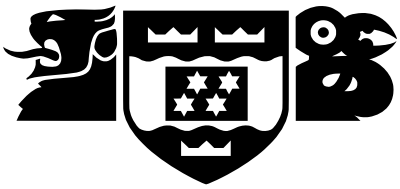
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui



School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

An Empirical Study of Delegation vs. Inheritance

Luke Inkster

Supervisors: Alex Potanin, James Noble & Tim Jones

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours.

Abstract

A study of how delegation and inheritance are used in existing programming languages. This paper explores patterns of delegation and inheritance in languages which support their implementation and provides a comparison of the uses of both.

Chapter 1

Introduction

The aim of this project is to provide a clearer understanding of the ways delegation and inheritance are used in real world software development projects across programming languages with varying support for each. This is achieved by parsing existing examples of programming projects to produce empirical evidence of the frequency at which these two structures are used in typical programs.

1.1 Motivation

The motivation of this study is to answer the question ”Is delegation useful?” by investigating the extent to which developers make use of it in their software projects. This information can inform the design of new programming languages which must, to some extent, make a choice between an inheritance or delegation based object model.

1.2 Proposed Solution

To determine the usefulness of delegation relative to classical inheritance, this study will compare the prevalence of patterns representative delegation in inheritance based languages and the prevalence of class usage in languages which natively do not support them.

1.3 Goals

Chapter 2

Literature Review

2.1 Are we Ready for a Safer Construction Environment [4]

This paper by Yossi Gil and Tali Shragai discusses the cases where a Java program is dependent on subclasses being constructed under the Uniform Identity inheritance model. It discusses the three key stages of object creation and how each of these contributes to the issues surrounding the construction of objects within class hierarchies. These stages are:

1. Memory allocation
2. Preliminary field initialisation
3. Establishment of invariants

Each of these is dealt with differently across different programming languages. As an example, preliminary field initialisation is approached quite differently in c++ when compared with Java. Java takes the approach of initialising these fields to default values (nulls, zeros and falses) whereas, in the interest of performance, c++ simply leaves these fields with whatever bytes were already present in the memory locations.
Variations between different languages implementations of the final stage, the establishment of invariants, are what leads to differing rules surrounding what the developer can and cant do safely in an object constructor. This is where we find that maintaining a Uniform Identity throughout construction is vital in ensuring that any references to the self which were stored externally during construction remain valid after this process is completed.
We also run into another issue with the changing of the self reference during the construction of an object. During the initialisation of a subclass, it is necessary at some point to initialise the superclass so that its fields are defined after construction. If, during the initialisation of the superclass, the self reference is different to that of the subclass, then any calls to overridden methods will execute the superclass's implementation rather than the subclasss.

2.2 Understanding the Shape of Java Software [1]

This paper details an empirical study of a Java corpus to find details about the structure of typical Java programs. The study collected a large set of Java classes and looked at the frequency of occurrence of a variety of common patterns including the ways developers are typically making use of inheritance and composition. As a result of the study it was found that the frequency of several of these patterns exhibited a power-law.
A further interesting finding of the study was a fairly wide variation in the occurrences of some patterns from project to project. This indicates that some architectural decisions may

contribute heavily to the patterns employed by developers as the project goes on. This also makes it evident that it will be important, in my own empirical study, to ensure that I have a wide range of Java projects from which to gather my statistics to minimise the bias that would be introduced in a smaller dataset.

2.3 Micro Patterns in Java Code [3]

This paper explores the use of micro patterns found in Java programs. These micro patterns are described as similar to design patterns, except standing at a lower, closer to the implementation, level of abstraction.
The patterns I intend to search for as possible examples of forwarding and delegation fit under this definition as each can be expressed as a function over the content of the class.

2.4 The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies [6]

This paper discusses many of the choices behind the construction of the Qualitas Corpus. Notably, the Java language was chosen for a few specific reasons:

- Open source Java code is abundant and easy to find. Much more so than c#, and similarly to c++.
- Java code tends to be relatively easier to analyse than many other languages including c++ due to some of the limitations it applies to program structure.

2.5 What Programmers Do with Inheritance in Java [8]

This paper goes into detail about the use of inheritance in Java projects and the extent to which classes extend others. To aid with this, the paper also contains a formal definitions of a few terms which are relevant to my study, including subtypes, supertypes and downcalls. These definitions are then used to indicate rates of presence in the Qualitas Corpus of a variety of combinations of the patterns. This is achieved by representing the dependencies within the projects as a graph structure and investigating the properties of that graph.

2.6 Towards a Metrics Suite for Object Oriented Design [2]

Included in this paper are a variety of useful terms for defining measurements of inheritance within programs written in object oriented languages. These include:

- **Depth of Inheritance Tree (DIT)** - A measure of the number of ancestor classes which can potentially affect a given class. This is the depth of the class in the class hierarchy.
- **Number of children (NOC)** - The number of immediate subclasses under a given class in the class hierarchy. This is the number of classes which will, unless explicitly overridden, inherit the methods of the parent class.
- **Coupling between objects (CBO)** - A measure of the non-inheritance relationships a class shares with other classes. This can be seen as a measure of the interdependence of classes in a given program.

2.7 How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software [7]

The authors here explore the use of inheritance in Java programs, primarily in large-scale software development projects, to form a more clear idea of the extent to which particular inheritance patterns are used. The analysis involved over 100,000 classes and interfaces across 90 Java projects. The results of this study show that approximately three quarters of all Java classes in the study had some transitive superclass other than Object in at least half of the examined corpus.
Another contribution of this paper is an explicit discussion of the distinction Java, along with similar languages, makes with regard to it's *extends* and *implements* relationships between classes and their superclasses and interfaces respectively.

2.8 Object Inheritance Without Classes [5]

Tim's paper talks about several different models of object based inheritance and the inherent limitations of both. From this, it becomes evident which Java programs are dependent on the Uniform Identity model which is used to construct instances of Java classes within inheritance hierarchies. Knowing which classes are dependent on this model also informs us about which classes would need to be substantially rethought in order to reimplement them in a language which does not support uniform identity.

Chapter 3

Background

3.1 Examples in Java

3.1.1 Forwarding

Forward this.f() calls to super.f() passing along any necessary information as call parameters

In Java, this involves looking for patterns where an objects method does very little work besides forwarding the call to another method. This is the simplest form of delegating responsibility to another class and should be independent of the state of the delegatee, since it needs to be able respond correctly to requests from other delegators without influence from state set in previous requests.
If the receiver of a forwarded requests were to hold state about an object delegating to it the wed likely run into issues if other objects also forward requests to it. Likewise, if we re-implemented a stateful forwarding recipient in a language which supports forwarding as object inheritance then wed run into the same problems when sharing it between parents.
In this example the Square object is forwarding responsibility for its area calculation to the SquareAreaCalculator. The SquareAreaCalculator could be shared by many Squares as it holds no state and therefore does not rely on being instantiated as an instance specific to any given square.

```
class Square{
   int x, y, wd;
   SquareAreaCalculator areaCalculator = new SquareAreaCalculator();

   int area(){return areaCalculator.calculate(wd);}

   Square(int x, int y, int wd){
      this.x = x; this.y = y; this.wd = wd;
   }
}

class SquareAreaCalculator{
   int calculate(int wd){return wd * wd;}
}
```

3.1.2 Delegation

Forward this.f() calls to super.f() ***on behalf of this.*** *That is, call super.f() but have the self reference within that call set to my self reference.*

Examples in Java which would be well suited to a language which supports delegation are where the code is effectively forwarding to an object which accepts **this** as either a constructor parameter or as a parameter to many of its public methods. This indicates that the object being called to is doing a lot of work which is dependent on the **this** reference being passed in. By using a language which supports delegation natively, it would be possible to change the self reference of the delegatee to instead be the self reference of the delegator.
In this example, the Square object is delegating responsibility for area calculation to the SquareAreaCalcuator. The SquareAreaCalculator contains a final field to point to the self reference of a single Square object which indicates that the calculator belongs to one instance of Square and always will. In an object delegation model the public final field could be removed, instead opting to have the self reference of the calculator set to the self reference of the Square object.

```
class Square{
   int x, y, wd;
   SquareAreaCalculator areaCalculator = new SquareAreaCalculator(this);

   int area(){return areaCalculator.calculate();}

   Square(int x, int y, int wd){
      this.x = x; this.y = y; this.wd = wd;
   }
}

class SquareAreaCalculator{
   private final Square square;

   SquareAreaCalculator(Square square){this.square = square;}

   int calculate(){return square.wd * square.wd;}
}
```

3.1.3 Uniform Identity

All examples of classical inheritance in Java follow the uniform identity construction model

Under uniform identity, objects are constructed by first going up the object hierarchy setting up fields, then going back down the hierarchy calling initialisers. This maintains a single object identity throughout the construction of the object.
To find examples to support Uniform Identity, we must simply look for typical uses of inheritance in Java where a subclass makes some use of functionality from the parent class. Uniform Identity is the most similar object inheritance model to Javas class inheritance pattern but others can also model the behaviour fairly closely. For example, Merged Identity closely matches the c++ model of class inheritance and, with a few exceptions, most examples of Java class based inheritance could also function in a Merged Identity model. This is the implementation Javas class based inheritance model encourages so it is expected to be the most common across corpus data, but any substantial usage of the previously men-

Java	

Forwarding	anything name (anything){
	return identifier[.identifier]*.name(anything);
	}
	Where ”name” is the same in both places

Delegation	anything name (anything) {
	return identifier[.identifier]*.name(this);
	}
	Where ”name” is the same in both places

Constructor Delegation	anything anything = new anything (this)
Inheritance	class extends anything

JavaScript	
Inheritance 1	var a = function (b) { c . call (this , d);}
Inheritance 2	function Bar (x , y) { Foo . call (this , x) ;}
Inheritance 3	Foo . prototype = object . create (Bar . prototype)
Inheritance 4 - Node.js	var className = defineClass(...)
Inheritance 5 - Node.js	util.inherits(...)

tioned examples would indicate that developers are intentionally dismissing Javas in-built language features as they believe they can produce better code with other patterns.
In this example, the Square class inherits from another class which knows how to calculate the area of a more general case so can also be used to offer the same functionality to the Square.

```
class Rectangle{
   int x, y, wd, ht;

   int area(){return wd * ht;}

   Rectangle(int x, int y, int wd, int ht){
      this.x = x; this.y = y; this.wd = wd; this.ht = ht;
   }
}

class Square extends Rectangle{
   Square(int x, int y, int wd){super(x, y, wd, wd);}
}
```

3.1.4 Patterns

Chapter 4

Conclusions

Not sure if this is actually a section in a preliminary report.....

Bibliography

[1] BAXTER, G., FREAN, M., NOBLE, J., RICKERBY, M., SMITH, H., VISSER, M., MELTON, H., AND TEMPERO, E. Understanding the Shape of Java Software. *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (2006), 397–412.

[2] CHIDAMBER, S. R., AND KEMERER, C. F. Towards a metrics suite for object oriented design. *SIGPLAN Not. 26*, 11 (Nov. 1991), 197–211.

[3] GILL, J. Y., AND MAMAN, I. Micro Patterns in Java Code. *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2005), 97–116.

[4] GILL, J. Y., AND SHRAGAI, T. Are We Ready for a Safer Construction Environment? *Genoa Proceedings of the 23rd European Conference on ECOOP* (2009), 495–519.

[5] JONES, T., HOMER, M., NOBLE, J., AND BRUCE, K. Object Inheritance Without Classes. *30th European Conference on Object-Oriented Programming* (2016).

[6] TEMPERO, E., ANSLOW, C., DIETRICH, J., HAN, T., LI, J., LUMPE, M., MELTON, H., AND NOBLE, J. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. *Asia Pacific Software Engineering Conference* (2010), 336–345.

[7] TEMPERO, E., NOBLE, J., AND MELTON, H. *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ch. How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software, pp. 667–691.

[8] TEMPERO, E., YANG, H. Y., AND NOBLE, J. *ECOOP 2013 – Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, ch. What Programmers Do with Inheritance in Java, pp. 577–601.