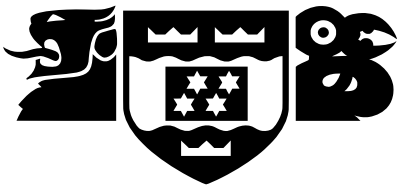
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui



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.

Contents

Chapter 1

Introduction

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

1.1 Motivation

Code reuse mechanisms are a vital aspect of software development. They allow engineers to write code once and make use of it in various places without having create duplicates of that code. In allowing this, code reuse aims to reduce the time and resources required to produce a software system by maximising the usefulness of each asset produced. Higher reuse of code also ensures that, when changes must be made to the system, a single software modification can enact the change in more areas of the software.
Typically, code reuse takes two major forms:

- Inheritance - Inheriting the properties of some parent object to a child object.
- Composition - Objects contain other objects and delegate responsibility to them.

Each of these has its uses and comes with distinct advantages and disadvantages. Languages built with native support for delegation typically encourage the use of composition over inheritance where possible. In contrast, languages built with native support for classical inheritance will usually encourage developers to reuse code through inheritance relationships between classes.
The motivation of this study is to answer the question ”Is delegation useful?”. This will be investigated by studying the extent to which developers make use of it in their software projects and the ways they could if their language had stronger native support. 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 of delegation in inheritance based languages and the prevalence of class usage in languages which natively do not support them. This involves employing a variety code analysis of tools to parse out these patterns from

samples of each language allowing an empirical analysis surrounding the extent to which programmers in each language are making use of each pattern.

1.3 Goals

The goal of this study is to gather information which can drive design decisions in new programming languages by offering an empirical perspective on the use of delegation and inheritance across software development projects in existing languages.

Chapter 2

Literature Review

2.1 Object Inheritance Models

The paper *Object Inheritance Without Classes [5]* by Tim Jones discusses a variety of different object inheritance models and the inherent benefits and limitations of each. From this work, 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. Additionally, this information also shows which patterns could be rewritten under other inheritance models without requiring much modification and, in some cases, more concisely.

In a 2009 paper titled *Are we Ready for a Safer Construction Environment [4]*, Yossi Gil and Tali Shragai discuss the cases where a Java program is dependent on class instances being constructed under the Uniform Identity inheritance model. It covers 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 lead 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. Without the presence of a Uniform Identity, any self references which are passed out from the constructor before object creation is complete cannot be guaranteed to point back to the expected object.
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 guaranteed to be 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 subclass's.

In Henry Lieberman's paper 1986 titled *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems [6]* he first coins the term "Delegation" with respect to software language design. Lieberman then provides a plain English definition of the term which allows a reader to clearly understand the concept he is describing:

> When a pen delegates a draw message to a prototypical pen, it is saying "I don't know how to handle the draw message. I'd like you answer it for me if you can, but if you have any further questions, like what is the value of my x variable, or need anything done, you should come back to me and ask." [6]

Lieberman's definition forms the basis of the delegation patterns considered in this paper.

2.2 JavaScript Analysis

Does JavaScript Software Embrace Classes? [7] explores the prevalence of classical inheritance patterns in a JavaScript corpus. JavaScript is a useful language to investigate for this study because it provides many examples where developers are deliberately using a language built for delegation and object based inheritance to model classical inheritance structures. The paper explores the ways in which JavaScript developers typically model class inheritance and the ways these patterns can be detected in corpora of JavaScript projects. As part of this paper, the researchers also create a tool named JSClassFinder which serves the purpose of identifying both class declaration patterns and method declaration patterns. The statistics returned by this tool can then be analysed to determine the extent to which JavaScript developers are working around the language's inbuilt structures.

2.3 Java Analysis

Understanding the Shape of Java Software [1] details an empirical study of a large Java corpus to uncover details about the structure of typical Java programs. The study collected a large set of Java classes and looked at the occurrence frequency of various common patterns including the ways developers are typically making use of inheritance and composition. As a result of this study, it was found that the frequency of several of these patterns, when broken down by project, exhibited a power-law distribution.
A further interesting finding of the study was a fairly wide variation in the occurrence frequency of some patterns from project to project. This indicates that some architectural decisions made in a project's infancy may contribute heavily to the patterns employed by developers as the project progresses. This also makes it evident that it will be important, in my own empirical study, to ensure that I have a wide range of projects for each language from which to gather statistics to minimise the biases that could be introduced by using a smaller dataset.

Micro Patterns in Java Code [3] explores the use of micro patterns found in Java programs. The paper also provides a clear definition of a micro pattern upon which further work can be based.

Micro patterns are similar to design patterns, except standing at a lower, closer to the implementation, level of abstraction." [3]

The patterns this paper will search for as possible examples of forwarding and delegation fit under this definition as the detection of each can be expressed as a function over the content of the class.

What Programmers Do with Inheritance in Java [10] 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.

The authors of *How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software [9]* explore the use of classical inheritance in Java programs, primarily in large-scale software development projects. This forms a more clear idea of the extent to which particular inheritance patterns are used in the real world. The analysis performed in this study 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.
A further contribution of this paper is an explicit discussion of the distinction Java, along with similar languages, makes with regard to its *extends* and *implements* relationships between classes and their superclasses or interfaces respectively. This distinction makes it clear that, in order for code to be reused through inheritance from classes further up the type hierarchy, an *extends* relationship is required.

2.4 Analysing Corpora

The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies [8] discusses many of the choices behind the construction of the Qualitas Corpus. Notably, the paper clarifies that 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 parse and analyse than many other languages including c++ due to some of the more strict limitations it applies to program structure.

The paper also justifies the choice of projects in the corpus as they are open source and provide a wide array of different usages of the language to help to ensure variation in the code.

Included in *Towards a Metrics Suite for Object Oriented Design [2]* 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. For a given class, this can be seen as its depth in the class hierarchy tree from the root object class.

- **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. For a given class, this can be calculated as the number of elements in the type hierarchy tree rooted at that class.
- **Coupling between objects (CBO)** - A measure of the non-inheritance relationships a class shares with other classes. This is an effective measure of the interdependence of classes in a given program which are neither subclasses nor superclasses of eachother.

Chapter 3

Code Patterns

3.1 Forms of Code Reuse

3.1.1 Forwarding

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

In Java, this involves searching for patterns where an objects method does very little work besides forwarding the call to method on another object. This is the simplest form of delegating responsibility to another class and should be independent from any state held by 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 request were to hold state about an object delegating to it then it would likely run into issues if other objects also forward requests to it. Likewise, if the system were re-implemented with a stateful forwarding recipient in a language which supports forwarding as object inheritance then it would run into the same problems when sharing it between parents.
In the following example, a 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 and isolated 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 other.f(...) ***on behalf of this.*** *That is, call other.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 with native support for delegation are those 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 designed to do a lot of work which is dependent on the **this** reference of another object 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, removing the need to pass it as a parameter.
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 SquareAreaCalculator 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 initialiser functions. This maintains a single object identity throughout construction of the object.
To find examples supporting the need for 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. Uniform Identity is the implementation Javas class based inheritance model encourages so

it is expected to be the most common across corpus data. Because of this, Any substantial usage of the previously mentioned examples would indicate that developers are intentionally dismissing Javas in-built language features as they believe it is possible to 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);}
}
```

3.2 Java Patterns

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

3.3 JavaScript Patterns

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(...)

Chapter 4

Analysis

The core of this empirical study is the analysis of corpora of code written in each of the investigated languages. This analysis makes use of many static code analysis methods including:

- ANTLR
- grep
- JSClassFinder
- Esprima

Each of these tools helps to extract valuable information from one of more of the languages analysed in this study.

4.1 Assembling Corpora

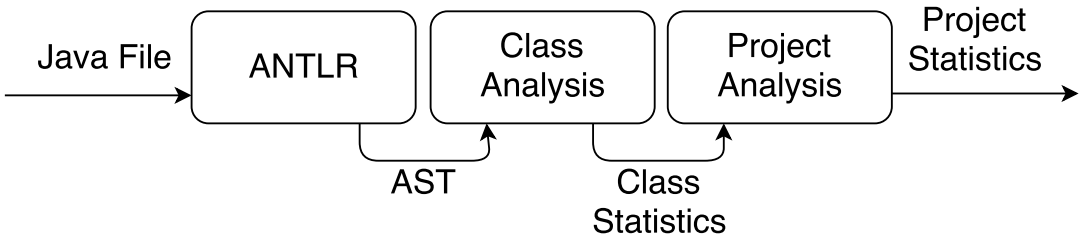
Each language analysed in this study required a corpus of code representative of the ways programmers in the real world are using that language. In the case of Java, prior studies have resulted in the creation of The Qualitas Corpus which is a large collection of open source projects written in the Java language [8].
For the other studied languages, JavaScript, Python, Lua, and Scala, quality existing corpora could not be found. For each of these languages, the top 25 open source projects were sourced from GitHub's "Trending this month" list. This source was chosen because it provides a particularly up to date snapshot of development in each language which ensures that the analysis performed will be as relevant as possible to modern software development.

4.2 Java

The intent in analysing the Qualitas Corpus of Java code is to determine the extent to which developers are making use of Java's inbuilt language features and what developers are doing to work around these language features. Specifically, a Java developers' usage of class inheritance will represent them conforming to the classical inheritance model encouraged by the Java language. In contrast, instances of code which model call forwarding or call delegation will represent cases where the developer could have expressed themselves more concisely through other object inheritance models where delegation and forwarding are supported natively.
Finding occurrences of classical inheritance in Java is as simple as looking for the extends

keyword with a grep search. Finding examples of delegation and forwarding is more difficult and requires more information about the syntax tree of the program. To achieve this, each program of the corpus was passed through ANTLR which parses each file according to a Java grammar and constructs an abstract syntax tree which can then be traversed to search for relevant patterns.

The process for extracting statistics from a Java project follows a pipeline structure where each file is parsed and analysed in isolation. The resulting statistics of each file are then aggregated to form the overall statistics across the projects. This file isolation is important because the syntax trees produced by ANTLR consume large amounts of memory so it is not possible to hold all the Java files for a project in memory simultaneously.



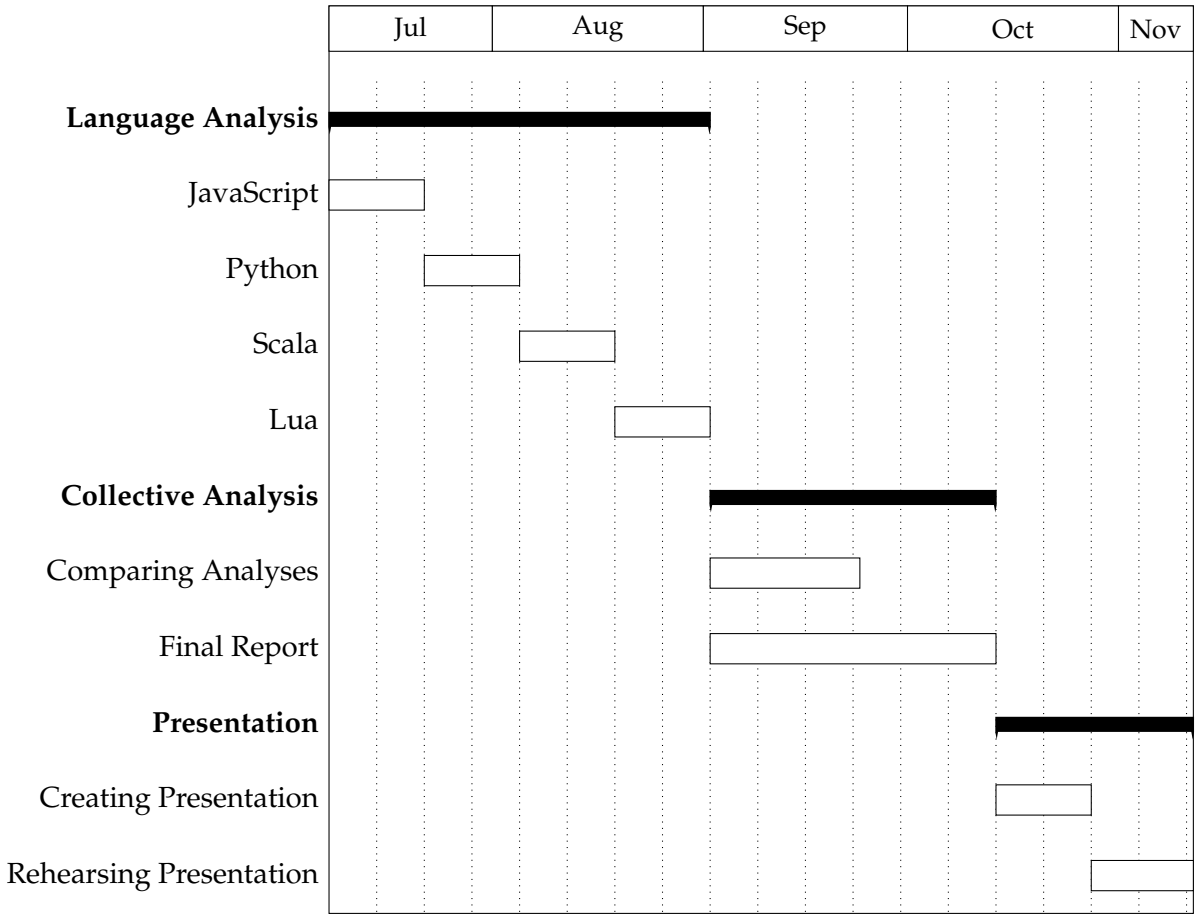
After each project has been parsed and analysed, the results are then further aggregated to produce corpus level analysis which can be found in the following table:

	Count	% of classes	% of extended classes
Total classes	116427		
Classes extend another class	71203	61.16%	
Classes are extended by another class	20751	17.82%	
Classes with forwarding	7087	6.09%	
Classes with forwarding that extend another class	3381	2.90%	
Classes with downcalls in constructors	16101	13.83%	
Classes storing this in constructors	2392	2.05%	
Classes with downcalls or storing this in constructor	17099	14.69%	
Extended classes with downcalls in constructors	1545	1.33%	7.45%
Extended classes storing this in constructors	178	0.15%	0.86%
Classes with delegation	5183	4.45%	

4.3 JavaScript

The JavaScript analysis of this study makes extensive use of the prior work in developing the JSClassFinder application [7]. The aim here is to find the cases where JavaScript developers are deliberately circumventing the native delegation support of the language and are instead modelling their programs with classical inheritance structures.

4.4 Timeline


Jul
Aug
Sep
Oct
Nov
Language Analysis
JavaScript
Python
Scala
Lua
Collective Analysis
Comparing Analyses
Final Report
Presentation
Creating Presentation
Rehearsing Presentation

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] LIEBERMAN, H. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications* (New York, NY, USA, 1986), OOPLSA '86, ACM, pp. 214–223.

[7] SILVA, L., RAMOS, M., VALENTE, M. T., BERGEL, A., AND ANQUETIL, N. Does JavaScript software embrace classes? In *Proceedings of the 22nd International Conference on Software Analysis, Evolution and Reengineering* (2015), Y.-G. Guhneuc, B. Adams, and A. Serebrenik, Eds., IEEE, pp. 73–82.

[8] 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.

[9] 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.

[10] 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.