to adjust which messages are sent to `sSelf` and which are sent to `this`, depending on the type of `sSelf`. This implies the need to manually propagate each change of a method to *N-1* local copies, rendering reuse *ad absurdum*.

This problem is common to the overloading *and* the dynamic casting approach: each method body in the overloading approach corresponds to an identical block of code in one of the success branches of the dynamic checking code. Furthermore, the base problem of typing `sSelf` and its implication on the simulation is not typical to the passed pointer model, it appears also in the stored pointer model. This variation was not worked out in detail simply due to the limited applicability of the stored pointer model.

We may conclude that the search for a faithful simulation of delegation has reached a dead end. Aiming for an approach that fosters reuse in that it does not require changes of existing classes when new classes are added, we have obtained a "solution" that is highly sensitive to changes *within* a class because each change must be consistently performed in *N* copies of a block of code. So we have traded undesirable inter-class dependencies for equally undesirable intra-class dependencies plus excessive simulation costs.

4 Evaluation and Comparison of Simulations

In the previous chapter we have reviewed the two basic approaches for simulating delegation in statically typed class-based languages, known as the stored pointer model and the passed pointer model. Both models rely on aggregation and explicit resending of messages within forwarding methods. Of the requirements listed in section 3.1 they achieve the technical ones (1-3) at the cost of neglecting most usability aspects (4-5).

The main common disadvantages of the discussed simulations are:

- the need to *anticipate* the use of a class as a delegatee and to build in "hooks" that allow the correct treatment of `self` for resent messages. Classes that do not provide such hooks cannot be used as delegatees.
- the need to anticipate which messages in a class can be overridden in delegator classes (i.e. which ones are sent to `sSelf`) and which ones can only be

redefined in subclasses (i.e. which ones are sent to `this`). We have shown that no acceptable solution exists for this problem in current typed class-based languages. Both possible "solutions", using either dynamic type checks or method overloading, lead to the same explosion of code size and maintenance costs.

- the need to edit (or at least recompile) a "delegating" class when the interface of one of its superclasses or delegatee classes changes (e.g. addition of a method). This introduces two variants of the "*syntactic fragile base class problem*":

  – *Fragile superclass*: If a superclass is extended by a method that is forwarded in the subclass, the forwarding method must be deleted and the subclass recompiled, otherwise the new inherited behavior would not take effect in the subclass.

  – *Fragile delegatee class*: Changes to the delegatee's interface require adding/deleting forwarding methods in delegator classes in order to propagate the change.

  In both cases, existing compiled code of subclasses / delegator classes is invalidated.

- the tedious and error-prone process of writing forwarding methods, casting methods, explicit interface definitions and explicit subtyping relations among interfaces. This problem could partly be alleviated by "intelligent" development environments – nevertheless, a language design with well-defined semantics is preferable to dependence on the availability and "intelligence" of a particular tool.

Each of the individual simulation techniques has additional weaknesses:

- Storing a reference to `self` in delegates has a very limited applicability. Sharing of one delegate by multiple delegators cannot be expressed at all. Recursive delegation can only be simulated with significant added run-time costs and an amount of simulation code that makes the approach highly error-prone and sensitive to change.

- Passing a reference to `self` as an argument of forwarded messages is generally applicable but requires to extend the interface of methods in pre-existing classes / types by one extra argument.

The absence of a standard convention how to simulate delegation is another main problem because components made according to different conventions cannot be deployed together. The risk of standardizing immature conventions was demonstrated by JavaSoft's proposal for an"Object Aggregation and Delegation Model" (initially

contained in the Glasgow Proposal, [14]), which was dropped as result of public criticism of its limitations.

In the light of the above evaluation only the generally applicable passed pointer model appears to be an acceptable candidate for a standard simulation. However, due to their extensive common problems all simulations are acceptable only as a compromise as long as no language-level implementation of delegation is available.

5 Lava: Java with Delegation

In a language-level implementation the cost of using delegation reduces to adding the keyword `delegatee` to a variable declaration in the delegator class. In listing 1 our running example is rewritten in LAVA, an extension of Java with dynamic delegation:

```
class Delegator {

    // The keyword delegatee tells the system that all
    // locally unimplemented messages from the interface of
    // Delegatee are delegated to the object referenced by
    // "parent":
    delegatee Delegatee parent;


    // Overriding of a method from "parent". No change of
    // signature (i.e. no explicit sSelf parameter) is required:
    public aMethod(args) { ... }
}
```

LISTING 1 In Lava the use of delegation boils down to the addition of one keyword to a variable declaration. Nothing else is required from the programmer (e.g. no explicit forwarding and type casting methods, no interface declarations, no anticipation of the use of a class as a delegatee, no rewriting of existing classes in order to become delegatees, etc.)

Implementing delegation as a first class language concept makes it trivially simple to use because all aspects that had to be manually simulated before are now dealt with by the language. This includes the automatic generation of forwarding methods, the correct handling of `self` and typing.

The main aspects of the "impossible" combination of delegation with subtyping are informally described in [18] (and partly in [15]). The language design can be found in [3]. Early ideas for a C++-style implementation are presented in [15], the implementation of an extended Java run-time system in [21]. A detailed description of object model, typing, language design, implementation and use is forthcoming in [17].

6 Simulation or Language Extension?

The LAVA design sketched above avoids all of the problems of API-level simulations:

- it is generally applicable (static / dynamic delegation, simple / recursive delegation, shared / non-shared delegatee, optional / mandatory delegatee, transient / persistent delegators, sequential / concurrent execution),
- it avoids the need to anticipate which classes can be used as delegatees, or to rewrite existing classes in order to turn them into potential delegatees,

- it avoids the need to manually implement delegation in every pair of delegator and delegatee classes by writing forwarding and casting methods and replacing all messages to `self` by messages to an explicit `sSelf` receiver object,
- it avoids the need to manually write interface definitions in order to enable substitutability of delegators for delegatees,
- it avoids "syntactically fragile superclasses and delegatee classes",
- it avoids accidental overriding of methods with identic signature but unrelated semantics when independently developed classes become part of the same delegation hierarchy ([18], [16], [15]).

The language extension offers a well-defined semantics and minimizes the workload of the programmer while enabling maximal reuse and extensibility. Therefore we propose a design along the lines of LAVA as the method of choice for introducing delegation into Java and any other typed class-based language.

However, the efficiency of the prototypical implementation described in [3, 15, 21] is still unacceptable for a commercial production programming language. Additional efforts are to be undertaken towards a high-performance implementation that integrates modern compiler technology (e.g. [13]). We invite researchers worldwide to contribute to these efforts. Until delegation becomes part of wide-spread production programming languages the passed pointer simulation appears to be the only viable compromise.

7 Conclusions

The opportunity for increasing the flexibility and modelling power of class based languages by simulating dynamic object-based inheritance (delegation) was recognised long ago in the Smalltalk community. Whereas corresponding simulation techniques are simple and meanwhile standard in dynamically typed object-oriented languages like Smalltalk and Objective C, statically typed languages like Java and C++ lack easily usable and commonly accepted "delegation patterns". Components made using different simulation patterns cannot be effectively used together and the amount of coding required to implement the patterns and to modify the resulting software when further classes / requirements are added varies widely. This is a major hindrance for the cost effective production of reusable application software, since delegation patterns are at the core of many other widely used design patterns (e.g. state, strategy, flyweight, visitor, decorator).

In this context the contributions of this paper are two-fold:

- On one hand, the paper reviewed, extended and compared the different classes of delegation patterns and gave a recommendation for a "preferred" pattern along with criteria that will help programmers determine the most suitable approach with respect to their specific application requirements.

- On the other hand, the paper sketched the desing of an extension of Java that incorporates delegation as a first class concept overcoming the limitations that characterize even the “best” simulation patterns. We hope that LAVA will be a fruitful stimulation for other language designers and implementors.

8 References

[1] M. Abadi; L. Cardelli: *A Theory of Objects*. Springer, 1996.

[2] J.O. Coplien: *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1992.

[3] P. 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, Computer Science Department III, 14. January 1998, 125 pages.

[4] O.J. Dahl; B. Myrhaug; K. Nygaard: “SIMULA 67 Common Base Language”. *Norwegian Computing Center, Oslo, Report*.

[5] K. Fisher; J.C. Mitchell: “Notes on Typed Object-Oriented Programming”. In *Proceedings of TACS '94, LNCS 789*, pp. 844-885. 1994.

[6] K. Fisher; J.C. Mitchell: “A Delegation-based Object Calculus with Subtyping”. In *Proceedings of 10th International Conference on Fundamentals of Computation Theory (FCT '95)*, vol. 965, *Lecture Notes in Computer Science*, pp. 42-61. Springer, 1995.

[7] E. Gamma; R. Helm; R. Johnson, et al.: *Design Patterns - Elements of Reusable Object-Oriented Software*. Reading, MA: Addison Wesley, 1995.

[8] A. Goldberg; D. Robson: *Smalltalk-80: The Language*. Reading, MA: Addison-Wessley, 1989.

[9] J. Gosling; B. Joy; G. Steele: *The Java Language Specification*. Addison Wesley, 1996.

[10] W. Harrison; H. Ossher; P. Tarr: “Using Delegation for Software and Subject Composition”. *IBM Research Division, T.J. Watson Research Center, Research Report RC 20946 (922722)*, 5 August 1997.

[11] F.J. Hauck: “Class-based Inheritance is Not a Basic Concept”. *University of Nürnberg-Erlangen, Computer Science Department, IMMD IV, Technical Report TR-14-6-93*, July 1993.

[12] F.J. Hauck: “Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance”, *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, vol. 28, no. 10 (1993), pp. 231-239.

[13] U. Hölzle: “*Adaptive Optimization for* SELF*: Reconciling Hihg Performance With Exploratory Programming*”. PhD thesis, Stanford University, 1994,

[14] JavaSoft: “The Glasgow Model”. http://java.sun.com/beans/glasgow/.

[15] G. Kniesel: “Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems”. *Computer Science Department III, University of Bonn, Germany, Technical report IAI-TR-94-3 0944-8535*, October 1994.

[16] G. Kniesel: “Objects Don't migrate! - Perspectives on Objects with Roles”. *Computer Science Department III, University of Bonn, Germany, Technical report IAI-TR-96-11 0944-8535*, Nov 1996.

[17] G. Kniesel: “*Darwin - A Unified Model of Sharing for Object-Oriented Programming*”. Ph.D. thesis (forthcoming), University of Bonn, Computer Science Department III,

[18] G. Kniesel: “Type-safe Delegation for Dynamic Component Adaptation”. *University of Bonn, Technical Report (and position paper for workshop on component oriented programming, ECOOP '98) IAI-TR-98-5, ISSN 0944-8535,* May 1998.

[19] G. Kniesel; M. Rohen; A.B. Cremers: “A Management System for Distributed Knowledge Base Applications”. In *Verteilte Künstliche Intelligenz und Kooperatives Arbeiten (Distributed Artificial Intelligence and Cooperative Work)*, W. Brauer; D. Hernández (Eds.), pp. 65-76. Springer-Verlag, 1991.

[20] H. Lieberman: “Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems”, *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, vol. 21, no. 11 (1986), pp. 214-223.

[21] M. 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, Computer Science Department III, 15. December 1997, 94 pages.

[22] A. Snyder: “Encapsulation and Inheritance in Object-Oriented Programming Languages”, *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, vol. 21, no. 11 (1986), pp. 38-45.