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

This report covers an empirical study of how delegation and inheritance are used in existing programming languages. The aim of this study is to answer the question "Is delegation useful?" in a way that can be used to drive the design of new programming languages. This is achieved through an exploration of patterns of delegation and inheritance in languages which support their implementation and through 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 examining 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 duplicating that code. Code reuse aims to reduce the time and resources required to produce a software system by maximising the usefulness of each asset produced. Reuse of code also ensures that, when changes must be made to the system, a single software modification can enact the desired change in more areas of the program.
Typically, code reuse takes two major forms:

- Inheritance - Inheriting the properties of some parent object to a child object.
- Delegation - Objects pass messages to other objects, delegating responsibility to them.

Each of these has its uses and comes with distinct advantages and disadvantages. Languages built with native support for delegation object models typically encourage delegation of responsibility over inheriting properties where possible. In contrast, languages built with native support for classical inheritance will usually encourage developers to reuse code through inheritance relationships between classes.

1.2 Proposed Solution

To determine the use 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 do not natively support classical inheritance. This investigation involves employing a variety of code analysis tools to detect these patterns from representative samples of each language. The collected data will be used in an empirical analysis to determine the extent to which programmers in each language are making use of each pattern.

1.3 Goals

The goal of this study is to answer the question ”Is delegation useful?”. The question will be investigated by studying the extent to which developers make use of delegation in their software projects and the ways they could if their language had stronger native support. The results can inform the design of new programming languages which must, to some extent, make a choice between an inheritance or delegation based object model. The study will be a success if it is able to produce 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* [**?**] by Tim Jones discusses a variety of different object inheritance models and the inherent benefits and limitations of each. This paper also describes the Uniform Identity model where objects are constructed by first going up the object hierarchy setting up fields, then going back down the hierarchy calling initialiser functions. 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. Classes which are dependent on the Uniform Identity model can be expected to be more difficult to reimplement in a language without that native support. Additionally, this information shows which patterns could be rewritten under other inheritance models without requiring much modification and, in some cases, more concisely.

Henry Lieberman's 1986 paper *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems* [**?**] coins the term "Delegation" with respect to software language design. Lieberman provides a plain English example of delegation 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." [?]~~ *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." [**?**]*

Lieberman's definition forms the basis of the delegation patterns considered in this report. This definition is important as delegation is the object inheritance model natively supported by JavaScript.

In a 2009 paper titled *Are we Ready for a Safer Construction Environment* [**?**], 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, lead to different rules about what the program can and can't 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 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 constructed object after initialisation has completed.

~~We also run into another issue~~ The establishment of invariants phase of object construction also introduces potential issues 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.

Section 12.5 of the Java Language Specification [?] makes Java's approach to this problem of superclass constructor downcalls clear:

> *Unlike C++, the Java programming language does not specify altered rules for method dispatch during the creation of a new class instance. If methods are invoked that are overridden in subclasses in the object being initialized, then these overriding methods are used, even before the new object is completely initialized.* [?]

To give a short example of this, when constructing an instance of `B` in the following program, the call to `m()` in A's constructor will execute the method `m()` declared on `B`, printing the string "B".

```
\DIFadd{class A}{
   \DIFadd{A()}{ \DIFadd{this.m(); }}
   \DIFadd{void m()}{ \DIFadd{System.out.println("A"); }}
}
\DIFadd{class B extends A}{
   \DIFadd{B()}{}
   \DIFadd{void m()}{ \DIFadd{System.out.println("B"); }}
}
```

2.2 JavaScript Analysis

Does JavaScript Software Embrace Classes? [?] 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. The researchers also defined the term "Class Usage Ratio" which is a measure of the proportion of functions in a JavaScript project which are used to model class behaviour. This Class Usage Ratio is defined as:

$$CUR = \frac{|methods| + |classes|}{|functions|}$$

In this ratio, a class is considered to be any function which is used to mirror classical inheritance behaviour. Methods are functions which are held as members of instances of classes and perform some action related to that class [?].

The corpus used in the JSClassFinder study is also useful because it offers a selection of JavaScript projects which were collected before the release of the ECMAScript 6 language specification which introduces native support for classes [?] [?]. Analysing code which was created after the addition of native class support would not be interesting for this study because it would not show developers attempting to circumvent language features to achieve their desired outcome.

2.3 Java Analysis

Understanding the Shape of Java Software [?] 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 may contribute heavily to the patterns employed by developers as the project progresses. This variation 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 [?] 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." [?]~~ *Micro patterns are similar to design patterns, except standing at a lower, closer to the implementation, level of abstraction." [?]*

The patterns this study will be attempting to uncover as possible examples of forwarding and delegation fit under this definition. As such, the detection of each can be expressed as a function over the content of the class.

What Programmers Do with Inheritance in Java [?] goes into detail about the use of inheritance in Java projects and the extent to which classes extend other classes. To aid with this hierarchical analysis, the paper also contains a formal definitions of terms which are relevant to my study. These include:

1. Subtypes - A type S is a subtype of type T if an instance of S can be supplied where an object of type T is expected.

2. Supertypes - A type T is a supertype of types $S_1..S_n$ if an instance of any of $S_1..S_n$ can be supplied where an object of type T is expected.

3. Downcalls - A call to a method on an object with declared type T can call another method on a subtype S if an instance of S is provided.

These definitions are then used to measure the frequency of occurrence 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* [**?**] 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 [**?**] discusses many of the choices behind the construction of the Qualitas Corpus. A corpus is defined as *"a collection of writings, conversations, speeches, etc., that people use to study and describe a language"*. In the Qualitas Corpus, the collection is of projects written in the Java programming language. This paper explores the reasoning behind the choices which led to the structure of the corpus as it is. 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 the simpler grammar of the language.

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.

Towards a Metrics Suite for Object Oriented Design [**?**] includes 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

To determine how developers are using delegation and inheritance, the first stage is to define methods to identifying each. This is achieved by defining code patterns which, when found in a project, indicate the use of delegation or inheritance. This chapter outlines the patterns which are representative of forwarding, delegation and uniform identity.

3.1 Forwarding

Under a forwarding model of object inheritance, calls to ~~this.f(...)~~ `this.f(...)` are passed to some ~~other.f(...)~~`other.f(...)`, transferring any necessary information as call parameters.

In the following example, a ~~Square~~ `Square` object is forwarding responsibility for its area calculation to the ~~SquareAreaCalculator. The SquareAreaCalculator~~ `SquareAreaCalculator`. The `SquareAreaCalculator` could be shared by many ~~Squares~~ `Square` objects as it holds no state and therefore does not rely on being instantiated as an instance specific and isolated to any given ~~square~~`Square`.

```java
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;}
}
```

In Java, this involves searching for patterns where an object's method does very little work besides forwarding the call to method on another object. This is the simplest form of transferring responsibility to another class and should be independent from any state held by the forwardee. This is because needs to be able respond correctly to requests from other forwarders 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.

3.2 Delegation

Delegation can be seen as forwarding ~~this.f(...) calls to other.f(...)~~ `this.f(...)` calls to `other.f(...)` **on behalf of this.** That is, call ~~other.f(...)~~ `other.f(...)` but have the self reference within that call set to my self reference.

In this example, the ~~Square~~ `Square` object is delegating responsibility for area calculation to the ~~SquareAreaCalcuator. The SquareAreaCalculator~~ `SquareAreaCalcuator`. The `SquareAreaCalculator` contains a final field to point to the self reference of a single ~~Square~~ `Square` object which indicates that the ~~calculator~~ `SquareAreaCalculator` belongs to one instance of ~~Square~~ `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~~ `SquareAreaCalculator` set to the self reference of the ~~Square~~ `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;}
}
```

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**~~ `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**~~ `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.

3.3 Uniform Identity

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.

In this example, the ~~Square~~ `Square` class inherits from another class which knows how to calculate the area of a more general figure so can also be used to offer the same functionality to the ~~Square~~`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);}
}
```

All examples of classical inheritance in Java follow the Uniform Identity construction model. Therefore, 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 object inheritance model underlying Java's class inheritance structure, 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 Java's class based inheritance model encourages so it is expected to be the most common across corpus data. Because of this, any substantial use of forwarding or delegation would indicate that developers are intentionally dismissing Java's in-built language features as they believe it is possible to produce better code with other patterns.

Chapter 4

~~Analysis~~Methodology

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 the following:

- Grep is used to perform regular expression searches on files. This can detect the more simple patterns explored in this paper.
- ANTLR (Another Tool For Language Recognition) is a tool which accepts a language grammar and a valid file in that language. From this it constructs a syntax tree to represent the file.
- JSClassFinder is a tool which detects class and method declarations in JavaScript code.
- Esprima accepts a JavaScript file as input and produces a JSON representation of the syntax tree of that file. This JSON file is then used as the input to JSClassFinder.

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

4.1 Selecting Languages

The languages which have been explored thus far are limited to Java and JavaScript. These languages were covered first because they both have large open source communities and they differ on their native method of object inheritance which provides a good avenue for comparison. The other languages which will be explored in the remainder of the project are ~~Python, Lua and Scala~~Lua and C#. These are included to ensure that a wide enough variety of languages are analysed that it is possible to make conclusions about the usage of delegation and inheritance across languages.

4.2 Assembling Corpora

To analyse each language, we first needed to collect a corpus representative of that language's use in real world software development projects. In the case of Java, we adopted The Qualitas Corpus, which is a large collection of open source projects written in the Java language [**?**]. Likewise, with JavaScript, we have adopted an existing corpus used by the team that developed JSClassFinder [**?**].

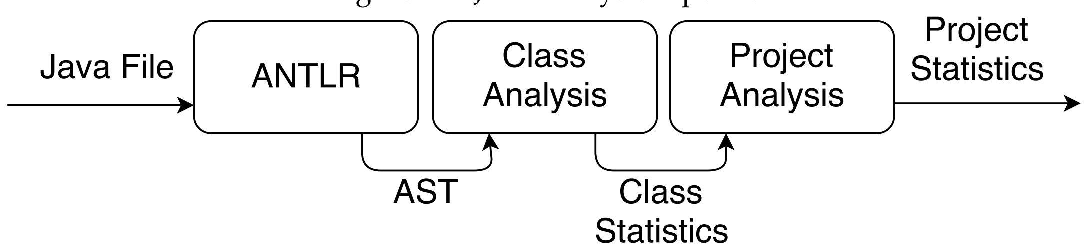
For the other studied languages, ~~Python, Lua , and Scala~~Lua and C#, quality existing corpora could not be found. For each of these languages, the top 25 open source projects were sourced from GitHub's "Trending this month" list for June, 2016. This source was chosen because it provides a group of projects for each language which are in active development as measured by GitHub, and which are easy to access. This helps to ensure that the analysis performed will be as relevant as possible to modern software development.

4.3 Java Analysis

Finding occurrences of classical inheritance in Java is as simple as looking for the extends keyword with a "grep" regular expression 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 lexer and parser generated from a Java grammar. ANTLR then constructs an abstract syntax tree which can then be traversed to search for relevant patterns.

The process for analysing 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.

Figure 4.1: Java Analysis Pipeline



4.4 C# Analysis

As with Java, C# was analysed using a lexer and parser generated by loading a C# 6 grammar into ANTLR. The analysis for each project in the corpus was performed in three major passes:

1. Create a syntax tree for each code file and traverse it to find all class declaration subtrees. Collect these class declarations to use in later steps.
2. Run a visitor down each class declaration subtree, searching for all the methods and recording their modifiers. A type hierarchy is also established at this step to allow classes to find information about method calls they make which may be dispatched to a method in their superclass.
3. Run another visitor down each class declaration tree and find constructors and check which methods are called against the modifiers found in the previous pass to determine

which methods could miss their intended target under a different object initialisation model.

The statistics gathered for each file in each project were then aggregated across the corpus to collect information about the corpus as a whole.

4.5 JavaScript Analysis

4.6 Lua Analysis

The Lua corpus was analysed with Grep to identify code patterns and keywords associated with class usage.

Chapter 5

Results

5.1 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. The following patterns are used to identify instances of each model of reuse within the Java projects.

Table 5.1: Java Patterns

Java	
Forwarding	~~anything name (anything){~~ `anything name (anything){` ~~return identifier.identifier*.name(anything);~~ `return identifier[.identifier]*.name(anything);` ~~}~~ `}`
Call Delegation	~~anything name (anything) {~~ `anything name (anything){` ~~return identifier.identifier*.name(this);~~ `return identifier[.identifier]*.name(this);` ~~}~~ `}`
Constructor Delegation	~~anything anything = new anything (this)~~ `anything anything = new anything ( this )`
Inheritance	~~class extends anything~~ `class extends anything`

The presence of two patterns representing delegation is because there are two main ways this behaviour can be represented in Java. The first, call delegation, is where an object passes itself as a parameter to some delegatee and has that delegatee perform some action on its behalf. The second, constructor delegation, is where a delegatee is constructed specifically for the instance of the delegator. This delegatee can then act on that constructor argument when its other methods are called.

~~Finding occurrences of classical inheritance in Java is as simple as looking for the extends keyword with a "grep" regular expression 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 analysing 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.~~

~~The results are then aggregated to~~ The frequency of occurrence of each of the above patterns was calculated and aggregated to produce corpus level analysis which can be found in the following table:

Table 5.2: Java Analysis Results

	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%	

5.1.1 Detecting Delegation

Due to the imprecise definition of what it means to model delegation in Java, it is expected that some of the results which are returned in the search for these delegation patterns will be false positives. Upon manual inspection of some of the corpus files, some of the patterns which meet the criteria outlined in 5.1 would not generally be considered to exhibit the true behaviour delegation. An example of this is object self registration where and object registers itself with some other object which will make use of the registered object in some way. To continue the example of the `Square` class, an instance of `Square` could, at construction time, register itself to a `Canvas` so that the `Canvas` can call back to the `Square` to request details necessary for drawing. The `Square` class would have a constructor parameter which is a reference to the `Canvas` object:

```
\DIFadd{public Square(Canvas canvas)}{
   \DIFadd{canvas.register(this);
}}
```

And a `Square` could be added to the `Canvas` as follows:

```
\DIFadd{void addSquare()}{
   \DIFadd{Square s = new Square(this);
}}
```

This matches the pattern of Constructor Delegation in 5.1 but , in reality, is modelling a different intent. Because of this, the statistics gathered for delegation should be treated as an upper bound on the actual frequency of occurrence of the behaviour.

5.2 C#

C# is a useful language to investigate because it requires use of the `virtual` keyword to enable overriding of any given method, otherwise defaulting to static method dispatch. This is interesting because it forces the developer to make their intent to override a method explicit, in contrast to Java where virtual dispatch is the default and occurs when the developer has simply omitted the `final` modifier. This makes it much more clear whether there could potentially be construction issues if we had a different way of initialising objects in place of Uniform Identity. When the `virtual` keyword is required, the only method calls which could miss their intended target when used in a constructor are those which are explicitly labelled as `virtual` dispatch calls.

Table 5.3: C# Analysis Results

	Total	% of methods	% of classes
Projects	25		
Files	31615		
Classes	71162		
Extending Classes	25654		36.05%
Methods	232282		
Virtual Methods	11194	4.82%	
Override Methods	28815	12.41%	
Classes with calls to local methods in constructors	1731		2.43%
Classes with calls to local virtual methods in constructors	117		0.16%
Classes with calls to local override methods in constructors	56		0.08%
Classes with calls to local abstract methods in constructors	17		0.02%
Classes with calls to methods that couldn't be found in constructors	479		0.67%

The first notable difference between the C# results and those for Java is the drastic reduction in the number of calls to local methods from constructors, 2.43% for C# compared with 13.18% for Java.

The valuable information gained from the C# analysis which was not able to be retrieved in the Java analysis is the breakdown of classes with local method calls based on whether those method calls are static or virtual dispatch. The analysis showed that only 0.26% of all classes contained a call to a method where a virtual, abstract or override declaration was found for that method. These are the method calls which would potentially miss their intended target in a construction environment different from Uniform Identity so the rare occurrence of these cases is valuable information. Across the corpus of 71,162 total classes, only 190 would need to be modified to mitigate potential construction issues under a new object initialisation environment.

5.2.1 Methods that couldn't be found

As with Java, there were some limitations to the accuracy of the analysis performed due to some of the limitations of pure source code analysis. The limits are most obvious in the final row of 5.3 where we find that 0.67% of classes contained a local method call where the destination of that call could not be found. This can occur when the analysis comes across a couple of cases.

- A constructor contains a non-local method call which is indistinguishable from a local method call without more information. An example of this is the use of C%'s `using static` syntax which imports the static functions from another namespace and a allows those functions to be called in a way that looks identical to a local method call. The `using static` pattern allows this code:

  ```
  \DIFadd{using System.Console;
  class Program : SuperProgram
  }{
     \DIFadd{static void Main()
     }{
        \DIFadd{Console.WriteLine("Hello world!");
     }}
  }
  ```

  to be changed to this:

  ```
  \DIFadd{using static System.Console;
  class Program : SuperProgram
  }{
     \DIFadd{static void Main()
     }{
        \DIFadd{WriteLine("Hello world!");
     }}
  }
  ```

  And without information about the contents of `System.Console`, it is not possible to determine that `WriteLine()` is a function defined in that namespace.

- A class extends a class in a pre-compiled library. If a call is made from a constructor to a local virtual or abstract method defined in a pre-compiled superclass then, unless that method has been overridden by the class being analysed, there is no way to know that the method was virtual or abstract.

5.3 JavaScript

In JavaScript, there are many ways developers make use of classical inheritance patterns despite the lack of native support in the language. This is largely a result of the numerous libraries which offer their own implementation of classical inheritance behaviour. Some examples of these patterns can be found in the following table:

Table 5.4: JavaScript Patterns

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

The JavaScript analysis in this study makes extensive use of the prior work in developing the JSClassFinder application [**?**]. The aim here is to find the cases where JavaScript developers are choosing not to use the native delegation support of the language and are instead modelling their programs with classical inheritance structures. The important factor here is the Class Usage Ratio (CUR) of a JavaScript project as defined in *Does JavaScript Software Embrace Classes? [**?**]*. Across a corpus of 50 JavaScript projects, the JSClassFinder returns interesting results about the prevalence of class usage in the language.

1. The median CUR across the corpus was 0.15
2. The upper quartile CUR across the corpus was 0.36
3. The lower quartile CUR across the corpus was 0.005, which was heavily impacted by 13 systems which had a CUR of zero

5.4 ~~Timeline~~Lua

~~The corpora for each of the remaining languages have been collected and the next stage is to analyse each to gather statistics which can be compared to those gathered for Java and JavaScript. The aim is to have a paper ready for the upcoming ECOOP paper submission deadline.~~ Table 5.5 shows a variety of patterns often representative of class usage and the percentage of files in the corpus which exhibit one or more of those patterns.
~~14~~

Often functions called class() will be created to encapsulate the `setmetatable()` logic which is used to create classes. It is also common to declare functions with the name "new" for use as constructors.

Table 5.5: Lua Analysis Results

Pattern	Test	Result	Percentage
= setmetatable()			
	Total matches	135	
	Files with matches	74	2.67%
class(			
	Total matches	487	
	Files with matches	380	13.69%
function something.new(			
	Total matches	31	
	Files with matches	30	1.08%
Union of all three			
	Total matches	653	
	Files with matches	473	17.04%

Chapter 6

Discussion

TODO - Possibly merged with results

Chapter 7

Conclusions

Some conclusions can be drawn from the data gathered thus far. This data covers the analyses of Java and JavaScript corpus data. It was found that there was a reasonable amount of crossover between the native support offered by each language and the patterns used by developers:

- In Java, where classical inheritance is natively supported, 6.09% of classes used forwarding patterns and 4.45% of classes used delegation patterns.
- In JavaScript, where delegation is natively supported, ~~36~~15% of the functions in the median project were used to emulate class or method behaviour.
- In Lua, where delegation is natively supported, 17.04% of all files contained patterns indicative of class behaviour.

From these numbers, it appears developers are more wiling to use inheritance structures, thus ignoring native delegation. Further research on other languages will help to clarify this claim.

Chapter 8

Limitations and Future Work

8.1 Limitations

There are some innate limitations to the data which can be gathered through static analysis of source code as carried out in this study. These limitations are largely a result of two factors. First, the inability to analyse code which is used by, but is not part of, the project; second, the disconnect between how often particular patterns are written (static frequency) and how often they are actually used during execution (dynamic frequency).

8.1.1 Inaccessible External Code

When only analysing source files, it is not possible to collect information about pre-compiled units which are used by those source files at runtime [?] . An example of where this can limit the effectiveness of the analysis in this study is the absence of information about calls dispatched to a superclass when that superclass is defined in a pre-compiled library. If a source file defines a class `A` which extends a class `B` where `B` is defined in a library with inaccessible source code, we cannot determine whether a local call in `A` to a method defined on `B` will be dispatched statically or virtually because we cannot see the method declaration. This could cause unforeseen changes to the behaviour of the methods on `A` if another class `C` were to extend `A` and override methods from `B` which we were previously unaware were virtually dispatched calls.

8.1.2 Static vs. Dynamic Frequency

It is generally not possible to determine whether any particular class or method is actually used in the execution of a program, or to determine which classes and methods are used more heavily than others. For example, we might prefer give different weight in our analyses to the patterns used in unit test files on the assumption that these are typically written with the expectation that they will rarely have to undergo structural changes after they are written, and will be executed less frequently than other core functionality.

8.1.3 Technical Limitations

The Mono project in the C# corpus had to be modified to allow the project to be analysed successfully due to technical constraints. As the Mono project sets out to provide an open source implementation of the necessary components of a C# compiler, it contained a copy of the entire .NET core library. This library was over 500MB in size so, after conversion to syntax trees through ANTLR, could not be reasonably analysed within 16GB of ram. For this reason, and because none of the other projects in this study or in [?] included copies of the source of their libraries, the library was removed from the Mono project. Every project across the corpora studied will have some dependency on a library, even if it is just the languages core libraries, so removing a copy of these libraries from Mono will ensure its comparability with the rest of the analysed data.

8.2 Future Work

There are a few other methods of software analysis which can work around the issues outlined above to provide a clearer understanding of a software project than analysis based solely on source code.

8.2.1 Analysis of Compiled Units

Java and C# both provide intermediate representations in the form of bytecode languages. For Java this is the Java Bytecode Language [?] and for C# this is Microsoft's Common Intermediate Language [?] . An analysis of these intermediate representations would be useful because they both offer varied instructions for method calls depending on whether the call is virtually or statically dispatched. This helps to overcome the limitation of being unaware of how a call will be dispatched when analysing source code only.

8.2.2 Dynamic Analysis

Analysing a program at runtime could provide useful information about how often particular patterns are used, as opposed to how often they are written. In JavaScript and Lua, a simple way to achieve this would be to modify common class libraries to add counters which record how often classes are created, modified or instantiated. In Java, a program called JVM Monitor [?] could be used to determine the invocation counts of methods which are of interest.