Delegator: Delegation in Java

Klaas van Schelven

December 28, 2005

Contents

Chapter 1

Introduction

Delegation is a powerful concept in object oriented Programming and Design. It provides a means to dynamically add functionality to objects, to share data between objects and to reduce code complexity dramatically. However, true delegation, as presented in this thesis, is virtually unknown amongst software developers and even in the academic world. It is implemented in a small number of languages, such as Simula [8], Self [6], flavors [9], JavaScript and Loops. The most popular Object Oriented Languages, such as Java and C++, have no delegation but rather take a class based approach.

Delegation has a number of advantages over a class based approach, but is especially powerful when used in combination with a class based approach such as is provided by the Java language. This is exactly what Delegator, the subject of this thesis, does: it brings delegation to Java as an Open Source framework. This thesis describes further research on Delegator, which was available as a proof of concept earlier but still had a significant amount of issues.

What prototype based languages are, and what delegation is will be discussed first. Prototyping will be introduced as a concept and delegation as a powerful way of implementing it. Secondly delegation will be put forward as an important concept in its own right. Since delegation allows for powerful dynamic manipulation of the class or object structure, the number of ways in which code can be reused is much higher than in a class based approach. Thirdly the state of the art will be examined. A number of alternatives to this thesis' research will be analyzed on their strengths and weaknesses. Next the Delegator API will be introduced. The interface it offers to its users will be explained, along with a number of implementation issues. Finally, and mainly, a number of important improvements to Delegator will be discussed. This thesis describes the results of extensive research on combining Delegation with Java's access modifiers, exception model, concurrency model, on further improving performance and on a number of other issues. As a result of this Delegator is now available to the public as a stable and well documented framework for delegation in Java.

Class based and prototype based languages

The system of inheritance that is used in Java and a number of other Object Oriented Programming Languages offers a means of describing the world by dividing it into classes and objects. Classes can be subclasses of other classes, inheriting their ancestor's behavior and possibly extending it with their own. They usually describe their objects' fields to store data and have methods to manifest some kind of behavior. Instances of the classes, called objects, inherit all their class' behavior. Different instances of the same class may differ, each of them having its own specific values in the data fields, but having the same fields and methods. For example, say there exists a class `Point` with the fields `x` and `y`, all instances of this class will have those fields, but each instance may have its own values for them. We will call this model the class based approach of Object Oriented Programming.

What does using this class based approach mean for the way reality is modeled? Let's examine this question by looking at how it is used to model a number of animals: a tiger called Anna (having a name, claws and striped fur), a tiger called Bob (same properties) and a bear called Chris (having a name, claws, a brown fur and a love for honey).

In a class based approach, classes need to be written before objects can be created and described, because every object is the instance of some class. To be able to describe the tiger Anna there needs to be a class of tigers first, describing the tiger properties (having a name, claws and striped fur). Other instances of the same class, like the tiger Bob, are then easily created based on that class description. For creating the bear Chris the same principle applies: a class of bears is needed first. A superclass of animals can then be deduced containing some commonalities like the having of claws and a name. Of course, this superclass can also be created before the subclasses using design upfront.

Though this approach models a natural way of human reasoning quite closely, it has some limitations. Describing single instances is, for example, quite counterintuitive, because even for animals that are the only examples of their species, separate classes describing just that one animal need to be created. Secondly, the structure of subclasses and superclasses leads to a tree shaped hierarchy. Sometimes, however, combining properties from objects in very different places of the hierarchy is desirable. Lastly the structure of the class hierarchy is fixed at compile time, making dynamic adaptations impossible.

Another way to describe the three animals is to start with describing the individual instances. So, we describe our favorite tiger Anna as having a name (Anna), claws, a high speed and striped fur. Then we consider another animal, and describe it as having all Anna's properties but different name: Bob. We then describe Chris as being like Anna, but having a different name (Chris), no striped but a brown fur and a taste for honey.

This way, we describe the actual objects as we see them, rather than as them being instances of a particular class. We call this the prototype-based approach, because rather than reasoning about a class of tigers we just take a prototypical tiger (Anna), inherit its behavior and make some adaptations for the other tigers. Rather than describing a class claw-animals to find common behavior between bears and tigers, we just describe one bear as looking like a tiger with some differences. If we need to describe other bears, we can describe them as looking like the first bear.

Note that even in a pure prototype based approach, with no explicit support for classes at all, one can still sneak in a class based approach if that proves to be more useful at some point. To do so simply create an object that represents a class and use that as a prototype. In the example, instead of basing all other tigers on the first tiger Anna, one could also base both of them on a special prototiger or tigerclass. The same goes for introducing hierarchies: instead of a bear looking like a tiger, we could base both of them on a claw animal prototype.

Delegation

How is behavior inherited from one object to the other in both the class based approach and in a prototype based approach? In a language using the class based approach inheritance of behavior follows the superclass-subclass relationships. When a method is called on a certain object, it is first looked up in the class of that object. If it isn't found there it is looked up in its superclass, and so on until the root class, usually called `Object`, is reached.

Prototype based languages are most often, though not always, implemented using delegation. In delegation, objects don't have classes or superclasses but may have a delegate. Each object is firstly able to respond to the method calls that are implemented by its own methods. All other method calls are automatically forwarded to its delegate, which may recursively forward it until some object without delegate is reached. Forms of delegation with multiple delegates per object can also be defined, in which case there is a need for some mechanism to determine in which order the delegates are tried. To use delegation to implement prototypes, simply let each object delegate to its prototype.

A number of authors refer to the concept that was just introduced as delegation as inheritance and speak of child and parent object as opposed to delegators and delegates. Strictly speaking they have a point: behavior is inherited by one object from another. In this paper however, we will show how delegation can be added to Java, a language which only has class based inheritance. To keep the distinction clear, we will speak only of inheritance when referring to Java's class-based, built in system.

In this paper we will define *true delegation* as having two important properties. Firstly it is automatic. Secondly it provides a solution for the *self problem.* Concepts that have the first property but not the second will be referred to as *automatic message forwarding* or *automatic forwarding.*

Automatic message forwarding

The first part of our definition of delegation states that it is automatic. Behavior that looks like delegation, but is less powerful, can be mimicked using manual message forwarding. In the following, a clear distinction will be made between the two concepts. Unfortunately, a number of authors do not make this distinction and use the word delegation to denote the less powerful manual message forwarding.

Manual message forwarding is the explicit forwarding of method calls to another object. This object is often called the forwardee or the delegate, though the latter is a bit of a misnomer. All methods that need to be forwarded to the forwardee need to be implemented in the original, i.e. the forwarding relationships are explicitly programmed by the user. These implementations is simple: they contain a calls to a methods with the same name and parameters on the forwardee. In automatic forwarding and true delegation, on the other hand, there is no need for such explicit forwarding code, since all forwarding is automatic. Any method call that isn't implemented by the original object is automatically tried on the delegate.

An example will clarify this. Say there is a `Square` object that needs to know its x and y coordinates to be able to draw itself. To implement this using manual message forwarding, simply implement all the `Point`'s methods in the `Square`, using a link to the forwardee's equivalent method (see Figure 1.1). This amounts to code duplication, since all forwarding methods are conceptually the same and their implementation can be deduced from the interface of the forwardee.

In true delegation or automatic message forwarding, there is no need for

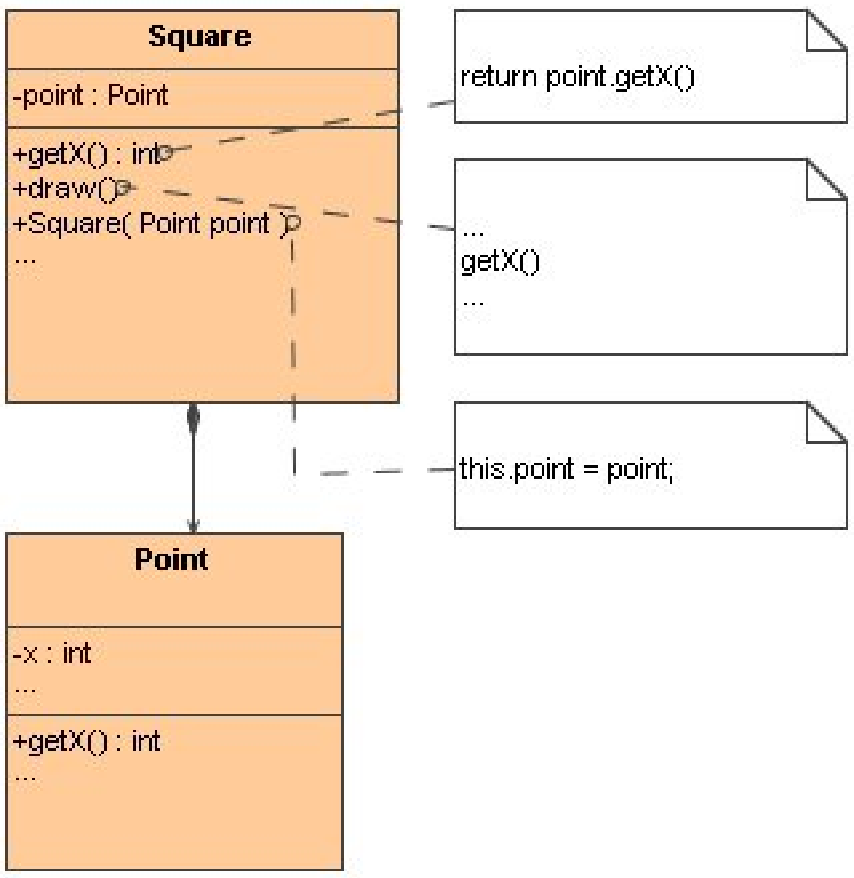


Figure 1.1: Manual message forwarding

such code duplication. Simply make sure that the original `Square` object delegates to the correct `point` and the rest is done automatically (see Figure 1.2).

The self problem

The second condition for true delegation is that it provides a solution for the self problem. The self problem was first explained by Lieberman in his paper about delegation [1]. We will first provide an intuition of it, a more precise definition will be given next. Intuitively, the self problem is defined as such: when a method call is passed to some object, it should respond as if it implements it itself, no matter which object from the chain of delegates really does so. Again, there will be a clear distinction between concepts that do not solve this problem, which will be referred to as message forwarding

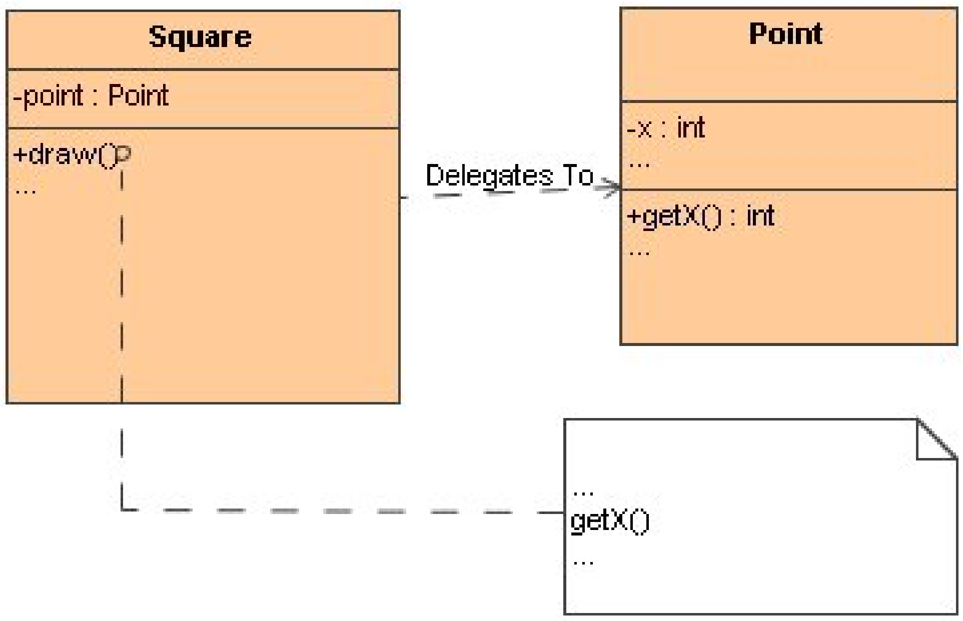


Figure 1.2: Automatic message forwarding or true delegation

(either manual or automatic) on the one hand, and delegation on the other hand.

Now for a more lengthy definition. In the following, the original receiver of a method call will be called the *client*. The variable that is bound to this receiver will be called the *self pointer*, though to confuse matters it is actually denoted by the word `this` in Java. The self problem is to make sure that, no matter how much methods are delegated to other objects, the self pointer stays bound to the client.

It is worthwhile noting that within the context of Java's class based inheritance the self problem is solved by definition. Whenever a method call is not dealt with by an object's class but by that class' superclass, the self pointer stays bound to the client. However, straightforward attempts to implement delegation using some form of message forwarding are troublesome, since the self pointer is rebound to the forwardee at the point of forwarding. At the moment of rebinding, the reference to the client is lost and subsequent method calls that should be either dealt with by the client, either directly or again using delegation, are now dealt with by the forwardee.

The following example, which uses delegation to extend behavior, will clarify this. Say there is a class `MyList` that contains only one method: `add(Object o)`. Some instance of this class is programmed to delegate to

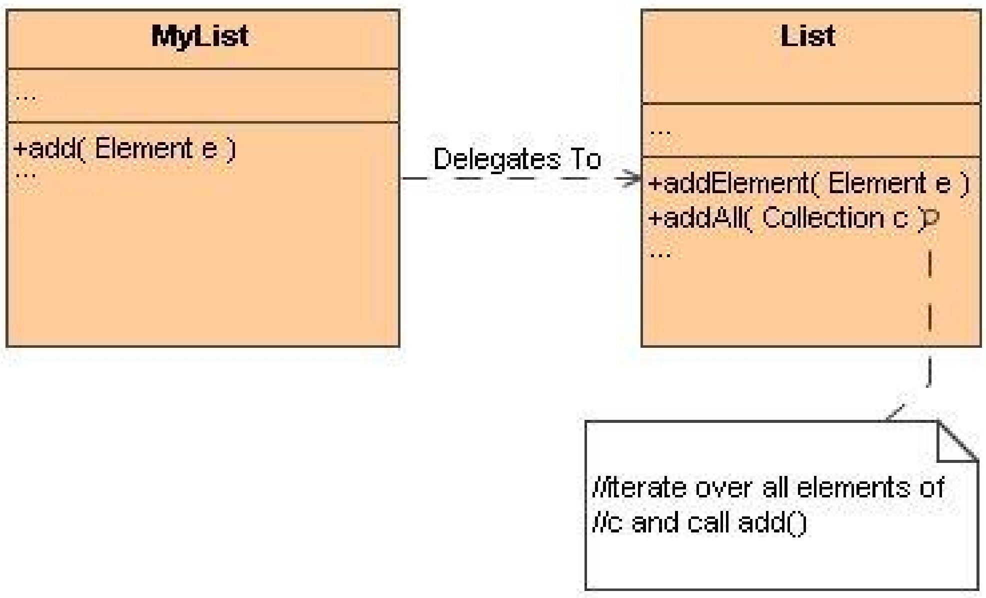


Figure 1.3: The Self problem (Class Diagram)

some `List`, resulting in an object that has `List` behavior with a new definition of `add`. The equivalent in class based inheritance is to create a subclass of `List` called `MyList` with the single method `add(Object o)`. Let's further say that the method `addAll(Collection c)` is implemented as an iteration over that collection with a call to `add` for each of its elements.

What is the distinction between message forwarding and delegation? In both cases, when the method `addAll` is called on an object of type `MyList` that either delegates or forwards to the `List`, the method is delegated or forwarded, since no implementation is available in the `MyList` code. The execution of `addAll` will then result in a number of calls to `add`. Of course, the desired behavior is that the client (`MyList`) receives these calls, so that `MyList`'s adapted version of `add` is used. The distinction between delegation and message forwarding is that the former indeed uses the adapted version of `add`, whereas the latter calls `List`'s `add` method and breaks the code (see Figures 1.3 and 1.4).

Advantages

Delegation provides a number of advantages over class based inheritance. A number of these follow directly from the definition of delegation. Firstly, since delegation is an advanced form of automatic forwarding, the reduction of code duplication following from that concept comes for free. Secondly, the fact that in delegation methods are delegated to objects as opposed to classes means that data may be shared amongst different objects and legacy code may be reused. Thirdly, delegation is dynamic, as opposed to the static inheritance chains from class based inheritance. Finally, since delegation with multiple delegates per delegator amounts to multiple inheritance, the advantages of that language feature are included automatically.

A number of design patterns, such as the Decorator, State and the Strategy pattern proposed in the famous book by the Gang of Four [19], depend heavily on message forwarding (or, as they call it, delegation). Using delegation to implement these reduces code duplication and reduces the maintenance burden. In fact, we'd like to point out that the very existence of these particular patterns indicates the lack of a widespread acceptance of delegation.

In delegation object can delegate to other objects, as opposed to just inherit from other classes. In a system of class based inheritance reuse between objects is limited to methods and field descriptions, i.e. classes and functionality can be reused but objects and state can't. In delegation state can be freely shared between different objects, leading to a reduction of code duplication. Another advantage that follows from delegating to objects is that it eases the adaptation of existing legacy or library code. Objects that are either provided by library factory methods, or classes that cannot be adapted, can be adapted using a new object with extended behavior that delegates to the existing object.

In the case of class based inheritance all chains of inheritance are determined statically, i.e. at compile time. A Java `Button` will always be a subclass of a `Component` and a `Component` will always be a subclass of `Object`. One may change this in the code, but it isn't possible to swap these subclass relationships during execution, for example to make `Button` a subclass of `MyComponent`. Delegation, on the other hand, is dynamic. This adds more possibilities for code reuse. This firstly allows for a more clear separation of concerns and secondly opens up the possibility to replace any object representing one of these concerns at runtime.

Delegation, if it allows multiple delegates, provides a way to implement multiple inheritance. One simply instructs one object to delegate to a number of others, indicating an order. Methods that can't be dealt with by the delegator will then be tried on the delegates in that order. Which of the superclasses should be used if a method is implemented by both, is made explicit by the order of the delegates.

The combination of these advantages allows for an even more clear separation of concerns. Secondly delegation proves to be more intuitive with respect to indicating how close objects are related. More extensive examples of the use of delegation and these advantages are provided in one of the final chapters.

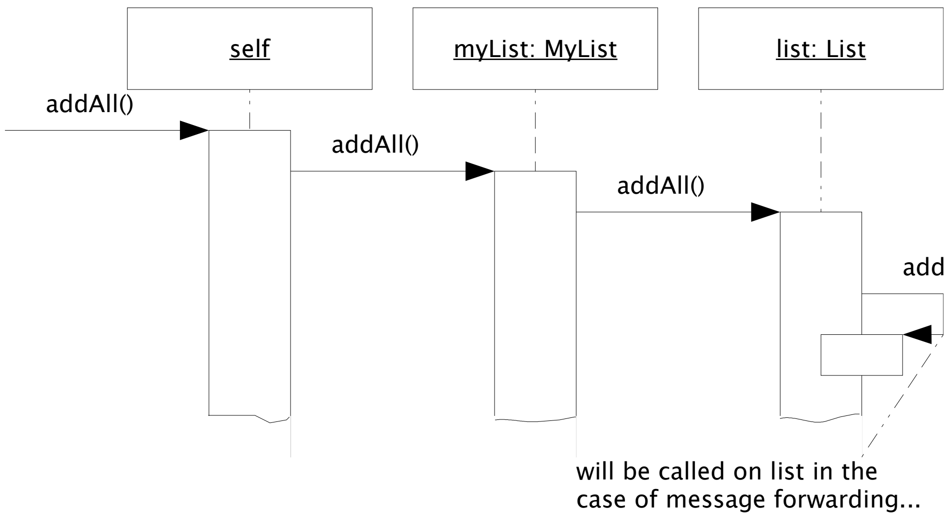
A note on notation

At a number of points code examples and diagrams will be provided; additionally there will inevitably be some Java code in the main text. The reader is expected to be familiar with (class based) Object Oriented Programming in general and as such to be able to understand the code examples provided. The way Java specifically deals with a number of issues, such as access modifiers and synchronization, will be explained in the relevant sections.

Delegator makes heavy use of byte code generation. However, examples of generated byte code will, as much as possible, be given in regular or pseudo Java code. We believe that, although this that at times means we have to resort to constructions that strictly speaking are not part of the Java language, it greatly improves readability.

We will adopt the well known convention that classes will be identified by names that begin with an uppercase character, whereas instances begin in lowercase. Whenever a instance identifier is used without further introduction it can be assumed to be a or the instance of the class with the same name. For example, if we say to call some method on `button`, one can assume it to be the only relevant instance of class `Button`.

Delegator is all about extending Java in such a way that it provides delegation, i.e. some manipulation of classes and methods. At the same time, Delegator is written completely in Java itself, being nothing more or less than a number of Java classes, each having a number of methods. When speaking of class or method it may therefore at times be unclear whether this is a class or method that is part of the Delegator API, or one that is being manipulated by it. In the following, when no further explanation is provided, the words *class* and *method* refer to classes and methods Delegator is manipulating, whereas Delegator's own classes and methods will be explicitly denoted as such.



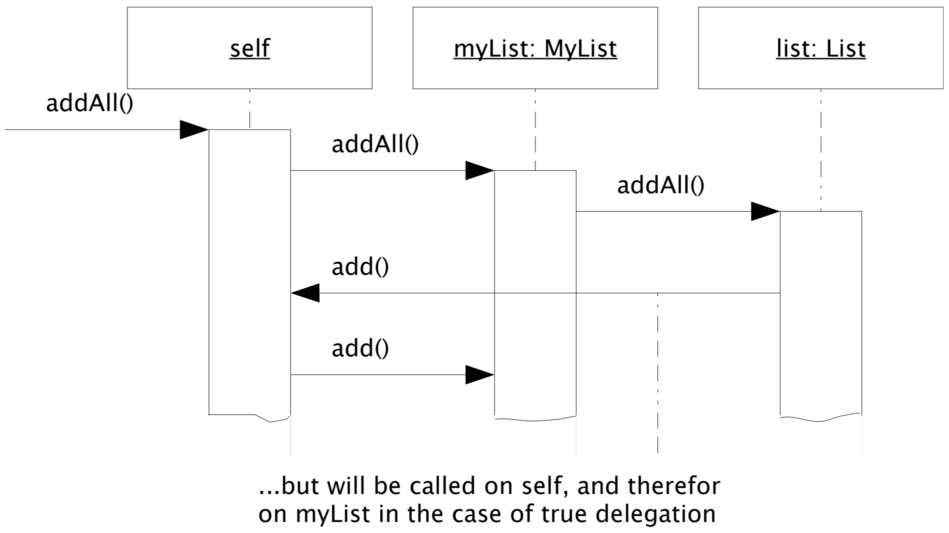

self
myList: MyList
list: List
addAll()
addAll()
addAll()
add()
add()
...but will be called on self, and therefor
on myList in the case of true delegation

Figure 1.4: The Self problem (Sequence Diagram)

Chapter 2

Delegation in Java

Purpose

The Java programming language is currently one of the most popular programming languages. Java programs can run on multiple platforms on a special Java Virtual Machine (JVM). The JVM and Java compiler freely provided on the internet by Sun Microsystems. Java has a class-based approach to Object Oriented Programming, combined with a strict static typing system.

Prototype based programming and delegation are, though very powerful concepts, hardly known and undervalued by the software development community. The reasons for this are largely historical. The first popular Object Oriented Programming Language, Smalltalk, has a class based approach. Even though at that time prototype based languages were available, this set the standard for Object Oriented Programming. In fact, most people in the community assume that a class based approach *is* Object Oriented Programming. Perhaps the most striking example of how unknown delegation is, is the need for this thesis to explain it in its introduction, whereas knowledge about class based inheritance is assumed.

Having a stable implementation of delegation for Java would bring the advantages of delegation to a large audience. Hopefully it would help to bridge the gap between prototype based languages and class based languages. Additionally it would bring the best of both worlds: classes could be used for speed and static typing, and delegation could be used for the benefits mentioned above. An implementation as a library, requiring no extra tools in the development process other than the addition of one more `.jar` file to the list of libraries, would ease adaptation.

Additionally we believe delegation as a concept itself deserves rehabilitation, and we believe Delegator could provide the means for that. Since most languages supporting delegation had little industrial application very few modern programmers ever actually had a serious opportunity to use it. We feel that providing delegation for Java may be a first step in realizing delegation's rehabilitation everywhere.

Goals

When developing an implementation of delegation in Java we will strive to achieve a number of goals. In the above a number of advantages are mentioned: automatic forwarding, dynamic composition of objects, sharing of state among different objects and multiple delegates per delegator, jointly leading to various ways of code reuse. Ideally, an implementation of delegation provides all of these. In any case, in the following it will be remembered that these properties form the very reason for implementing delegation and, whenever tradeoffs must be made, these properties will be held to be key.

In the above, one of the reasons mentioned for implementing delegation in Java is that to promote actual use in the industry. The likelihood of adoption of frameworks or tools is a function of the perceived benefits and the cost of adoption. One of Delegator's goals will therefore be to keep these costs as low as possible. Though implementing delegation using special tools such as precompilers might be easier for those implementing, we feel that in practice the extra costs on the build process would be too high for most possible users. The easiest and most common way to add any tool, library or framework to Java is to download it as a single `.jar` and include it in the classpath. Therefore, we have chosen that doing this will be the only requirement to use Delegator. Having said that, this does not restrict us in any way to add optional, more complex, tools for advanced users.

Another important factor in the decision to either use or ignore new technology is the cost of learning it. This is the case on a multitude of levels, starting at the understanding of the concept of delegation itself, moving through the recognition of new patterns when using delegation to a clear understanding of the inevitable exceptions and subtleties. Learning will be eased by keeping the interface simple and providing enough learning material.

Secondly, to ease learning, a number of code samples will be provided as examples of the possibilities of delegation. Some of these will be presented as JUnit tests, providing a clear contract for the framework and a sandbox for new users to experiment in.

Keeping code simple and reducing duplication are two major underlying reasons for having delegation in Java in the first place. It is therefore of paramount importance that such an implementation does not introduce so much overhead of its own that the result is nullified. In fact, considering all other costs of adaptation, using delegation should lead to a substantial drop in duplication and increase of simplicity. An important goal is a maximum of transparency, meaning that, when comparing code using delegation and code using regular Java, these should behave identically on all other accounts than the means of inheritance.

Performance, both in terms of speed and of memory should be good enough so programs containing the implementation can be run in realistic settings. If a framework is to slow it won't be adopted, no matter how clever or useful it may furthermore be. Finally Delegator will be robust enough to be used in production code. A major factor in the decision whether to use any framework is the likelihood of that framework crashing, taking your product with it. We therefore demand JUnit tests to be provided for all functionality. This both ensures a much higher robustness of the product than would otherwise have been possible, and provides any user with an automatically checkable specification.

Context

Now that we've set our own goals, what is it that can be expected from Delegator's users? In the following, we will assume our users to be forerunners in the field. It is unlikely that novice users will resort to the concept of delegation before they have actually run into the kind of problems that make its adoption valuable. Since Delegator's users have proven to be bold enough to download a package that is so much on the edge of development, we feel it's justified to expect them to deal with some level of complication. At some points there will be a number of options available to our users and we expect them to read through the necessary sections before coming to a decision. Secondly, when users start exploring the boundaries of the possible, we expect them to read about the explorations that have already been made.

This is not to say that we expect a large investment in learning from the outset. The initial interface that will be offered to the user is very simple, and initially no understanding of our implementation is expected.

Finally, delegation at some point compromises Java's static typing, which is a key concept for the language. This may then lead to code that exposes its problems only at runtime. We leave the responsibility for dealing with this extra complexity on the user. However, with the advance of Test Driven Development, we do point out to the user that frameworks like JUnit are available for dynamic testing, and that it is indeed advisable and good practice to use such frameworks in the first place.

Summarizing, we expect a forerunner in the field, someone advanced enough to know for example about JUnit, but offer her a simple initial interface.

The state of the art

Multiple solutions for an implementation of delegation in Java have been proposed and implemented. Before moving on to a discussion of Delegator it will be examined to what extend each of these are viable solutions with respect to the goals mentioned above.

JavaSoft

JavaSoft proposed a convention for something they called delegation for its JavaBeans architecture (the Glasgow proposal), but withdrew this due to public criticism. In fact, regarding the definition of true delegation from this thesis, the proposal provides a solution for the self problem, but no automatic forwarding. The proposed Javabeans setup is therefore not really an implementation of delegation, but rather a set of coding conventions that supposedly ease the mimicking of the concept. They demand the programmer to manually pass around self pointers in some way, either by passing them as a parameter, or by storing them as a field.

Needless to say this approach has a number of rather large drawbacks, most obviously its complete lack of transparency, the resulting code clutter and the fact that the code of existing classes will have to be adapted to allow for use in the context of delegation. A more detailed description of the Glasgow proposal's drawbacks can be found in Kniesel's commentary [12]. There are no indications of delegation becoming in any way officially part of the Java language soon.

Jamie

Several other products exist that provide or claim to provide very limited forms of delegation, mostly based on precompilers. An example is Jamie [13]. In fact, Jamie is really not delegation, but rather a way to introduce multiple inheritance into Java.

Unfortunately, a large number of the other advantages of true delegation, such as sharing of state and the dynamic manipulation of the delegates, aren't available in this setup, because it is based on delegating to a class, not an instance. Jamie works for Java 1.1 and is no longer maintained.

Darwin/Lava

The most promising implementation for delegation in Java, other than Delegator, is the Darwin project, which maintains the programming language Lava. Lava uses special language constructs which are translated into pure Java by a precompiler. Unfortunately the project has been inactive since 2002.

The project has a large number of limitations, though these are claimed to be due to lack of manpower rather than to conceptual problems. The most important ones are that a superset of Java 1.3, but not the entire Java 1.4 or even 1.5 language is supported (rendering the project practically unusable in these times) and that external types, i.e. classes that have not been compiled in Lava cannot be delegated to (canceling the possible benefit of extending legacy objects). Furthermore, with respect to the set of goals from this thesis, Lava misses the mark because it requires the use of a precompiler to begin with.

To get a feel for the Lava syntax, consider the following example.

```
public class Square {

  mandatory delegatee atomic Point point;

  public Square(Point point) {
    this.point = point;
  }

  //local behavior, such as draw()

}
```

The class `Square` may freely use methods from the delegatee `point`. The delegatee can be passed as a parameter of the constructor, as is done here, but it may also be manipulated using a setter. In Lava the fact that a delegatee is used and its type are fixed in the code of the class of the delegator. We'll see Delegator has a different approach, in which these bindings are created outside of the delegator's class.

The fact that a delegatee itself doesn't do any further delegation is indicated by the keyword `atomic`. Atomic delegatees are used to ensure there are no circular references between objects. Lava currently doesn't support multiple delegatees per delegator. Using multiple delegatees introduces similar problems as multiple inheritance because delegatees may have different methods with equal signatures, leading to the diamond problem. Since Lava doesn't provide a mechanism to order the search in the delegatee classes, it excludes the possibility of multiple delegatees altogether.

The keyword `delegatee` supports true delegation, i.e. including a solution of the self problem. By using a different keyword, `forwardee`, Lava also supports automatic message forwarding.

Lava manages to work with delegation within the strict Java typing system because the precompiler gives a type safe result. Another safety check is that delegatees that are marked with the keyword mandatory may never refer to null. This is dynamically checked and an exception is thrown if an assignment to null is made.

Chapter 3

The Delegator API

Delegator is an implementation of delegation in Java as a Java library. It includes both automatic forwarding and a solution to the self problem. Delegator was developed by Erik Groeneveld for Seek You Too Software B.V. in the period 2001 - 2004 as an open source project and was presented at OT2004. Its source can be found on sourcefourge.net. The adaptations the author made to Delegator are the main subject of this thesis. Wherever differences between the versions of Delegator before and after the authors work are being described the former will be indicated as *Groeneveld's version*.

In the following Delegator's interface to its user will be presented. Delegator's approach is slightly different than the one seen so far. To bridge the gap between the two approaches the concept of the empty container *Self* will be introduced first. Secondly a number of implementation issues will be discussed. Groeneveld's approach will be introduced, followed by a number of newly solved problems.

An approach using Self

Above, we distinguished between the concepts of a delegator and a delegate: a delegator may delegate to one or more delegates. If we call any method on the delegator, it will first attempt to deal with this call itself, and if this fails it will look for an implementation in the delegates.

Using these distinct concepts of a delegator and a delegate is perfectly natural in explaining the idea of delegation. Consider for example the case where we extend the behavior of a button with the behavior of an observable object. It would then be reasonable to say we deal with a button (a delegator) that delegates a small part of its behavior to an observable (a delegate). Though this difference is useful in reasoning about delegation it is not fundamental. In the following we will therefore abstract from it.

To do so, we will introduce the concept of a component. Note that both delegators and delegates are simply objects that are able to deal with a fixed set of method calls. Components are therefore defined to be non delegating objects that are used to compose delegating objects. Both a delegator and a delegate are abstracted to a component by taking the methods that are actually implemented by them, as opposed to including the methods that deal with adding and removing delegates and such.

Additionally we introduce the concept of a container that contains a (possibly empty) ordered list of components. We will call methods on the container, which forwards the calls to the components. Additionally the container offers an interface for the maintenance of the list of components: adding, removing, inserting, etc. This container acts as a single object to the outside world. We will call it *Self*, since it is this container that acts as the client in the self problem.

Note that the approach using a delegator and delegates is indeed equivalent to the approach using a container with components. To translate a solution formulated using the first approach to one using the second, simply add the delegator as the first component of the container; to translate back simply take the first component and make that a delegator.

The formulation of the self problem also changes slightly in this approach. Remember that the self problem says that calls from within object to themselves should be interpreted as calls to the original receiver, called the client. In the delegator / delegate approach this reads as: calls from within delegates should be interpreted as calls on the delegator. In the self / component approach this reads as: calls from within components should be interpreted as calls on self.

An interface in Java

The concept of a Self with a number of components is transposed directly to the implementation in Java. Firstly we will discuss the interface that is offered to the user of the API, the programmer.

The concept of Self is translated directly into an Java class `Self`. This class offers a number of methods for the maintenance of the list of components, i.e. adding and removing components by class or index and reordering them. The component implementations are shielded from the programmer and will be discussed later. Self, our unit of access, is the only thing exposed to the user.

Java's static typing

In the conceptual discussion on delegation it was simply stated that methods can be called on instances of `Self`. However, in the context of Java and its type system, this is impossible to do directly. Consider this example.

Say we've created an instance of `Self` called `self` with as one of its components a `HashMap`, implementing the following method:

```
  put(Object key, Object value)
```

How can a programmer call this method `put` on `self`? It is impossible to simply use `self.put(key, value)`: the class `Self` of which `self` is an instance has no method `put` so Java's type system forbids this. Casting self to a `HashMap` doesn't really help either: `self` is no instance of the class `HashMap` so this results in a `ClassCastException` or even a compiler error.

To enable calls to methods on `self` its interface is extended by a method `cast(Class clazz)`. Note that since Java also provides a mechanism called *casting*, that term is now overloaded. In the following the term "the method `cast`" will refer to the method on `Self`, whereas the term "cast" or "cast using java's casting mechanism" will refer to the built in functionality casting objects down to more specific classes.

The method `cast(Class clazz)` returns a *proxy* for the instance of `Self` it is called on of the class or interface `clazz`. In our example, a call to `self.cast(HashMap.class)` would return an object that is a `HashMap` and knows it represents self. Because the method `cast` may return objects of any class depending on its parameter its return type is `Object`. A cast, using Java's casting mechanism, of the result to a `HashMap` is therefore required. This results in the following rather cryptic line:

```
((HashMap) self.cast(HashMap.class)).put(key, value);
```

Obviously, factoring the casting mechanism to a local variable out will pay off in many cases:

```
HashMap map = (HashMap) self.cast(HashMap.class);
map.put(key1, value1);
map.put(key2, value2);
```

Some utility methods are provided by the static class Delegator that serve to simplify notation. A call to the method `extend` with as parameters `Class subclass` and `Class[] superclasses` returns a proxy of type subclass for a `Self` having subclass and superclasses as its components. Of course, one may also simply see it as an object that is a subclass of multiple superclasses without understanding the underlying implementation. This provides a rather natural notation for multiple inheritance.

```
Map map = (Map) Delegator.extend(HashMap.class,
  new Class[]{WebPage.class, TreeNode.class});
```

Referring to other components

The mechanisms provided so far allow programmers to compose objects and refer to the various methods of the components from outside the composed object. However, the very reason to compose objects is that there is some communication between the various parts. To make such communication possible, the programmer will typically have to pull one more trick to circumvent Javas static typing. In fact she can choose among two available tricks to do so. As a running example a `Square` that delegates some information to a `Point` will be used. The `Square` uses information about its `x` and `y` coordinates, available via the `Point`'s methods, in its `draw` method. Again, since the `Square` has no such methods of its own, the Java compiler complains.

The first option is to use an interface that contains all methods of the delegate, and let the delegator declare to implement this interface. Of course, the point of automatic forwarding is not to let the programmer actually implement all these methods. Therefore, to make sure the compiler doesnt complain about the lacking implementation of the interface, the class of the delegator will be declared to be abstract:

```
public interface IPoint {
  public int getX();
  //...
}

public abstract Square implements IPoint {
  public void draw() {
    //...
    x = getX();
    //...
  }

  //other square-specific methods
}
```

```
//in the test class:
public void testOne() {
  Square square = Delegator.extend(Square.class, Point.class);
  square.draw();
  int x = square.getX();
}
```

This approach enables the user to make references to `Point` specific methods from within the `Square`'s code. Obviously, the `Square` can no longer be created using Java's traditional `new` operator because `Square` is now abstract. Therefore, the task of creating a component based on this class is left to the Delegator API

The second option is to add abstract methods to the delegating object. Sometimes adding a separate interface is overkill, for example if only one or two methods from the delegate are needed, the delegate has a long interface and that interface is not already available separately. In that case these methods can be added as separate, abstract methods to the delegator:

```
public abstract Square {
  public abstract int getX();
  //other getters and setters from Point

  public void draw() {
    //...
    x = getX();
    //...
  }

  //other square-specific methods
}

//in the test class:
public void testOne() {
  Square square = Delegator.extend(Square.class, Point.class);
  square.draw();
  int x = square.getX();
}
```

A third option is to cast to the delegator to the class of the delegate where required. A lack of transparency following from the fact that the use of `Point` is made more explicit, is at the same time this method's main advantage and disadvantage:

```
public abstract Square {
  public void draw() {
    //...
    x = ((Point) cast(Point.class)).getX();
    //...
  }

  //other square-specific methods
}

//in the test class:
public void testOne() {
  Square square = Delegator.extend(Square.class, Point.class);
  square.draw();
}
```

Convention

When using Delegator the user is often faced with a recurring pattern. We provide a convention for dealing with that here. In Delegator the coupling of the delegator to the delegate can be done anywhere and at any time. However, sometimes it is very clear that there is only one delegator and a fixed number of delegates. In those cases we suggest to put the code for creating new instances in a static method `create()` of the delegator.

Implementation overview

This section provides an overview of the Delegator API. To be more precise, it is an overview of Groeneveld's version's inner workings. This will provide insight in how the self problem can be solved and how delegation can be integrated in static typing, while leaving some details for later. As seen above, the Delegator API consists of three main parts: proxies, `Self` and components. An overview of how these three parts interrelate will be given first, followed by a discussion of their individual implementations.

To provide a basic understanding of Delegator's implementation a single method call's execution will be traced through Delegator's various parts. As seen above, `Self`'s methods are always called on one of its proxies first.

This proxy then somehow forwards this call to the right instance of `Self`, which then performs a lookup and forwards it to the right component. If the component contains method calls on itself, these are sent back to self which again performs method lookup: this is critical for a solution of the self problem. Let's look at the way these parts of Delegator communicate and the way they are implemented in some more detail.

Proxies

In this section the implementation of the proxies will be discussed. We will first show what the task of the proxies is, i.e. *what* they do. Secondly we will show *how* they do it, first by explaining how proxies know what their particular instance of `Self` is and secondly by showing how they are generated.

A proxy is the result of a call to `Self`'s method `cast`, and enables the user to call methods on `self` which would otherwise have been impossible due to Java's type system. Whenever a method is called on a proxy, it notifies the correct `self` which method was called and with which parameters. However, it cannot do this by simply calling a method of the same name. Such a method does not exist in self, which is the very reason we have proxies in the first place. To provide an entry point for method calls, the class `Self` is therefore equipped with a method

```
  invoke(Object proxy, Method method, Object[] args)
```

The proxy calls this method with as parameters the proxy itself, the method that `self` needs to respond to and its parameters. These methods on the proxy, which for each call simply do a call to `self`'s `invoke` method, will be called *forwarding methods* in the following.

So how do the proxies know which `self`'s `invoke` method to call? Quite simply: the only method that creates new proxies is method `cast` on `Self`. In that method a new proxy is created with a reference to the `self` that created it. This reference is stored as a field of the proxy, and the forwarding methods simply read this field.

Knowing that a proxy is simply an instance of a class containing a number of forwarding methods, how are they created? That depends on whether they are created for interfaces or classes. Proxies for interfaces are generated using java's built in

```
java.lang.reflect.Proxy.newProxyInstance(
  Class[] interfaces, InvocationHandler h)
```

This method returns a new object that implements all interfaces from `interfaces` and forwards their method calls to the `Invocationhandler h`. The interface `InvocationHandler` contains the single method `invoke` described above. The class `Self` implements this interface and at the moment of the creation of the proxy `self` is passed as the parameter `h`.

For classes, as opposed to interfaces, there is no built in functionality to generate proxies. The Delegator implementation therefore provides this behavior itself. In the following, the class that is being modeled by proxy the will be referred to as the *model class*. For each class for which proxies are generated a proxy class is generated first. These proxy classes are generated dynamically using apache BCEL, and are then loaded dynamically using a specialized ClassLoader. The proxy class is given the name of its model class followed by `$proxy`, e.g. `HashMap$proxy`. They are declared to implement the token interface `Proxy` for reference purposes. The individual proxies are generated as instances of their particular proxy class using the `java.lang.Class.newInstance` method. To ensure that the proxy instances can be treated as instances of their model classes, the proxy classes must be subclasses of their models. To clarify this, consider the following line of code again:

```
HashMap map = (HashMap) self.cast(HashMap.class);
```

The Java cast `(HashMap)` is only legal if the result from `self.cast`, the proxy, is either a `HashMap` or a subclass of a `HashMap`. However, since the proxy classes are a subclass of their model, they automatically inherit all of their model's public and protected methods and fields. Of course, the purpose of a proxy is to be a proxy that forwards to some self, not to actually implement behavior itself. Therefore all protected and public methods are overridden in the generated code and implemented as forwarding methods as described above.

Since the proxy classes are in reality created at run time and directly as byte code, there is never any Java code describing proxy classes available. Furthermore, some of the constructions used in the Java byte code (such as the use of a `$` in a class name) are illegal in Java but legal in byte code. However, for purposes of clarity it is interesting to give an outline in Java code of what a proxy class would look like if it existed as Java code:

```
class HashMap$proxy extends HashMap implements Proxy {

  public InvocationHandler self;
  public HashMap$proxy() {
  }

  public Object put(Object key, Object value) {
    Method method = HashMap.getClass().getDeclaredMethod(
      "put", new Class[]{Object.class, Object.class});
    parameters = new Object[]{key, value};
    return self.invoke(this, method, parameters);
  }

  //other forwarding methods
}
```

Self and its method lookup mechanism

The class self really consists of two main parts. Firstly it has a number of methods to add, remove and order its components and create new instances of `Self`. These are trivially implemented as manipulations on an array of objects and will not be explained in great detail here. Secondly, as introduced above, it has a method `invoke` with as parameters the calling proxy, the method to be invoked and its parameters. Whenever this method is called `self` will search in its list of components for a method that matches the given method, and then this method is called. In fact, more precisely, an adapted version of it is called for reasons that will become clear later.

The method lookup is done by simply examining the components one by one in the given order and checking whether they provide a method that matches the signature of the method that was called on our proxy. Whenever no matching method is found in any of the components an Exception is thrown. A contribution of this thesis is, that with respect to Groenevelds version this mechanism has been extended to be more precise.

The algorithm for scanning the components one by one may be simple, but the definition of a match between a method on the proxy and the method on the component is slightly more complex and will be dealt with more elaborately later. For now, it suffices to say that this definition is kept as close as possible to regular Java method calls.

Components

In the beginning of this chapter a new approach was introduced, based on a container called Self, containing a number of components. Note also that the self problem was reformulated for this approach to ”calls from within components should be interpreted as calls on self". Having reformulated the self problem to a problem of the components, it is in this section that a solution will finally be given.

As with proxies, we will first examine the role of components in our model, i.e. *what* they do. Secondly it will be shown how they do that, first by showing how they reference back to `self` and secondly how they have been implemented generally.

So what is the task of these components? The answer to this question is twofold. Firstly the components are the source of the actual implementation or behavior of the composed object that is represented by self. Note that nowhere in the discussion of proxies or self there was any reference to something actually happening. All that has been delegated to the components. Secondly it is in the components that a solution for the self problem is given. This means, that if within the implementation of a component method there is a call to another method of that same component, the component must forward this to self. Indeed, roughly the same forwarding methods as in the case of proxies are used for this purpose.

How do the components know which `self`'s `invoke` method to call? Unfortunately, unlike in the case of proxies, this question isn't trivial. One of the main formulated advantages of delegation, and therefore one of the main formulated goals of Delegator, is the ability to share components among a number of different instances of `Self`. Because components can be shared among a number of instances they can't simply be equipped with a reference to their single owner at instantiation time.

Remember that the correct `self` to call the `invoke` method on is the client of the original method call. If we are able to shield the components from the user, the only way methods can be called on them are via some `self`. This is the premise we'll be working with. Using this premise, method calls always reach some `self` before they reach their actual implementation in some component. It is exactly this `self` that is the client of the call that is being executed in the component. Therefore, the moment to store information about which `self` is the current client is right before calling component's methods. The final challenge is to actually let the component know somehow.

A first intuition may be to simply set a field on the component. However, since some scenarios, such as recursive methods calls or simultaneous method calls from multiple threads require more than one bit of information to be stored, this approach doesn't suffice. In Groeneveld's implementation, the chosen solution is to let the components peek at a (thread local) stack of instances of `Self`. Self pushes itself on top of this stack before calling a method on one of its components and is popped off afterwards.

How are the components implemented? As with proxies, components resemble, but are not exactly the same as the class they model. Therefore, again, a special component class is generated and loaded dynamically of which the instances are used as components. This class has the suffix `$component` and implements the token interface `Component`. Again, the components are a subclass from the class they model. Note however that, unlike in the cases of proxies, this is strictly speaking no necessity, since the components are completely shielded from the user and it is up to the implementers of Delegator to implement calls to them.

Now, components have two main tasks: to implement behavior and to solve the self problem. To implement the component behavior, for each of the component's methods a special implementing method is added. It has a slightly modified signature with respect to the method it implements, for example by adding a prefix to its name. This method calls a matching method in the component's superclass, the model class which actually implements behavior. Of course, in `Self`'s `invoke` method's code, calls must be made to these methods, not to the originals. Again, consider how this would look if it were to be done in Java (as opposed to bytecode):

```
class HashMap$component extends HashMap implements Component {

  public HashMap$component() {
  }

  public Object __supercall_put(Object key, Object value) {
    return super.put(key, value);
  }

  //other supercall methods
  //forwarding methods
}
```

Secondly and finally, we turn to the solution of the self problem. All of the component's protected and public methods are overridden by forwarding methods. These are identical to the forwarding methods of the proxies, with the exception of to which `self` they forward, since they use the stack discussed in this section. Whenever one of the superclass' implementing methods calls a method on itself, this is dynamically dispatched to the component. However, since all of these methods are overridden to be forwarding methods, these calls end up in `self`. Hence, the self problem is solved! See also Figure 3.1.

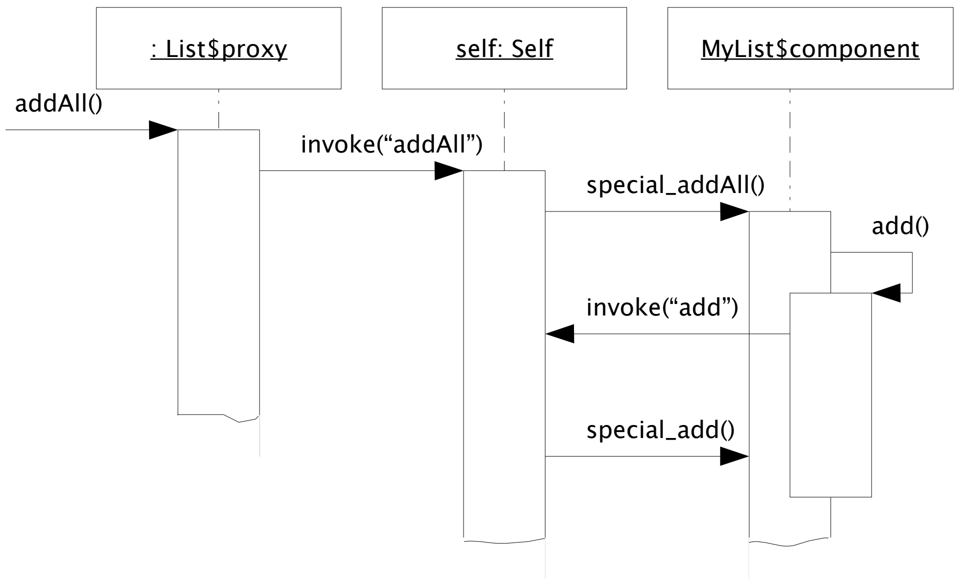


Figure 3.1: The self problem solved

Chapter 4

Java's access control

The Java language provides a mechanism of access control to allow for information hiding and encapsulation. Using this mechanism, each field or methods must be declared to be either private, default, protected or public. The attribution of these *access levels* are made by starting the declaration of the field or method with one of the keywords `private`, `protected` or `public`, or by leaving all of these keywords out. In that case the field or method is of default. Since the semantics of the default access level are much better described by the term *package* the latter term will be used in the following.

Private members can be accessed only by objects of the same class. This includes access across objects, so one object may access the private members of another object if this other object is of the same class. Default or package members may additionally be accessed from objects from anywhere in their own package. Protected members may be accessed in the same ways as default members, but may additionally be accessed from objects of a subclass. A minor caveat is that in that case objects can only access the protected methods of their own instance. Public members may be accessed from anywhere.

This chapter will describe how Delegator includes the concept of access levels and deals with the limitations that are a result of it. Firstly, the fact that the Java language has access levels creates a class of problems of its own. The main reason is that since proxies and components are generated on the fly, they are of a different class than their model class. Since the access levels are based on the class of objects, some things Delegator needs to do are restricted. The issue whether everything that's possible in regular Java will be possible in Delegator will be called *completeness* in the following.

Secondly, again in the interest of transparency, what is illegal in regular Java must also be illegal in Delegator. In other words, Delegator should be as restrictive in Java with respect to access levels. Groeneveld's prototype basically ignored both of these issues. This issue will be called *safeness* in the following.

With respect to the issue of completeness it can already be said that Delegator is *not complete*. Where incompletions occur it will be defended that it is indeed impossible to make Delegator complete. Secondly, it will be shown how the user can work around the issue, and how likely a particular incompletion will lead to unexpected errors.

In both the cases of completeness and safeness the following two code patterns will be compared:

```
//with Delegation:
(SomeClass (new Self(SomeClass.class).
  cast(SomeClass.class))).someMethod();

//without Delegation:
new SomeClass().someMethod();
```

In the following sections the results of these two lines of code will be compared for private, package and protected instances of `SomeClass` and `someMethod()`. If they behave equally we say Delegator has been implemented well for that accessor.

Public access level allows access from anywhere and thus imposes no rules. Since it doesn't do so, implementing it is trivial. Firstly there are no issues with Delegator code not being allowed to call public methods, since there are no rules to prohibit it from doing so. Secondly there is no requirement for Delegator to restrict access, since to model "no rules" no rules are required. The public access level will therefore not be dealt with separately.

Finally the way Delegator deals with fields will be discussed. Though strictly speaking not directly linked to the subject of access levels, the issues surrounding fields are exactly the same.

Private accessor

The `private` access level is the most restricted of the four different types and will be considered first. The completeness of Delegator with respect to the private access level will be examined first. Since Delegator is not complete, it will be shown that it cannot be made complete second. How to work around this incompletion and how dangerous it exactly is will be shown third. The safeness of Delegator with regards to encapsulation will be considered in the passing.

Private methods can be called from three different types of locations, and it is this location that determines whether its possible within Delegator. Firstly an object may always call a private method on itself. Next private methods may be called on other instances of the same class. Finally private methods of inner classes may be called by their containing classes.

Before considering the three cases, consider the implementation of private method calls. In Java Private method calls are implemented differently than protected and public method calls. Whereas the latter are dynamically dispatched, the former are statically linked. To understand why, consider the purpose of private methods. The whole point of private methods is that they are private to their class and should therefore never be overridden in subclasses. Any call to a private method of a class' instance will be dealt with by that class' code, never by a subclass' code. Since the compiler knows at compile time which class provides that method, the link to that method is made at compile time.

Additionally consider what kind of overriding behavior is desirable for Delegator. Seeing that private methods may in regular Java never be accessed outside their own class, we say Delegator should encapsulate similarly, restricting the accessibility of private methods to their own components. We therefore say *the self problem should not be solved for private methods*, since that would presume that another component can deal with a private method as opposed to it being encapsulated.

Now, consider the three different ways in which private method can be called. Firstly, consider the case of an object calling a private method on itself. Say we have a class `PrivateSelfCallingClass` which calls the private method `method` in one of its methods bodies. This call is executed normally in regular Java. If we now create an instance of `PrivateSelfCallingClass` using Delegator, i.e. create an instance of Self with as a single component a `PrivateSelfCallingClass`, we want this behavior to be preserved.

How does this work? The call to `method` in the body of another method is statically linked, meaning that a link to `PrivateSelfCallingClass.method` is created. This means that the private method is neatly dealt with by the component itself, as opposed to redirected via `self`. This is exactly the behavior specified in the above as correct.

Secondly we will examine private methods of inner classes being called from their containing classes. Consider our test case again: we are going to compare calling a method on a `self` with one component and the same method on a normal object. Consider an inner class `PrivateMethod` with one private method `method`. When doing a method call from the containing class on a regular instance of this class we expect `method` to be executed on that instance, which is what happens. Say we create an instance of this class using delegation and call the method `method` on it from the containing class. It turns out that a call of `method` is actually *not* handled by `self` at all, and therefore by its single component, but by the proxy instead.

A proxy actually responding to method calls itself, as opposed to forwarding them to `self`, is very undesirable. The proxy may contain random data and is not really a part of the composed object `self`. In fact, multiple proxies may be created for one `self`.

So how can this happen? As we have seen above Delegator generates proxies as subclasses of the class they are a proxy for. When the method `method` is called on this proxy we want the newly generated forwarding method, the one that forwards the call to `self`, to be executed. However, since the call to the proxy's private method is statically linked, the new, overriding implementation is never reached.

The third scenario is to call a private method on another instance of the same class. If, in such a scenario, the reference to the other class is replaced by a proxy to that class, the code breaks. This is because of exactly the same reason as in the second scenario: a private method is called on a proxy, but since private methods are statically linked the proxy's forwarding methods are never reached.

Solutions and workarounds

This research has set a number of goals. Firstly Delegator is to be a runtime API, functioning within the Java language itself. Therefore, the fact that Java uses static linking of private methods can not be altered for our purposes. Also, in Delegator proxies may be generated for any class. This means there are no restrictions on proxies having private methods. Finally, proxies must be of a different class than the class they are a proxy for, since they need to behave differently. Unfortunately, no solution for this problem is available within these conditions.

Knowing that this problem is unsolvable under the given circumstances we will turn our attention to dealing with it by working around it. Strictly speaking Delegator is not complete with respect to the private accessor. The next question is whether this renders Delegator generally useless, or whether there are such workarounds that all options remain open. We will call this *functional completeness*. Secondly, we know that Delegator is safe with respect to the private accessor, since private methods cannot be inadvertedly called using Delegator. However, the fact that proxies may be answering private method calls themselves poses the question whether unexpected errors will occur. We will call this *functional safety*.

Firstly we will examine the issue of completeness. Does the way private methods cannot be called on proxies limit the programmer?

Say it would be possible for the programmer to replace all called private methods by for example protected ones, by changing their method signatures in the code. We say that though this would be slightly less elegant, it would provide all the benefits of delegation and would often be worthwhile. There are many cases where the programmer indeed has full control over all source code and may choose that the benefit of using Delegator is greater than the loss of elegance resulting from using protected in stead of private methods.

On the other hand, as we have seen, there may also be cases in which the user is not in control of all code. In fact, the very scenarios of reusing legacy code and extending library code have been mentioned as particular cases where Delegator may prove very useful.

There are two cases in which the user may be faced with the fact that private methods don't work in Delegator. Firstly a private method may be called from the containing class of their class in the case of inner classes. Secondly it may be called on an instance of the same class created using Delegator. Is it possible that in one of these cases the user is not in control of the component code and is using Delegator to create the component?

Inner classes can only be accessed from their containing class. This implies they can also only be created from their containing class or from themselves. Assuming file level access rights, they can be created as proxies only in the containing class. Therefore it is clear that the programmer who decided to use proxies can also alter the inner class in such a way that the private methods causing trouble are removed. We can safely conclude that a workaround is available for private inner classes.

If a private method is called on a different instance of the same class a workaround may not be available. There is no restriction on creating objects that call private methods on other instances of the same class in any particular place. Therefore, it cannot be assumed that because a proxy was created, the programmer has access to the code of that class.

Summarizing it is fair to say that Delegator carries itself a long way through Java's access restrictions. Only when the model class cannot be adapted *and* there is a private method call from model code to another instance of that class that happens to be a proxy do we run into trouble.

Safety

What does this mean for the likelihood of unexpected programming errors? Semantically it is nonsense to call private methods on proxies. However, there is no technical limitation on doing so whatsoever. Is there any way this can be combined such that programming errors do not occur on a large scale? The question here is divided in two parts: prevention and restoration. Firstly, how likely is it that the programmer will not make any mistakes? Secondly, how easy will it be to track and restore these mistakes, given that they may be made.

Preventing this type of error to occur is pretty hard. Of course documentation, for example in the form of this document, is provided, carefully describing what is legal and what isn't. At the same time it is not realistic to say that developers will spend a large amount of time reading about possible mistakes and caveats before they have actually run into them.

Therefore it is of paramount importance that, once a mistake is made, it is easy to track. Unfortunately, the private method seems to be executed normally, since it *is* executed, only not on the right instance. Debuggers and `System.out.println` statements will also show it to be executed. Since there is no error at the point where the proxy's method is executed, and yet at that time it is not called on `self`'s component, in a lot of circumstances the error will only show at a completely random point in the program's code. In short, one doesn't expect a method call to be executed on the wrong object. The very point of Delegator is to offer the programmer an interface to a composed object as if it where one. If all of a sudden a method is executed only on part of this object, without any warning, this is highly confusing. Finally, the case in which the user has no access to the source code of the model class is highly problematic, because she cannot even see whether private methods are called in an illegal way or not.

However, as seen above, we do expect users of Delegator to have some interest in the technology they are using. Moreover, we expect them to be using some testing environment like JUnit, at least to test the areas where they are using Delegator. Such a testing environment will immediately indicate an error. At that point in time we expect the user to delve into Delegator's documentation, where this source of errors will be the first to be indicated. At this moment we believe such a framework to provide enough of a safety net. However, if in the future it turns out that this source of errors is indeed a major one, one of the following solutions may be tried.

Restriction to interfaces

A possible restriction on Delegator's casting mechanism would be to allow only interfaces for casting. This would solve a number of the problems considering accessors. In this scenario, it is impossible to accidentally access a proxy's field, private or package method. This protects the user against the whole range of errors described above.

On the other hand, it would introduce the extra burden of having to write interfaces for all the classes to be used as proxies. In what sense this really is a burden depends largely on programming style. Some programmers and APIs work with interfaces a lot anyway.

Furthermore, this would also be limiting in more important ways. Consider the case where legacy code is being reused, and there is a pointer to an object of the same class somewhere in that code. In a setup where only interfaces are allowed, these interfaces can now no longer be used instead of the originals without modifying the code.

A possible middle way would be to allow proxies for classes only after the user has explicitly requested to be able to do so. In that case the programmer is more or less forced to read the documentation about the risks of using the version that allows classes. As explained above, none of these restrictions to proxies have been implemented so far since no need has presented itself yet.

Generating warnings

How about generating warnings for ill uses of private methods on proxies? A naive solution is to generate some kind of warning for every time a private method is added to a proxy. However, this generates a great number of false positives, since proxies may contain a great number of private methods. The only case in which this is problematic is when these are actually called by their clients.

However, when generating a proxy the code of the model class could be analyzed for calls to private methods. Such an analysis could be done at runtime, generating a runtime error when finding such calls, or at compile time with an optional tool. At this point, however, the full analysis of a models class code as bytecode is outside the scope of this thesis.

Package accessor

The package access level is the second most restricted level of access. Like with private methods, we will examine whether Delegator works with package methods in all cases. Again we will notice how Delegator is limited, explain which further solutions are available and examine the implications and risks of these limitations. The safeness with regards to encapsulation will be dealt with in the passing.

Package methods can be called from the same three types of locations as private methods and additionally from anywhere within the package. As we have seen with private methods, the interesting distinctions are between calls to package methods on regular objects and on proxies. We will also see and explain that the package accessors cause similar problems to private accessors. In fact, it will be shown that the package access level is the most complicated access level to implement.

As we have seen previously, proxies are generated dynamically as subclasses of the classes they are a proxy for. The generated proxy classes are put in the same package as their superclass. To be able to load the proxy class and prepare it for actual use a special `ClassLoader` is used, that is distinct from the default one.

Now consider a call to a package scope method from either one of the locations private methods can be called or from anywhere within the package of the method receiver. Again, compare the two cases where firstly this call is made on a regular object, and secondly on a proxy of some component. In the second case, the desired behavior is that the proxy, which has implemented forwarding methods, responds to that call by forwarding it to its `self`. Unfortunately, again, this is not the case.

To understand why not, consider how Java identifies its classes. Java classes are uniquely identified by a 3-tuple: their name, their package and the ClassLoader instance that was finally responsible for loading them. It turns out that because the subclass `SomeClass$proxy` is loaded by another `ClassLoader` than the default `ClassLoader` the JVM does not recognize it to be in the same package as `SomeClass`. Therefore the JVM's method lookup will skip the proxy's method and resolve to `SomeClass`' method, again containing real implementation as opposed to a forwarding method.

Solutions and workarounds

Again, under the conditions chosen for this research no solution is available. Since proxies can be created for any class they may also contain package methods, and there is no restriction on calling these. Also, since the Delegator framework is offered as an API, the default `ClassLoader` cannot be modified. Finally, the JVM's definition of what uniquely identifies a class cannot be modified either.

The problems resulting from nonsense calls to package methods on proxies and their solutions are very similar to their private counterparts and a discussion of them won't be repeated here. It is worth noting though, that the scope of the problems is larger, since package methods may be called from anywhere within their package.

Additionally to the problems concerning proxies there is also a complication in the components. Package methods in components behave identically to their private counterparts. Because overriding methods are not considered, the self problem is not solved for them. In the case of private methods this, however, this was in fact the desired behavior. For package methods, on the other hand, there is no reason to say the self problem should not be solved. Unfortunately this issue is analogous with the issues concerning proxies and cannot be solved. On the other hand, the workarounds are analogous as well, and if source code is available, the access level can be lifted to protected to ensure the self problem is solved again.

On the upside, these limitations automatically guarantee safety with regards to encapsulation of package methods. Since due to the problems describe previously package methods cannot be exposed even within the package, they can surely not be undesirably exposed outside of it.

Since the problems with package methods originate from way they are loaded, replacing the default `Classloader` at program start solves *all* the above problems. However, since the installation of a single `.jar` file has been phrased as one of the goals of this research, this option has not been investigated further.

Fields

As a minor sidestep from the subject of this chapter, consider the use of fields. It must be noted that Delegator simply *does not* support fields in any way. This means that firstly any reference to a proxy's field will be simply manipulating the proxy instead of one of `self`'s components. Secondly, any reference to fields from within a component will be interpreted as a reference to that component's field. In other words, the self problem is not solved for fields.

Consider that all problems mentioned for private and package methods, and their workarounds, also hold for fields of all four access levels. Fortunately, for fields, they are less problematic. Firstly consider the case of the use of fields within components. There is no particular reason to assume the self problem must be solved for them. Secondly consider field access from outside self, i.e. field access on proxies. It is a well accepted convention that there should be no non private fields, and that fields should always be made available via getter and setter methods. This rule has one exception, which is the case of performance optimizations. However, we feel it is unlikely that users will both do performance optimizations and use Delegator in the same place, since Delegator is significantly slower than regular Java. This leaves the issue of private field access on proxies, which is completely analogous to the issues around private methods and proxies.

Protected accessor

Protected accessors are the second most public accessors in the Java language. Methods may be accessed from all the same locations as with package accessor, and additionally from all descendants.

None of the problems of private and package accessor are present in the case of protected accessor. The definition of protected methods is that they can be overridden in any subclass. This is exactly what happens and therefore what exactly what works. Completeness is thereby guaranteed by definition.

Secondly on the issue of safety, let's examine the encapsulation of protected methods. In doing so the access level of both the forwarding method and the actual method will be examined. The forwarding method can be either on the proxy or the component, and is the method that is being called. The actual method is the method on the component that has the actual implementation. As seen above, these methods are matched to each other in `self`'s method lookup mechanism.

In Groeneveld's version this match did not regard access levels in any way. Protected and public methods were treated identically. However, to protect the programmer, an additional check is required and has indeed been implemented: only a method that is equally or more accessible than the forwarding method may be called on the component. To see why, consider the opposite. In that case a programmer can accidentally call a protected method on a component by calling a public method on a proxy.

Chapter 5

Exceptions

As explained in the section on Delegator's implementation, `Self`'s `invoke` method provides a method lookup mechanism. It does so to dynamically locate a method in `self`'s list of components that matches the call's signature and execute some code for it. In the following this mechanism is extended to deal with java's exception throwing mechanism.

Like many modern programming languages, Java provides extended support for dealing with exceptions. An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Exceptions enable the programmer to separate error handling code from normal code, enable propagation of errors up the stack and allow for grouping and differentiating error types.

How do exceptions work? Programmers may use a `throw` statement to throw an exception somewhere in a method indicating an error has occurred. This means the method creates an exception object and hands it off to the runtime system. After a method has been thrown the runtime system attempts to find something to handle it, moving up the call stack to find an exception handler. This is a block of code, indicating which types of exceptions it can handle and how this is to be done. When a method provides an exception handler for a certain type of exception we say it *catches* that exception.

The class of objects that may be thrown as an exception is called `Throwable`. Two of its descendants, `Error` and `RuntimeException`, form the roots of so called *unchecked exceptions.* In principle unchecked exceptions are thrown by the JVM, as opposed to explicitly in the program code. It is, however, also possible for the programmer to throw unchecked exceptions. All other descendants of the class `Throwable` are called *checked exceptions.*

A method may also specify that it can throw a particular type of exception by making this explicit this in its method declaration. Exceptions that can be thrown by a method are defined to be all exceptions that are either thrown directly in that method by a `throw` statement or are specified to be thrown by one of its called methods. In fact, all methods have to comply to the *Catch or Specify Requirement.* This requirement states that a method must either catch or specify all checked exceptions that can be thrown by that method. In fact, if at runtime a checked exception is thrown by a method that by its declaration isn't allow to do so, the JVM wraps this exception in an unchecked `UndeclaredThrowableException`.

Knowing that methods may specify a number of exception types that that method may throw, how does this relate to overriding or implementing methods? A method that overrides or implements a superclass or interface's method must match with that method's signature. The subclass' or implementing method may never throw more checked exceptions than its superclass or interface. It may however, throw less exceptions or more specific exceptions. To realize why this is so consider the following example:

```
BaseClass base = getBaseClass();
try{
  base.mayThrowBaseException();
} catch (BaseException e) {
  //handling code
}
```

In this example base may either be an instance of `BaseClass` or of one of its subclasses. If `base.mayThrowBaseException()` would throw more than just `BaseException`s this code would break, since the catch clause cannot deal with those. However, if it base throws less exceptions or more specific exceptions the catch clause can deal with all of them.

Above we introduced `self`'s method lookup mechanism. This tries to find a match between the forwarding method that and one of the component's methods. The forwarding method may either be on a proxy, or on a component. Say we already have a method on a component that considering all other things, like name and parameters, matches the forwarding method.

The question is, under which combinations of declared Exceptions for the proxy's and the component's method the latter implements the former. The answer is to simply copy Java's definition for implementation and subclassing, i.e. that there is a match when the component's method does not throw any more exceptions, or more general exceptions than the forwarding method.

With respect to Groenevelds version, a number of things have changed. For the user, the only change is the fact that a check on the above condition has now been added to `Self`'s method forwarding mechanism. In Groenevelds version, the lack of that check led to an `UndeclaredThrowableException`.

On the implementation side this implied that the exception's declarations now had to be copied to both the proxy's and the component's implementations.

Though this new approach is more precise than the old one, it may cause some confusion for the user at times. Consider the case where only the failing of the test on declared exceptions stops a match between a component's and a proxy's method. Since there is no precise match, a `NoSuchMethodException` may be thrown. A comparable situation in regular Java, is when a programmer calls a method on an object via an interface. In that situation, the programmer explicitly sets in code that `MethodClass` implements `MethodInterface`, and the compiler static check fails: because the implementing class throws an `Exception`. However, again we assume an environment that is as supportive of the programmer in the case of runtime errors as it is in the case of compile time errors, i.e. a complete cover of tests using a framework like JUnit. Under that assumption the situation is in no way aggravated.

Finally, there is one more complication. Although it is not required to declare unchecked exceptions in the method header, it is allowed. However, doing this in no way disturbs the method overriding or implementation mechanism. In other words, it is perfectly legal to let a method which explicitly declares to throw a `RuntimeException` implement a method which does not. Again, Delegator has been adapted to behave according to these, admittedly rather bizarre, rules.

Chapter 6

Component encapsulation

Above, both the concept of a component as a part of a composed object, and its implementation in Delegator using the dynamically generated classes that implement the token interface `Component` were introduced. In this chapter the reader is reminded of the distinction between the component as a concept and a `Component` as an implementation detail. The use or lack of use of `code` font will indicate either.

The main problem of this chapter is to maintain encapsulation of the `Component`. Why this is required will be reminded first. Secondly it will be shown that references to a components are very desirable, and how this can be combined with the encapsulation of the `Component`. Finally, and mainly, a problem in Groeneveld's version will be exposed, and how the current version of Delegator deals with this problem.

In the explanation of Delegator's solution to the self problem, we made one key assumption: `Components` can not be accessed from outside `self`, they are shielded from the user. Firstly, the reader is reminded that though a `Component` is a subclass of its model class, all its methods are implemented as forwarding methods to some `self`. Building on the assumption of `Component` encapsulation, a stack of instances of `Self` is maintained in the method `Self.invoke`. The top of this stack is used in the forwarding methods of the `Component` as the client `self` that should be forwarded to. For now, it suffices to stress that the encapsulation of `Components` is key in Groeneveld's implementation.

References to components

In the above it is shown that instances of `Component` must remain encapsulated. On the other hand, it is very desirable to have access to the concept of a component. One of the advantages put forward in the introduction and one of the goals of Delegator is the fact that the composition of `self` may be manipulated. Specific components may be added, removed and even shared among different selves. To do so, some kind of reference to the components is required. Summarizing it can be said that though references to components are desirable, a regular Java reference to a `Component` is very undesirable because of `Component` encapsulation.

Groeneveld's version did not meet these dual requirements. To enable direct manipulation of the `Component`, the user had multiple available paths. Firstly she could directly access the method

```
ProxyGenerator.newComponentInstance()
```

that returned a new `Component`, secondly she could retrieve them by index from `self`. The current version of Delegator no longer allows for such direct access to Components. Instead, if a single component is retrieved from `self` it is wrapped in a new instance of `Self`. This can either be used as an object composed out of one component, or it can be added to another `self`. The latter then adds the former's single `Component` to itself. This solution allows for manipulation of single conceptual components and the calling of their methods without violating the encapsulation of the implementation detail `Component`.

Leaking this pointers

The second, and much more complex, issue that surrounds the encapsulation of `Components` are *leaking this pointers.* In Groeneveld's implementation of Delegator it is conceivable that a reference to a `Component` may be acquired outside `self`, by accident. This may happen when a reference in the `Component`'s code to itself, using the keyword `this`, is passed to the outside world. This may happen either because it is passed as a parameter to a method of another object, or because it is returned directly as the result of a method.

In the original model class the explicit use of the pointer `this` simply refers to an instance of that class. Passing a reference to that class outside of it is in no way illegal. However, since the `Components` is a dynamic subclass of the model class, the reference `this` refers to an instance of `Component` in that context. The assumption of encapsulation is violated if such a reference is passed to any other place than the `Component` itself or the `Self` code.

This violation is indeed catastrophic to Groeneveld's implementation. Since a `Component` is a subclass of its model class, and in passing the reference it is indeed treated as such, it may be expected that methods are called on it from outside. However, all of the `Components` methods are overridden to be forwarding methods that depend on the manipulation of the stack of instances of `Self`. The effects of calling a method on a `Component` without doing that manipulation are random.

To see how this actually leads to problems in practice, consider the example of the factory pattern. Say there is some class `Document` that maintains a number of nodes in a graph. Instances of `Document` act as factories to generate these nodes, in such a way that the nodes are instantiated with a reference to the `Document` they belong to. Note the use of `this`, that becomes problematic if the `Document` is used in a Delegator context.

```
public class Document {

  public Node createNode() {
    return new Node(this);
  }

}
```

Groenevelds workaround solution

Groeneveld's version of Delegator did provide a workaround solution for this problem. It does this in the form of a required, but minimal adaptation of the `Component`'s model class' source code.

The model class' source code can be executed either as part of a `Component` or regularly. In the former case it is desirable that instead of a pointer to `this`, something is either passed or returned that doesn't break the code. In the latter case, modifications in the behavior are undesirable. `Self` provides a method that does exactly this. For instances of `Component` it returns a proxy to the right instance of `Self`. For other objects it returns that object. The workaround is to replace the relevant instances of `this` by `Self.self(this)`.

It can be assumed that the amount of cases in which `this` is actually returned or passed as a parameter in code that ends up in a `Component` is relatively small. Under this assumption, and since the workaround is limited to a 11 character adaptation per instance, we feel it is very justifiable.

However, if it is not possible to alter the existing code, this solution is not workable. Since the reuse of existing code was formulated as one of

Delegator's main advantages, the lack of a solution that keeps the existing code intact is problematic. In the following a number of possible alternatives will be explored.

Copying byte code

Each `Component` contains, for each method it implements, both an implementing method and a forwarding method. In the implementation of Delegator explained in the above, the implementing methods simply call its superclass' equivalent, leading to the problem described. Is it possible to modify the implementing methods in such a way, that instances of `this` are replaced by instances of a self pointer? Two answer this question, two approaches will be examined in particular.

Local variable 0

In Java, the pointer to `this` is simply stored as the first local variable of a method, i.e. it can retrieved with the instruction ALOAD0. Does storing a new value, i.e. some reference to `self`, in this field solve the problem? It may seem likely, since if it would prove possible any reference to `this` leaking out would then really be this reference to `self`. In pseudo Java, i.e. Java bytecode rewritten as Java, the implementing methods are adapted as such for this purpose:

```
public void __supercall_someMethod() {
  //fetch self from the stack
  this = self.cast(SomeClass.class);
  super.someMethod();
}
```

However, for each method call, also for method calls on the same object including method calls using `super`, a new local variable stack is created. This means that in the context of `super.someMethod` the value of the first local variable is set back to a reference to `this`.

Copying byte code

A second approach is to replace the implementing methods by a complete copy of the byte code of their supermethods. This code can then be freely modified in such a way that the proper instances of `this` are replaced by equivalent references to `self`. Before we will show the ultimate limitation of this approach, let's examine some of its consequences.

Firstly, the code of the method may not be available in the direct superclass of the `Component`. Therefore, the entire inheritance chain must be examined and copied. However, since references from the superclass' methods to private fields or methods of that class cannot be used, all of these must be copied to the `Component` class as well. The resulting class is a huge collection of all superclasses' bytecode. Secondly, the actual replacement of `this` pointers isn't trivial either. Since only the pointers to `this` that are passed out need to be replaced, a full analysis of the bytecode of the method is required. A number of further complications will be left out here for reasons of brevity.

The main problem, however, is the combination of copying code from superclasses and Java's access control. Consider all issues with Java's access control seen above. Now consider that code is being copied from some original class to a new class. Any of the issues are now present not just in the case of proxies, but in the actual implementation of the component. For example, consider the case of referring to another object in the same package using a package accessor, anywhere in the component's code. Such a reference is illegal if it's copied to a new class and then reloaded using a different `ClassLoader`.

It can be assumed that the number of instances in which pointers to a `Component` will leak in practice is relatively small. The number of references to private or package methods anywhere within a component's code will indeed be many times larger. A single reference of that kind renders this approach useless. Therefore, on balance, this approach is less attractive than Groeneveld's.

Unsharable components

As we have seen, preventing pointers to a `Component` to leak is a very hard problem. Remember that the only reason to do so in the first place is to enable a solution for the self problem in a context of sharable components. Let's, for now, consider the possibility of solving the self problem using components, but dropping the goal of being able to share components.

If the requirement of being able to share components is dropped each `Component` can be equipped with a field that refers to the single instance of `Self` it belongs to. This is exactly the implementation that was chosen for proxies. In this approach, forwarding methods simply delegate to this single instance. So, components are simply proxies that also implement behavior.

This alone is enough to solve the problem. Admittedly, components may still leak out. However, since components are simply extended proxies, calling methods on them is safe. This solves the second problem of this chapter.

However, this solution doesn't combine with the ability to share components among different instances of `Self`. In fact, no such solution has been found. Therefore, the question arises what is more important: to be able to share components or to have a safe implementation.

Delegation has a large number of very dissimilar advantages. In practice users will rarely use all of these at the same time. It is hard to say which feature is more important without any knowledge of the user's particular system. Therefore, the latest version of Delegator offers its users a choice. It can either create components as sharable instances, in which case it is up to check for leaking `this` pointers, or as non sharable instances.

The default implementation hereby is to use non sharable components, meaning that the user must explicitly request sharable components. The number of cases in which either implementation is required is expected to be roughly equal. However, sharable components fail much less elegantly than their non sharable counterparts. The former may generate errors at random places in the program, as shown above. Non sharable components will simply throw a runtime exception when being added to a second `self`. This exception can even refer to the necessary use of a sharable component. Additionally, it is at the moment of "upgrading" to a sharable component that the documentation on their particular pitfalls is most likely actually read by the user. The only situation that isn't dealt with by this combined solution is unmodifyable code which passes `this` pointers around and must be used in multiple instances of `Self`.

Chapter 7

Concurrency

One of the many features of the Java language is that it supports concurrency. A Java program may contain a number of independently running *threads*. One of Delegator's main goals is to be transparent, meaning that, as much as possible, anything that can be done in regular Java can done in Delegator in the same way. In this chapter we will examine how Delegator has been made transparent with respect to concurrency.

Regarding support for concurrency in Delegator there are two distinguishable steps. The first step is to ensure that Delegator keeps functioning, even if multiple threads are in one composed object's code at the same time. The second step is to provide an analogy for Java's locking mechanism in the context of delegation.

Before delving into the specific situation of Delegator, let's take a short look at some research on the topic. The combination of concurrency and delegation has been researched by various authors. A language that supports delegation for a distributed environment is dPico. It additionally describes a number of concurrency issues and proposes another language on the way that deals with concurrency, cPico. The approach the authors take, however, is to base cPico on Active Objects, an approach that relates to Java's thread paradigm only in a very limited way.

The language Self supports concurrency on a very basic level. New processes can be created and semaphores used to denote critical sections. dSelf is a language, based on Self, that supports distribution. The mechanisms for concurrency provided by dSelf are quite limited: the idea was to implement the issues concerning distribution before the issues concerning concurrency were solved. Now that Self is no longer maintained, this seems unlikely. Concluding, the research on the combination of Java's thread paradigm and delegation is very limited.

A stable Self

Looking at the combination of Java's thread model and Delegator, we will firstly examine the use of one `self` by multiple threads at the same time. Remember that Delegator allows for dynamic composition of objects. This implies that a particular composed object `self` may at some points in the program respond to any method call `m()`, while at other points it may not. Additionally the response to the method may vary from time to time.

Looking at this in the light of multiple threads, one thread may be manipulating the composition of an object while another calls a method on it simultaneously. We put the responsibility for checking whether a called method actually exists squarely on the shoulders of the programmer, even in a context of multiple threads. Consider the scenario of manipulating an object and calling a method on it from different threads again. In this case the thread calling the method `m()` will have to wait until that method is available before actually calling it. Moreover, it should acquire some kind of lock to make sure `self` is not manipulated between the check and the actual call.

However, there are also scenarios in which the programmer has no such responsibility. Consider the case where a method `m()` is called on a composed object. The component `c2` of this object has an implementation for this method. However, at the same time, a component `c1` is removed or inserted. If `c1` contains no method `m()`, the method call can still be dealt with by the component `c2`, irrespectively of the simultaneous manipulation of the composed object. If `c1` does contain a method `m()`, the method call should be dealt with by exactly one of the components `c1` or `c2`.

Exactly this behavior was not implemented in Groeneveld's version. Note that in the context of delegation a method call is by no means a single operation by definition. Rather, it implies a lookup in the list of components of a composed object `self`. However, if this list is simultaneously manipulated by another thread, components may be either overlooked or the mechanism may look for components that no longer are part of the list.

So, the method lookup mechanism may never be executed simultaneously with any manipulation of the component list. This can be achieved by creating a monitor, using the Java keyword `synchronized` for both the method lookup and any manipulation. Any thread trying to do either will then first make sure no other is doing the same.

Monitors

The second addition to Delegator is the introduction of monitors in the context of delegation. Hoare introduced the monitor as a concept to prevent simultaneous data access in [18]. A monitor is a combination of data and procedures on that data. At any given point in time there is no more than one thread executing the monitor's code.

Java's concurrency support owes a lot to monitors. Java's objects map quite naturally to the idea of a monitor: objects too contain both data and methods. Blocks of statements, methods and classes may be declared to be `synchronized`. A statement block is synchronized by acquiring a lock on some object:

```
public void test() {
  synchronized(someObject) {
    //... critical section
  }
}
```

Any thread attempting to enter the synchronized block will first have to acquire a lock on `someObject`. If another object already has this lock, the thread waits until this other object releases the lock. One thread may recursively obtain the same lock multiple times, but multiple threads may never obtain the same lock simultaneously. Synchronized methods are equivilent to synchronized statement blocks that obtain a lock on `this`. Synchronized classes contain only synchronized methods.

Considering Hoare's model, the set of synchronized statements for one particular reference form a monitor together. In Java this reference is most often the reference of the object `this`. In an inheritance based language such as Java there is a natural one to one mapping from objects to monitors. Every object is clearly identifiable and separate entity having it own data and modifiers. In a delegation based language this separation is less clear. Objects that are composed using delegation may have a number of components and components may be shared across different instances of `Self`. So, firstly, conceptually the delegation model maps less clearly to monitors than a model of clearly separated classes.

Secondly, note that with respect to implementation a new issue arises. In Java's context of class based inheritance acquiring locks on `this` makes a lot of sense. All methods in the class hierarchy that are synchronized, and all statement blocks that are synchronized using `this`, acquire and release exactly the same lock. This means the monitor may span multiple levels of the class hierarchy.

Mimicking class based inheritance using delegation this mechanism fails. In any setup using delegation the components are separate entities, though they are connected by one `self`. If distinct components, however, each have their own set of synchronized methods, each of these will form a monitor of its own. Of course, if delegation is being used to mimic class based inheritance, this is undesirable.

It is very tempting to look for a solution for this problem in `self`. Is it possible to somehow regenerate the component's synchronization clauses in such a way that locks are acquired on the right instance of self? Alternatively, is it possible to redesign `self` in such a way that it acts as a single monitor where desirable?

Unfortunately, this is not the case, since there is no one to one mapping from composed objects to monitors. In fact, monitors may either span composed objects, composed objects may contain multiple monitors or both. Since the concept of a monitor can only be distilled by knowing what the code does, this responsibility will be left to the programmer.

However, having seen that Java's mechanisms aren't sufficient to ensure locking in the context of delegation, two similar solutions will be offered to the user to do. Because they differ slightly and have distinct advantages both are offered as part of the Delegator package. Though they differ, they are also similar. Both approaches offer an object which may be added to `self` to uniquely represent a lock for the critical sections. It is left to the user to make explicit references to this object.

In the first approach we introduce two methods, `acquire()` and `release()`, which respectively acquire and release a unique lock. The methods themselves are also synchronized, to assure that they can only be accessed by one thread at the same time. If the lock is free `acquire()` will give the calling thread the lock. If the lock is not free the calling thread will wait until it is released by the `release()` method.

Java provides a built in class `java.util.concurrent.Semaphore` that provides this behavior. However, one important feature is missing. As shown above Java's built in synchronization allows for nested calls. Unfortunately, the built in class `Semaphore` does not, meaning that acquiring the same lock twice, as in any recursive call, leads to a deadlock. To amend this Delegator provides a class `Semaphore` of its own.

Obviously, instances of this class can be combined with the rest of our code using delegation. The use of the Semaphore is demonstrated below:

```
public class Counter implements ISemaphore

  public int inc() throws InterruptedException {
    acquire();
    int result = value++;
    release();
    return result;
  }

}
```

This approach is very flexible and allows for a high level of programmer control. The semaphore may be either linked in using delegation as shown above, or explicit links may be made to a separate instance. Locks may be acquired and released at any location in the code, so critical sections may span methods or even objects.

On the other hand, the syntax is rather bulky. The equivalent of the above example without locks is `return value++`. Firstly, the user needs to add the two statements `acquire()` and `release()` for each critical section. In the case of `return` statements an additional local variable is needed because `return` statement must be the last statement of the method.

The second approach uses the built in `synchronized` keyword for statement blocks, and provides a distinct object that can be used as the lock. Above it was made clear that using `this` as the lock in a `synchronized` statement is often not a workable solution in the context of delegation. However, using any other object may be such a solution.

To do so we simply provide a class `Monitor` with one method `getMonitor()` returning the monitor itself. We also provide an interface `IMonitor`, implemented by `Monitor`, with this one method. Blocks that need to be synchronized can now be wrapped by synchronization statements like this:

```
public class Counter implements IMonitor {

  public int inc() {
    synchronized(getMonitor()) {
      return value++;
    }
  }

}
```

Note, by the way, that this solution amounts to exposure of a component, since the code of `getMonitor()` is a single `return this` statement. However, since no further methods are called on the monitor and only its reference is used, it can be implemented either as a sharable or a non sharable component.

Concluding, in comparison to Java's notation for synchronization, which also allows for synchronized methods, these two solutions are slightly less elegant. However, we feel that it is a very workable solution indeed.

Chapter 8

Performance and redesign

In this chapter we turn our attention to the performance characteristics of Delegator. It is the goal of Delegator to be actually useable in practical applications. For it to be acceptable for any professional business application any framework needs to have *known* performance properties. Secondly these properties need to be within reasonable limits: if Delegation is too slow to be workable it is not going to be used, no matter how elegant it may be.

Before this research the performance characteristics of Delegator were unknown, which was a drawback by itself. Since almost no effort had been put into performance - in fact only one optimization had been implemented - the expected performance was furthermore very poor.

In this chapter we will firstly look at the methodology that was used for benchmarking and improving performance. Secondly a number of categories of performance improvements that have been used will be explained. Firstly we will look at an overhaul of the entire design to adapt to one particular optimization and facilitate further optimizations, and secondly at a number of different smaller improvements. Finally some remarks about the general application of these optimizations in other areas will be made.

Methodology and starting point

Our methodology and initial performance will be presented before the results of the optimization process. We have used two tools to measure and support our performance progress: a benchmarking tool and a profiler. The former compares a large number of operations in Delegator with an equal number of equivalents in regular Java. The latter was used to find out which parts of our API were using most processor time.

Let's examine the benchmarking tool in some more detail. This tool was built specifically for the purpose of improving Delegator's performance.

However, though very important for the process, its implementation is simple.

The basic principle is two run two loops and compare the results. In the first loop one of Delegator's operations is executed a large number, say 100.000 or 1.000.000, times. In the second loop the same is done for a Java equivalent. In both cases Java's `System.currentTimeMillis()` built in functionality is used to determine the time before and after the loop. Delegator's runtime is then divided by the Java equivalent's runtime to determine the ratio between the two.

To get more stable results we have extended this simple method with some frills. Most Java Virtual Machines have some kind of dynamic optimization going on behind the scenes. These are usually triggered by large numbers of executions of the same code. Since these optimizations may only be activated after a large number of runs, they could skew our results. We have remedied this by running each of the loops once directly before the actual measurement, to trigger the optimization. Secondly the results may be influenced of other processes by their usage of processor power. A first remedy is, of course, to kill as many of these as possible. Secondly each of the loops is run a number of times, say five or ten, to take into account and visualize fluctuations. Of these 5 loops we display the average runtime, minimal runtime and maximal runtime. If minimal and maximal runtime vary greatly this indicates some disturbance and the need for a rerun. A final disturbance may be the loops themselves. However, since these take only a tenth of even the simplest Java instruction's time, these have been neglected.

Having a simple framework, it must be determined which things to measure. We have focused mainly on processor performance, as opposed to memory overhead. The reason is that the memory footprint of Delegator was acceptable to begin with, and has not been increased significantly. In fact, the memory footprint of a `self` with a single `Object` component is about 5 times as large as that `Object` alone. If this difference seems large, remember that the comparison is made with the smallest of all possible objects.

In measuring performance we have considered a number of important operations from the Delegator framework. We have looked at method calls, object creation times, casting and changes in the composition of components. Method calls have been examined in the greatest detail, and a number of different kinds of method calls has been tested. The simplest form is a method without any parameters or return value. Methods containing those, and exceptions, have been tested separately. Finally we have examined a method call that resolves to the second component. In Delegator we can compose `self` out of a number of components. In the process of method resolution these components are checked in sequence for a matching method. Since this checking is done in sequence a `self` with a matching method in the second component took about twice as long to resolve. All of these method calls have been compared with their equivalent in Java, meaning that a call to a method with three Object parameters using Delegator has been compared to a call to a method with three Object parameters without delegation.

Another area of study were creation times; the creation of new instances of `Self` containing a single component as compared to the creation of that component alone. We examined the time it took to switch between different compositions of objects, i.e. to remove and add components to `self`. Finally we measured the time casting using the special method `cast` took.

These measurements have been compared to the simplest possible counterparts in normal Java. For example, for method calls methods without any implementation have been used. In reality, with the implementation of the method taking most of the time, the constant overhead will be much lower in comparison. In many cases the average expected ratios for Delegator will therefore be much more favorable than the numbers presented here may suggest.

An evaluation of Delegator's performance on small numbers of operations, as opposed to the very large numbers mentioned before, has not been done for a number of reasons. Firstly doing performance evaluation on small numbers is very hard because the setup costs and variations are relatively high, so measuring becomes very hard. More importantly, the scenario of high numbers is the only relevant scenario to begin with. Even before any optimizations an old 500 mhz pentium III could do 10.000 delegator method calls in one second. Doing optimizations for single calls on this scale is not very interesting.

Before any optimizations calling methods turned out to be roughly 2000 - 8000 times more slow than doing a method call in normal Java. Since method resolution took most of the time in the initial setup, the differences between method calls with or without parameters or exceptions were minimal. A major difference was caused by the location of the called method, meaning that methods that were easy to find resulted in lower execution times. Creation of a new `self` with a single component was roughly 30 times more slow than its counterpart of creating the component alone. The method `cast`, creating a new proxy, was roughly 500 times more slow then the creation of a new object.

Once a benchmarking tool was in place, performance was improved step by step using a well established process. Every iteration starts with by running the benchmarking tool, of which the results are saved. Secondly the profiling is run which indicates which areas are considerably more slow than others. In these areas some improvement is then implemented. Finally the benchmarking tool is used again to see whether this theoretical improvement also amounts to a real speedup. If it does, keep the improvements, otherwise try something else.

This way of working leads to large optimizations first, and smaller optimizations later. In the following, the optimizations aren't described in the order they were actually implemented.

In the end a number of accumulated improvements led to the evolution of the code base to a new design. For reasons of clarity this new design will be introduced first. Its implementation opened up the way for a final improvement in speed: the removal of some particularly slow uses of the Java `reflect` framework and their replacement by even more dynamically generated code. This will be dealt with secondly. Finally we will deal with a small number of miscellaneous improvements.

Redesign

In speeding up Delegator's method calls the first and largest improvement was to introduce a cache for the called methods. In explaining this a more fundamental view of what has been done so far and how this could lead to a redesign will be given first. That this redesign has the added benefit of a performance improvement will be explained second.

Regular Java has the concepts of classes and objects. Extended with the concept of delegation, objects can also be composed out of other objects. We have used the term `self` to denote such a composed object. Analogously to regular Java it can be said that the composed objects are instances of some composed class. Whereas a composed object maintains an ordered list of components, its composed class maintains an ordered collection of classes.

The methods an object responds to, and how, are uniquely described by its class. Therefore the methods a composed object responds to, and how, are uniquely described by its ordered collection of classes. Concluding, each time a method is called on a specific composed object it is resolved to the same component. However, recall that in the implementation of Delegator presented above, method lookup was dealt with at the level of `Self` by the method `invoke`. Following the analysis presented here, we introduce a new class `ComposedClass`. It has two related responsibilities: firstly to resolve method calls to specific components and do related caching, and secondly to maintain its own composition.

By moving the resolving of methods to the level of a composed class that algorithm itself doesn't change fundamentally. Objects of class `Self` maintain a reference to an instance of their `ComposedClass`. The method `invoke` is no longer called on `self` directly, but rather on its composed class with the instance `self` as a parameter. Since all composed objects of the same composed class may share a single instance of that class that instance doubles as a starting point for caching.

Each time the method `invoke` is called on the composed class with a particular forwarding method as a parameter, the composed class first checks whether the method is already resolved. If this is the case it is invoked immediately. Otherwise it is resolved and the result is stored. The method used to store it will be described later, since it is itself further optimized.

Note that since the method `invoke` has been moved and its signature changed it longer implements the `InvocationHandler` interface. Since proxies for interfaces are generated using the `java.reflect` framework these no longer work. Therefore, we have chosen to generate proxies for interfaces ourselves too.

Let's secondly look at how a `ComposedClass` knows what kind of composed class it is. To do so an instance it maintains an ordered list of `Class` objects. This list is used in the process of resolving methods. In this process the composed class simply checks each of its contained classes in order for a match.

Changing the composition of a composed object, and thereby changing its composed class, has also been optimized. Though dynamic component composition may seem one of Delegator's more advanced features it is heavily used, making for an excellent candidate for optimization. In fact, changing composed class happens by default on most object constructions: in that case there is a switch from the empty composed class to a specific composed class. To support this feature `ComposedClass` offers a number of methods for obtaining its neighbors, meaning any instance that can be reached in exactly one step by either adding, inserting or removing a specific class. Since these are exactly the operations that can be performed on composed objects and, in the case of adding, are performed at object creation, optimizing them is worthwhile.

This optimization is implemented using a dictionary for each of these three operations, with the parameter of the operation as its key and the resulting `ComposedClass` as its value. For example, in the case of adding, the added class serves as a key and a composed class with the extra class as its value. These data structures are lazily initialized, so if from a particular `ComposedClass` nothing is ever removed, there is no dictionary created for removing from it. If a particular neighbor is requested from a `ComposedClass` which is stored in its cache that neighbor is naturally returned. If no such neighbor is present in the cache yet, it is looked up in a slower dictionary containing all instances of `ComposedClass`. If it is not present there it is created and added to the dictionary. A link to the empty composed class is maintained separately.

Let's look at a single example to clarify this. To obtain the ComposedClass representing a `Vector` and a `HashMap` in that order we can call.

```
ComposedClass composedClass = ComposedClass.getEmptyClass().
  add(Vector.class).add(HashMap.class);
```

Of course, in actual use the composed classes are mostly obtained from `Self`, each time the composition of a composed object is changed. The speed with which components can change composition, that was about 10 times as slow as object creation with a naive dictionary implementation, is now exactly as fast as one object creation.

The caching of methods to prevent recurring method lookups proved to be a major improvement. After implementing it method calls were roughly 100 times more slow than their counterparts in Java. In other words, it represents an improvement of a factor 20 to 80.

Regenerating Java Reflection

Delegator's performance was further improved by changing the implementation of invoking methods. In the implementation described above, methods are passed around as instances of the `java.reflect.Method`. Obtaining, manipulating and invoking these is extraordinarily slow. Let's first examine their use in Groeneveld's implementation in some more detail, zooming in on the forwarding method of a proxy or component and on the `invoke` method of `ComposedClass`. In the following, for reasons of brevity and without loss of generality, it will be assumed that the forwarding method is part of a proxy or non sharable component and has `self` as a field.

Firstly, in the forwarding method, a matching instance of a `Method` is fetched. Secondly, the parameters of the method need to be wrapped in such a way that they can be passed to the method `invoke`. After returning from the method the result may be unwrapped before it is returned. Consider this example in generated Java.

```
//forwarding method
public int lastIndexOf(Object elem, int index) {
  Method forwardedMethod = getClass().getSuperclass().getMethod(
    "lastIndexOf", new Class[]{Object.class, Integer.TYPE});
  return ((Integer) self.composedClass.invoke(self, forwardedMethod,
    new Object[]{elem, new Integer(index)})).intValue();
}
```

The `ComposedClass` `invoke` method first looks up this `forwardedMethod` in its cache or component list, returning an instance of `Method` for the implementing method. This method is then invoked on the matching component using its own `invoke` method and the argument list:

```
  implementingMethod.invoke(component, args);
```

Besides that, `ComposedClass`'s `invoke` deals with any uncaught exceptions and manages the stack of instances of `Self`.

This setup contains a number of consecutive and unnecessary bottlenecks. Firstly acquiring the `forwardingMethod` using `getMethod()` is slow, more so since this is done for each call and the associated array of classes is generated for each of these. Secondly, in the `ComposedClass` it is this `forwardingMethod` that is used as a key for each lookup in the cache. Comparing instances of `Method` using the method `equals` involves comparing a large number of fields for equality. Thirdly, wrapping and unwrapping the parameters and result involve further waste, especially since another array is generated on the fly and casting is involved. Finally the method `invoke` on the `implementingMethod` itself involves extra overhead.[1] In the following all of these bottlenecks will be addressed.

Indexing forwardingMethod

The first two bottlenecks will be dealt with first. Firstly note that for each implementing method the result of the call to

```
getClass().getSuperclass().getMethod
```

is constant. Note furthermore that the resulting `forwardingMethod` is used in one of two ways. Firstly its precise description is necessary to find a matching implementing method in one of the components. This is done only once per combination of composed class and method, since the result of this lookup is cached. Secondly it is used as the key of exactly that cache. So, in the majority of cases, the only point in time the variable `forwardingMethod` is read is to compare it, using a large number of fields, to a stored key.

[1]In fact one major bottleneck is not mentioned here. In Groeneveld's version, implementing methods and forwarding methods were not distinguished by adding Self as an extra parameter, as opposed to simply by prefixing the implementing method's name with a string such as "supercall" as described in the introduction. Though this parameter wasn't used anywhere, adding it to both the list of classes and the list of arguments involved further costs.

Since the we control both the generating of the implementing method and the implementation of the cache, we have chosen a cheaper alternative. To do so we first introduce the concept of a *method register*, implemented by the class `MethodRegister`. The singleton method register contains methods of the type `Method`, and associated indices of the type `int`. Furthermore it offers methods to retrieve methods by index and vice versa. Both methods and indexes are unique. Finally it is lazy, meaning that a method is only stored with an associated index if such an index is requested by one of its clients.

Next, the call to `getClass().getSuperclass().getMethod` is moved to the moment a forwarding method is generated. At this point the method register is asked for a unique index, and this unique index is included in the byte code, resulting in the following example code:

```
//forwarding method
public int lastIndexOf(Object elem, int index) {
  return ((Integer) self.composedClass.invoke(self,
    23, new Object[]{elem, new Integer(index)})).intValue();
}
```

Note that only methods that are contained in any of the components or proxies are contained in the method register, so no unnecessary overhead occurs.

On the side of `self` and its associated `ComposedClass` these changes are mirrored. If for a particular index the associated `Method` is needed to do a method lookup, it is fetched from the method register. This happens only once for each combination of a method and a composed class. If the result of this lookup is already stored, which is the vast majority of cases, the index is simply used as an index to an array of implementing methods. Using these, and associated improvements, method calls were roughly 20 times more slow than their counterparts in Java. However, the exact result depends greatly on the number of parameters and return type of the method.

Replacing Method by bytecode

The last remaining bottleneck is that the `java.reflect` framework itself is slow, and that this is aggravated by the wrapping of the parameters and the unwrapping of the result. We have therefore, decided to generate our own dynamically adapted equivalents of`Method` for each method. This will do away with the need for casting both the return values and the wrapping of parameters.

To do so, code will be generated that calls the target method of the target component directly. Regarding the running example of this chapter we propose the following ideal situation:

```
//forwarding method
public int lastIndexOf(Object elem, int index) {
  //obtain the right componentIndex from self's ComposedClass
  return ((Vector$component) self.components[componentIndex]).
    __supercall_lastIndexOf(elem, index);
}
```

However, the class of the target component is not known at the creation time of the forwarding method as is the assumption of the above code. Moreover, the Java byte code instruction to cast to another class uses a static link to the class that is being cast to. In other words, it is not simply possible to dynamically change this cast after the code has been generated. Concluding, it is not generally possible to generate the above code. A workaround for this is the final subject of this chapter.

We propose the following skeleton for the running example:

```
//forwarding method
public int lastIndexOf(Object elem, int index) {
  return [...] .invoke(self, elem, index);
}
```

We will first fill in the dots and then adapt the rest of the code accordingly. What is it the code on the dots should do? Note that with respect to the previous working version of the example a number of things are missing. Firstly, of course, the cast to an `(Integer)` and its associated `intValue()` call, and the creation of the parameter array `new Object[]{... new Integer(...)}`. The fact that these are gone is the purpose of the final optimization, so they should *not* be recreated on the dots. However, the path to `self.composedClass`, and the forwarding method index `23` are also missing, and they both contain required information. We therefore adapt the code as such:

```
//forwarding method
public int lastIndexOf(Object elem, int index) {
  return ((ForwardingMethod_23) self.composedClass.getMethod(23)).
    invoke(self, elem, index);
}
```

The class `ForwardingMethod_23` is simply an abstract class that has a single method `invoke` with as parameters a `Self`, an `Object` and an `int` and returns an `int`. Note again that this is really all the information that is available at the moment this code is generated, since the class of the implementing method may be anything.

```
public abstract class ForwardingMethod_23 extends ForwardingMethod {

  public abstract int invoke(Self self, Object elem, int index);

}
```

We now turn our attention to `ComposedClass`. Above implementation the method `getMethod()` on that class returned a `Method`. In the adapted version it returns an instance of `ForwardingMethod_[methodIndex]` that is extended to contain a call to the corresponding method of the right component. To construct that call, the index and the class of the component must be known, so that it can be looked up in `self` and cast.

How is such a `ForwardingMethod_[methodIndex]` obtained? A successful lookup of the forwarding method in the composed class will result in both the index and the class of the component implementing the method. After that a class `ImplementingMethod_[methodIndex]_[componentClassIndex]` is generated dynamically to include the cast to the component class. An instance of this class is then created to include the component index.

To clarify this, consider a `ComposedClass` that is composed of a `HashMap$component` and a `Vector$component`, in that order. The method lookup of method 23 will result in the method `__supercall_lastIndexOf(elem, index)` of the second component, since `HashMap$component` has no such method and `Vector$component` has. This component has class `Vector$component`.

Using this information, the class

```
ImplementingMethod_[methodIndex]_[componentClassIndex]
```

is created dynamically. Hereby, for `[methodIndex]` the index of the forwarding method (23) is substituted. For `[componentClassIndex]` an additional lookup is required. To that end, all component classes are stored in a register to provide a unique number per component class, just like forwarding methods. If the class `Vector$component` has an index of 4, this will result in the following generated Java:

```
public abstract class ImplementingMethod_23_4
                 extends ForwardingMethod_23 {
```

```
  public int componentIndex;

  public int invoke(Self self, Object elem, int index) {
    return ((Vector$component) self.components[componentIndex]).
      __supercall_lastIndexOf(elem, index);
  }

}
```

An instance of this class is created in the method `getMethod()` and its field is set to 1 (like anything in Java components are stored 0-based). Note that this is exactly the desired ideal code proposed at the beginning of this section.

The details are left to the imagination of the reader. It suffices to say that separate generators for the different replacements of `Method` are required, that the entire setup is extended with a number of caches for dynamically generated classes and instances and that existing code is adapted to work with the above set of classes where necessary.

This final improvement has brought Delegator's method invocation up to one tenth of the speed of a regular method invocation. Especially with regards to methods that had large parameter arrays this amounts to a large step. Remember, though, that these are worst times, because they amount to empty method calls being prepared. We assume that in regular use empty method calls are very unlikely and therefore that the average times will be much better.

Finally, we think the setup of replacing Methods by bytecode provides an interesting perspective on improving the speed of the `java.reflect` framework, also outside the context of this thesis. It falls outside the scope of this research to generalize the approach, though.

Results

A number of optimizations have been explained in some detail here. However, it must be noted that a large part of this work was not so glamorous as to deserve an explanation in the above. For example, note that no mention has been made of speeding up proxy creation times, though a great speedup has been achieved there. A part of these unexplained improvements are due to the fact that parts of Delegator use other parts, and that therefore improvement in one area may lead to improvement in others. For the remainder, performance was simply improved by even more caches, often in the form of local variables or fields.

Method calls have been speeded up from 2000 - 8000 to 10 times more slow than doing a method call in normal Java, which amounts to a factor 200 - 800 speedup. Creation of new composed objects has been speeded up from 30 times its equivalent to 7 times, which amounts to a factor 4. Though this result may seem less spectacular than its counterpart for methods, it must be remembered that there really isn't much going on for composed object creation in the first place. Finally, creating new proxies using the method `cast` has been speeded up from 500 to 16 times the creation time of a java `Object`, amounting to a factor 30.

Chapter 9

Miscellaneous features

In the process of creating and using the Delegator API a number of issues arose. Some of these presented themselves as problems when testing, others sprang up as new ideas building further on the possibilities that Delegator already presented, yet others followed from trying to complete the analysis in this thesis. All of these features are important additions to the framework from perspectives of usability and from the perspective of providing a full analysis of what delegation is about and which features are useful. Implementation, however, was often very straightforward, following directly from the description of the feature. Therefore, some of these topics are much more briefly described than others in the same thesis.

In this chapter, we will first present a new way to create components and do an analysis of its pros and cons. Secondly, we will see how Delegator deals with Java's standard object methods, such as `toString()` and `equals()`. Thirdly we will look at some methods on `Self` that could prove to be useful for all its subclasses.

Automatic forwarding

In this section we will introduce a lesser form of delegation and its implementation in Delegator. We will contend however, that this form may firstly prove more intuitive to learn for new users and secondly offers a number of options delegator doesn't.

As shown in the introduction, the word delegation may be used for a number of rather different concepts. To qualify for true delegation as used in Delegator we've imposed two conditions. Firstly delegation implies automatic message forwarding and secondly the self problem is solved.

Groeneveld's version of Delegator only offers true delegation. As seen in

Darwin/Lava, we may also offer the user the option of automatic forwarding. In automatic forwarding, `self` is still composed out of components. However, whenever a method is called on one of these, further method calls will be handled by that component alone, not by the client `self`.

The implementation of automatic forwarding is straightforward, since it mainly consists of offering *less* to the user. To implement delegation without a solution for the self problem, we remove the part that solved that problem. This part is, of course, the automatic generating of components. So, in the context of automatic forwarding we simply add an existing object to `self` as a component. In a number of areas we also modify code to make sure that, in the case of this kind of component, no attempt is made to call the implementing methods by using the special prefix `__supercall_`.

Offering the use of automatic forwarding next to true delegation has a number of benefits. Firstly for a great number of users there may be no need for true delegation at all, whereas automatic forwarding would be a feature of great help. Automatic message forwarding requires a lower level of understanding of the concepts of delegation and is closer to other concepts from object oriented programming and well known design patterns. Secondly automatic forwarding is much faster than true delegation, because the overhead of redirecting every method call within the component via `self` is eliminated.

Thirdly automatic forwarding provides a way around a limitation of Delegator. Because Delegator creates subclasses of the class it creates proxies and components for final classes cannot be used anywhere in the context of delegation. Final classes can be used as automatic forwarding components, because we only need an instance, not a subclass. Having automatic forwarding as an option allows us to have a great number of delegation's advantages for final classes too. Specifically, in the context of unmodifiable code, it allows the user to have an extension mechanism for final classes. However, these extended classes cannot be used as instances of their superclass since no proxies can be created for final classes.

Finally, this way of forwarding allows for a dynamic addition to `self`. As we have seen, components are generated on the fly, which means that they are created at the moment they are added to some `self`. If we, however, already have some live object that we would like to be added to `self` this approach doesn't work. An approach with automatic forwarding allows live objects to be added. In some cases, all we can obtain of an object is a live version, for example because it's part of some framework.

equals

A number of methods is shared across all Java Objects. Some of them are expected to be redefined for subclasses in specific cases. Since `Self` is a subclass of `Object` there may exist a need for them to be overridden. For each of these methods their semantics will be examined and, if necessary, a new implementation will be given.

The first of the object methods that will be examined is `equals`. Some definition of equals existed in Groeneveld's implementation, but the choices made were not defended in any way so I've taken the liberty of simply choosing a new, defended one. Since the definition of `equals()` itself raises a number of questions we will deal with these first. Secondly we will argue for a particular choice of semantics for the `equals()` method, considering as everywhere in this paper how we can find a mix of the ideal situation and what's possible in Java. Finally we will shortly look at implementation.

A definition of equality

Firstly, what is the `equals` method? Because the method is problematic to begin with, a short answer to that question is hard to find. Intuitively, equals provides a mechanism to query for value equality amongst objects. This as opposed to object identity, which means that two objects are identical, i.e. that two references point to the same address.

Take, for example, a look at this code:

```
Point point1 = new Point(2, 3);
Point point2 = new Point(2, 3);
if (point1 == point2)
  System.out.println("Something is wrong (1)");
if (!point1.equals(point2))
  System.out.println("Something is wrong (2)");
if (point1.equals(new Point(1, 3))
  System.out.println("Something is wrong (3)");
```

This presumes that `Point.equals()` is implemented in such a way that the values 2 and 3 are actually considered. The default implementation for equals is object identity.

A more precise definition is harder to give. Let's look at Java's definition from the Javadocs:

> The equals method implements an equivalence relation on non-null object references. It is reflexive: for any non-null reference

> value x, x.equals(x) should return true. It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true. It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true. It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified. For any non-null reference value x, x.equals(null) should return false.

Additionally, it's important to state the Liskov Substitution Principle here, since it is an important principle of object oriented design and clashes with the definition above. "In class hierarchies, it should be possible to treat a specialized object as if it were a base class object."

To see how these two principles clash, consider the following. Say we have two-dimensional points `Point` that are equal if and only if both their `x` and `y` coordinates are equal. How do we implement the `equals` method in that case? Which of the options in the following fragment should we use?

```
public class Point {

  //...

  public boolean equals(Object o) {
    if (!(o instanceof Point)) return false; //option A
    if (o.getClass() != Point.class) return false; //option B
    Point other = (Point) o;
    if (other.x != x) return false;
    return (other.y == y);
  }
}
```

For cases where `o` is a `Point`, the two options above will give the same result. However, say the class `Point` is extended to a class `ColoredPoint` that has an additional property for color. Its `equals` method has the two options seen previously, and is extended with an extra check for color equality:

```
public class ColoredPoint {

  //...
  public boolean equals(Object o) {
    if (!(o instanceof ColoredPointPoint)) //option A
    if (o.getClass() != ColoredPoint.class) //option B
      return false;
    ColoredPoint other = (ColoredPoint) o;
    if (other.x != x) return false;
    if (other.y != y) return false;
    return (other.color.equals(color));
  }
}
```

Now, which option is the correct one? If we choose option A we violate the property of symmetry, since colored points expect other colored points, but points expect only points:

```
public class TestCase1 extends TestCase {

  public void testSymmetry() {
    Point point = new Point(2, 3);
    ColoredPoint coloredPoint = new ColoredPoint(2, 3, Color.RED);
    assertTrue(point.equals(coloredPoint));
    assertFalse(coloredPoint.equals(point));
  }
}
```

However, if we choose option B, we violate the Liskov Substitution Principle. Are colored points equal to their equivalents without color? The Liskov principle would suggest so. We will leave these questions open until providing a solution in our implementation of the method.

Seeing that the theory surrounding object equality is highly problematic, let's look at its practice. The method `equals` is used mainly in two cases. Firstly in testing, and secondly in the Collection framework. Following from this definition, we'll say that `equals` should return true whenever two objects have the same value.

Equality in Delegator

In the following we will present our definition of equality in Delegator and then defend it.

> If two instances of `Self` have equal, and equally ordered lists of component classes, and for each of the components holds that `equals()` produces true with respect to the other component with the same index, the two instances are equal. Proxies are considered equal to the selves they're a proxy for. Following from this, different proxies to the same `self` are equal.

The definition of equality among different instances of `self` follows directly from the intuition of value equality given in the above. The equality of a proxy and its `self` is defended. The proxy as it's been presented in this thesis is really only introduced as a reference to `self` that allows for method calls that would otherwise be impossible. Though it is in its implementation a separate object from `self`, its only purpose is to serve as such a reference. Therefore, we believe it should not be treated differently in the `equals` method.

Objects may be composed for a variety of reasons, and what *value equality* means under different circumstances is up to the programmer. Therefore, in our setup, the user can set a filter to the composed object, saying which components should be taken into account in comparing for equality and which shouldn't.

Implementing this definition in `Self`'s method `equals` is pretty straightforward. Problems do arise, however, depending on the implementations of the implementation of `equals` in the component classes.

The first complication arises depending on the implementation of `equals` in the components. The choice between the `==` identity operator and the `instanceof` operator that was left open above has consequences here. Since most components are instances of subclasses of their model classes, an implementation using the identity operator to check for a matching class would result in a `false` result for any two components. Therefore, without making a moral statement about which of `instanceof` or `==` is better for the `equals` method, we simply see that only the former works in combination with Delegator.

Another problem arises with the use of proxies. We've already seen an example of a component's `equals` method, i.e. that of `Point`. Often these methods contain long lists of fields that are compared to fields of the other object. We have seen in the introduction, however, that though proxies contain fields for reasons of implementation, that these fields may not be used. A call to `equals` to an object with as parameter a proxy may have any result, depending on the values of the proxy's fields. This is the exact reason the definition above does not say that single component instances of `self` are equal to objects that are equal to their single component: it is not possible to create reflexitivety for that definition.

hashcode

The second object method that will be examined is `hashcode`. This method is supported for the benefit of hashtables such as provided by `Hashtable` from the `java.util` package. The most important points from its contract are that multiple executions give the same result provided the object hasn't changed, and that it produces the same result for two objects that are equal according to the `equals` method. Hashcodes may be equal for differing objects, however, for an efficient implementation of `Hashtable` a roughly equal distribution of hashes proves useful. The fact that the definition of `equals` is updated forces us to update `hashCode` as well.

The solution chosen is to just pick the first component's hashcode. Though strictly speaking a better result would be generated by taking a combination over the different components, we believe for most practical applications simply picking the first will do the trick.

toString

The method `toString` returns a string representation of an object. It is used mainly for debugging purpopses. Since the general purpose of isn't well specified, we have chosen for a pretty limited implementation. We simply examine the components in order to see whether they have a specific implementation of the method and call the first if any such method exists. If none of them have an implementation we simply return `Self`'s standard `toString`, which the standard implementation of the method. However, users are free to add their own implementations by adding an object with a different implementation as the first component of `Self`.

finalize

Each object has a method `finalize()`. This method is called by the garbage collector before the memory of that object is released. This method should be called by the garbage collector only. The garbage collector does this automatically for all objects. So, if for a particular instance of a composed object no references exist anymore, at some point it will likely be garbage collected. The garbage collector has no concept of delegation and just sees a bunch of objects. The whole point of our implementation is that all kinds of methods - specifically methods on a proxy - are "implemented" as methods that call other methods. However, if we do this with `finalize()`, the call to finalize on that proxy is forwarded to the object self. This way, finalize() may be called multiple times for one self, resulting in memory references being freed multiple times - with unknown consequences. We therefore disable this method forwarding mechanism for the finalize method.

clone

The method `clone` is defined on the class `Object`, and therefore part of any `Object`. However, only if the interface `cloneable` is implemented are the objects clonable, otherwise an exception is thrown. How does this reflect on cloneable composed objects? Firstly this implies that, only if all of a composed object's components are clonable the whole is clonable. However, the scenario of a single method being executed on *all* of a composed object's components is against the general architecture of Delegator. Since no clear need has been established for an implementation of `clone` (which is a highly debatable construction to begin with), we have chosen not to implement this.

Additional methods to support delegation

When working with delegation new needs for some specific methods arise. In a context of class based inheritance it is possible to make a call to the superclass' method implementation. This is done by using the keyword `super`. A call can be made from within the code of some subclass to the same or another method that is implemented by its superclass. Say we implement some extension of a `Vector`. In this implementation we want the overriding method `add` to have additional behavior first and then call its superclass' method `add`.

```
public boolean add(Object o) {
  //...additional behavior
  super.add(o);
```

How can the same be done using delegation? The parallel structure in that paradigm is an object composed of firstly the extension, and secondly a `Vector` component. However, from the extension's code the keyword `super` does not refer to the next class in `self`'s component list, but to that class' static superclass.

We therefore add a new construct. Methods that start with the, in the context of delegator reserved prefix `__next__`, are interpreted in a special way. These methods refer to the first next component after the calling component in the component list that can deal with the same method without that prefix. For example, `__next__add` in the extension's code will refer to `add` in the `Vector`'s code. As always, it is up to the programmer to integrate the calls into the type system, like so:

```
public abstract boolean __next__add(Object o);

public boolean add(Object o) {
  //...additional behavior
  __next__add(o);
```

In Groeneveld's version method resolution was done in the `invoke` method. This implies that at the time of resolving there is a reference to the calling object. In the new setup resolving methods is done once for each combination of classes in the `ComposedClass`. This also implies the reference to the proxy that calls a specific method is not available at the time of resolving. However, when dealing with a `__next__` method, we need to know which component is doing the call to know which component is the next one.

Therefore, when generating forwarding methods, methods starting with `__next__` get special treatment. Instead of asking `self` for its `composedClass` field, the forwarding method asks self for an adopted version of `composedClass`: one that starts only after a the component of a particular class.

Polymorphism and method resolution

Since Delegator adds the possibility of mapping any forwarding method to any implementing method its implementers may be tempted to add a mechanism that maps basetypes to their object wrappers or vice versa. One of the suggestions in Groeneveld's version was to have this kind behavior: allow basetypes and their wrapping classes to be freely interchanged in the forwarding methods and be implemented in either case. In such a scenario it would be possible to define a method in the proxy as such

```
public void set(int position, Object value)
```

and have it be implemented by the component's method:

```
public void set(Object key, Object value)
```

Though this may seem like a very valuable feature, it doesn't work for the general case.

Say we create a class that combines the behaviors of an associative array and a regular array. One can add elements as in a `Map`, using any object as key, or as an array, using an `int` as key.

```
public static class PHPArray {

  public void set(Object key, Object value) {
    //...
  }

  public void set(int position, Object value) {
    //...
  }

  //...

}

//...code usage:
PHPArray array = (PHPArray) new Self(PHPArray.class).
  cast(PHPArray.class);
array.set(0, "value");
```

However, following Groeneveld's suggestion, this would, using the useful property described above, lead to a call to the method

```
public void set(int position, Object value)
```

as opposed to a call to

```
public void set(Object key, Object value)
```

We have therefore chosen not to implement it.

Chapter 10

Examples

The main focus of this thesis has been to explain the progress made with Delegator. However, this has been done with the background claim that Delegator is actually of some use. In this chapter we will show a number of extended examples that should serve to illustrate that claim. The reader is invited to play with these herself.

Delegator was programmed using test driven development, i.e. for each and every single feature there are tests that illustrate its use. Furthermore there is a considerable number of tests that serve only to illustrate the use of the Delegator package as a whole. Some of these examples are highlighted below.

In doing so the structure of the introduction's claimed advantages of delegation will be followed. These are the reduction of code duplication that follows from automatic forwarding firstly, secondly the sharing of data among objects, thirdly, the power of dynamic inheritance chains and finally the general advantages of multiple inheritance.

Automatic forwarding

A number of design patterns, such as the Decorator, State and the Strategy pattern proposed in the famous book by the Gang of Four [19], depend heavily on message forwarding, or, as they call it, delegation. Using automatic message forwarding or even true delegation to implement these reduces code duplication and reduces the maintenance burden. In fact, we'd like to point out that the very existence of these particular patterns indicates the lack of a widespread acceptance of delegation.

In 1995 the "Gang of Four" introduced a number of design patterns in the book by the same name [19]. A number of these can be implemented using delegation, solving some of the drawbacks presented in the book and providing clarity. In the following, we will consider the State and Strategy patterns. In fact these two patterns are two of the most popular patterns from the book.

Let's look at what the two patterns are at some more detail before showing how Delegator can be useful. The Gang of Four's descriptions follow below:

State: "Allow an object to alter its behavior when its internal state changes. The object will appear to change its class."

Strategy: "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."

These two patterns are very closely related in their structure. The difference is in their semantics or usage. In State behavior is changed, in Strategy the behavior stays the same but the algorithm is changed. Because they are so closely related in their implementation, we will see that the benefits of using delegation will be identical.

In Delegator we simply replace the message forwarding link by a composition. This solves a number of things. Thirdly the programmer is relieved from the duty of forwarding all delegate methods manually. In a context without Delegation every method that is called on first the mother object, and then on the State or Strategy needs an explicit equivalent in the code of the mother object.

Secondly, in a context using Delegator, we have the self problem solved. This means the State or Strategy can actually use methods from the delegating object without this object being explicitly passed as a parameter.

Thirdly, this is done in a context where encapsulation is preserved. Any protected methods may be accessed from the delegate, but not from any client. To clarify this point, consider the following: solutions that try to mimic delegation using message forwarding have trouble encapsulating well and at the same time enable close cooperation between the object playing the role of delegate and the one playing the role of a delegator.

Let's look at the State design pattern in some more detail. In Gamma's example a `TCPConnection` maintains a reference to a `TCPState`. The methods `open()`, `close()` and `acknowledge()` on the connection are forwarded (manually) to a subclass of `TCPState` which implements specific behavior for the state the connection is in.

However, we run into trouble at this point. Say a `TCPConnection` has a number of properties independent of the rest of it state, like a port number. In fact this is very likely, because if `TCPConnection` would have no behavior of its own there wouldnt be much reason to have it in the first place. These properties will be represented as fields of the `TCPConnection`. Now say these fields can be changed in different ways in the different states. This poses a problem: how does the delegate access the delegators fields? A few options are available. We could pass the fields that may be manipulated as parameters, pass the delegator (`TCPConnection`) as a parameter or instantiate the delegate (`TCPState`) with a reference to the delegator (`TCPConnection`).

None of the available options seem desirable. If we pass only the necessary fields to the delegate, we have to treat all delegates in the same way, even though some delegate classes may not require any of these parameters. Furthermore we violate the principle of encapsulation because to know which fields to pass the delegator needs information on the delegates possible functionality. If we try to access the delegators fields via a reference to the delegator (the latter two options) we cannot encapsulate the fields anymore, since the delegate has to be able to access them. Gamma proposes to declare the two classes as friends, but this construction isn't available in the Java language. In delegation we can make it so that the delegate can access the delegators protected methods as if they where its own.

Sharing of state

In delegation object can delegate to other objects, as opposed to just inherit from other classes. In a system of class based inheritance reuse between objects is limited to methods and field descriptions, i.e. classes and functionality can be reused but objects and state can't. In delegation state can be freely shared between different objects, leading to a reduction of code duplication. Another advantage that follows from delegating to objects is that it eases the adaptation of existing legacy or library code. Objects that are either provided by library factory methods, or classes that cannot be adapted, can be adapted using a new object with extended behavior that delegates to the existing object.

Consider a drawing application. Suppose we want to draw squares and circles on the screen based on their center. Using inheritance we could make both the class `Square` and the class `Circle` a subclass of the class `Point`. The metods `getX()`, `getY()`, `setX()` and `setY()` would be inherited by both circles and squares. The actual values of the fields `x` and `y` would, however, not be shared between squares and circles, or even between instances of `Square`.

Now suppose we want to connect a square and a circle by giving them a common center. Using delegation an instance of a `Square` and an instance of a `Circle` can share a common instance of `Point`. Instead of inheriting from the class `Point` we let the instances of a `Square` and a `Circle` delegate to the same instance of a `Point`. Now every time these instances are drawn, based on the values of `getX()` and `getY()`, these methods will be delegated to the same point.

Dynamic inheritance and multiple inheritance

In the case of class based inheritance all chains of inheritance are determined statically, i.e. at compile time. Furthermore, Delegator allows for multiple delegates, and therefore for multiple inheritance. The combination of these advantages allows for an even more clear separation of concerns.

Consider this example. Suppose we want certain objects in some example application to be notified of changes in others using the Observer pattern. Several objects implement the interface `Observer`, meaning that they have a method `notifyChanged()`. There are also objects that can be observed: they implement the interface `Observable`. This interface provides methods to add and remove `Observer`s. The idea is that whenever something of interest changes in the `Observable`, the `Observer`s that are added to that `Observable` are notified of the change. Our example application needs to observe changes in objects of multiple classes, let's say of type `Vector` and `TreeNode`.

We could create a subclass for each of the observable classes, changing them in such a way that they notify the `Observer`s in case of a relevant change. This however, results in code duplication between these different subclasses. All of them need code for the addition and removal of `Observer`s and code to loop over all `Observer`s and call `notifyChanged()` on all of them.

Delegation provides a way out. The methods that actually change `Observable`s in an interesting way are extended with calls to the method `changed()`. This method, however, isn't implemented in those classes. We instead make sure all `Observable`s delegate to an instance of the class `ObservableImpl`, that also contains the implementation of addition and removing `Observer`s (see Figure 10.1). In fact, this solution is already provided by systems that allow for multiple inheritance (which Java doesn't).

However, with Delegation we can go even further in the dynamic composition of components and choose an entirely different model for observing. In this new model we make the instance of the observing object a delegate of the `Observable`. The `ObservableVector` could then directly call `notifyChanged()`, that is implemented by the observer. This would do away altogether with the code to maintain multiple observers, albeit at the cost of not being able to maintain multiple observers. An elegant solution that

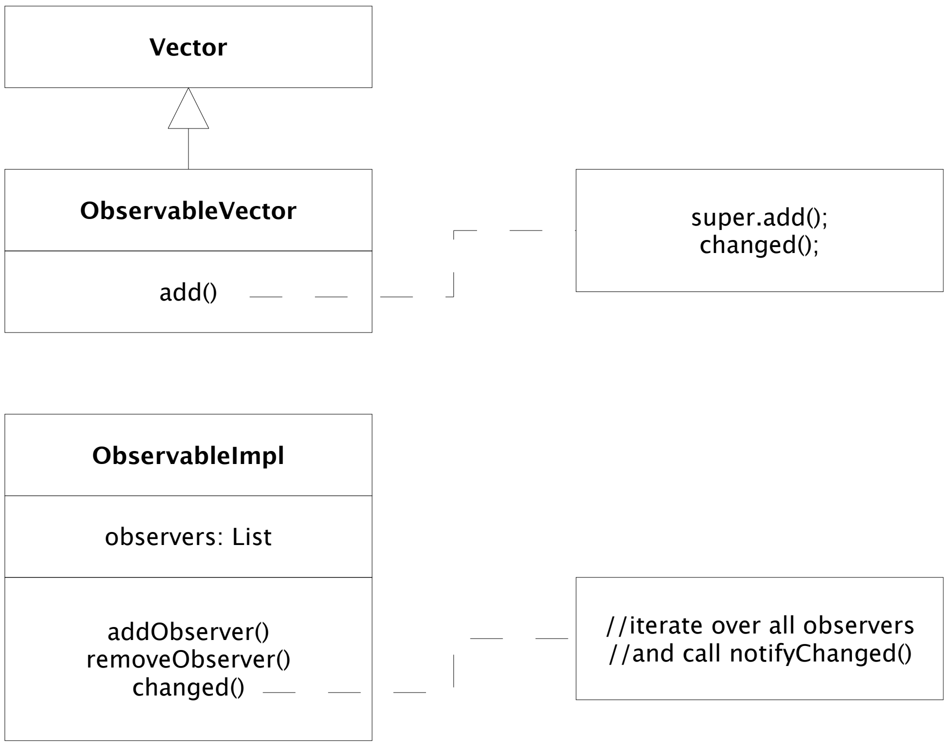


Figure 10.1: Separation of concerns using delegation

combines the best of both worlds would be to rename `ObservableImpl`'s method `changed()` to `notifyChanged()`. In this case we could use the `ObservableImpl` if multiple `Observer`s are necessary and leave it out if thats not the case.

Code reuse

There may be many parts of a program that can not easily be adapted. Code may for example be impossible to adapt because the source code isn't available or not editable in the case of a standard library. In other cases source code may be undesirable to edit because the legacy code is considered too unstable to do so.

Using delegation it is possible to add new functionality to existing code in a much cleaner fashion than without. The new functionality can simply be put in some delegator which forwards all other behavior to the original, the delegate.

Chapter 11

Conclusion

Delegation remains, for many even experienced developers and architects, an unknown subject. This thesis is part of an effort to change that.

Delegation is a very powerful concept. Its first advantage is that it relieves programmers from the burden of manually coding and maintaining the forwarding of methods to some forwardee. We feel that this advantage alone may prove to be Delegator's main benefit for many users. Some important design patterns, such as State and Strategy, rely heavily on this concept. True delegation, however, also solves the self problem. This means parts of the composed object can indeed refer to themselves as if they were the whole. In many situations, such as in the case of creating multiple inheritance, this is an important requirement.

We've seen that it is possible to combine a powerful dynamic concept such as delegation with a statically and strongly typed language such as Java. Secondly we've seen that it is possible to do so without changing the compiler, or requiring some kind of tool in the build process. All we need is the API provided as a `.jar` file. We've also seen that a number of possible problems were easily solved. Delegation combines easily with Java's exception mechanism. We have also shown that it is easy to combine Delegation with Java's concurrency model, both in terms of making sure it is safe as in using concurrency and locking in the context of delegation.

Combining Delegation with Java's access control proved considerably harder. Fortunately we have also been able to provide a number of entry points for workarounds and a set of rules that makes Delegator safe within the context of access control.

This thesis has been a considerable contribution to Delegator's performance. Whereas the virtual total lack of attention for this subject had resulted in a slow framework, Delegator is now at least in the same order of magnitude as Java without Delegation. The design of delegator has been retrofitted as a collory of this - providing insight in what Delegation is about in the first place.

It has been attempted to keep the actual user of Delegator in mind when developing. One of the results thereof has been considerable effort being put in useful standard-object methods on self and the availability of `__next__` as a prefix similar to the super construct in Java.

We believe the reintroduction of delegation in object oriented programming is a logical step. This thesis has provided a number of examples in which we think the case for Delegation is very strong. We hope that Delegator will find widespread adoption.

The number of obvious further improvements to Delegator is small. However, since the adoption of Delegator is now very limited, we believe that more widespread adoption will lead to further development. Firstly this is likely to expose current limitations. However, we believe also that new possibilities will be exposed.

A number of further developments has been mentioned throughout the text and will be summarized here. One of the main presumptions of this work has been that Delegator should be runable as a single API. However, for a number of issues, especially concerning Java's access control, it would be very useful to have the ability to do some form of source-code analysis at compile time. We think adding a separate analyzer during compile-time could prove useful.

Finally, introducing Delegation in other languages than Java could prove an interesting possibility. A likely candidate would be Python. Its lack of static typing and very strong reflection model would be useful traits in introducing Delegation. Furthermore, its community of users is, we feel very open to new developments in the object oriented field.

Bibliography

[1] Lieberman, H., “Using prototypical objects to implement shared behavior in object-oriented systems,” *Proceedings of OOPSLA ’86, Object-Oriented Programming Systems, Languages, and Applications,* (November 1986): pp. 214–223.

[2] Ulrich, Frank, “Delegation: An Important Concept for the Appropriate Design of Object Models,” *Journal of Object-Oriented Programming,* **volume**(13, No. 3), (June 2000): pp. 13-18.

[3] Kniesel, Gunter, “Article Title,” *Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems,* (Bonn 1995)

[4] *http://research.sun.com/self/language.html*

[5] *http://javalab.cs.uni-bonn.de/research/darwin/*

[6] Randall B. Smith and David Ungar, *Programming as an Experience: The Inspiration for Self.*

[7] Robert Tolksdorf and Kai Knubben, *dSelf, A Distributed SELF, Technische Universitat Berlin, Fachbereich Informatik* (Berlin, 2001).

[8] Birtwistle, G.M. (Graham M.), *SIMULA begin,* (Philadelphia, Auerbach, 1973).

[9] *http://www.franz.com/support/documentation/6.2/doc/flavors.htm*

[10] Kniesel, Gunter, *Darwin – Dynamic Object-Based Inheritance with Subtyping,* (University of Bonn, Germany, 2000)

[11] *http://java.sun.com/products/javabeans/glasgow/*

[12] Kniesel, Gunter, “Delegation for Java: API or Language Extension?,” *Technical Report IAI-TR-98-5,* (Computer Science Department III, University of Bonn).

[13] John Viega, Bill Tutt, and Reimer Behrends, "Automated delegation is a viable alternative to multiple inheritance in class based languages," *Technical Report CS-98-03,* **volume**(2), (1998)

[14] *http://www.cq2.nl/en/oss/delegator/toon*

[15] *http://www.cq2.nl/en/oss/delegator/delegationinjava/toon*

[16] *http://sourceforge.net/projects/delegator/*

[17] *http://java.sun.com/docs/books/tutorial/java/javaOO/accesscontrol.html*

[18] C.A.R. Hoare, "Monitors: An operating system structuring concept," *Communications of the ACM,* **volume**(17(10)), (October 1974): 549–557.

[19] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* (AdissonWesly, Riding, MA, 1995)