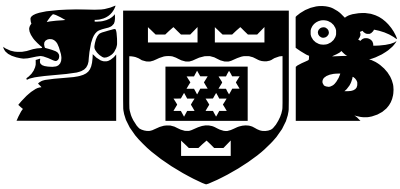
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui



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: Dr. Alex Potanin, Prof. James Noble &
Tim Jones

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

Abstract

This report presents 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 representing delegation and inheritance in languages which support their implementation followed by a comparison of their use.

Acknowledgments

I would like to acknowledge a number of people who have helped and supported me throughout this project.

First, I would like to express my gratitude to Dr. Alex Potanin, Prof. James Noble, and Timothy Jones who supervised this project throughout the year. Their support and guidance has been invaluable to this project, and they have always been willing to help when I need it.

Secondly, I would like to thank my friends and family who have supported me through the year. Especially to my Mum, who has offered endless support throughout my studies. I am thankful to Brianna, who has helped me get through this final year. She was always being available to proof read my work, and to get my head straight when I need it. I am also grateful to my flatmates, Lucy, Mohana, and Ged, who made sure I ate and slept well, despite all the work.

Thirdly, I would like to thank my classmates. Jack and Glen, in particular, have helped me more than they could know through the last few years, always offering feedback on my work and helping out when I need it.

Contents

Chapter 1

Introduction

The aim of this study is to determine the usefulness of delegation in modern programming languages and to explore replacing classical object inheritance models in existing software projects with that of delegation. A clearer understanding must be formed detailing the use of delegation and classical inheritance in real world software development projects. These projects must cover languages with varying native support for the object inheritance models. In order to produce empirical evidence of the frequency at which these structures are utilised in typical software projects, the corpora of existing projects are examined.

1.1 Motivation

Code reuse mechanisms are a vital aspect of software development. These mechanisms allow engineers to write code once and make use of the code in various places without duplicating it. Code reuse aims to reduce the time and resources required to produce a software system by maximising the use 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.
This study investigates the use of two common forms of code reuse found in object oriented software systems:

- 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 is optimised for different scenarios 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?". This question will be investigated by studying the extent to which developers make use of delegation in their software projects and whether this might increase with stronger language-level support. The results can inform the design of new programming languages which must, to some extent, make a choice between classical inheritance or a delegation based object model.

The next stage is to investigate software written under a classical inheritance model to determine which components of their implementation are dependent on the model. This will provide a collection of examples which would be difficult to reimplement under a delegation model. From these examples, an investigation can be carried out to determine how much effort it would be expected to take to move these projects to a delegation 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

In a 2009 paper titled *Are we Ready for a Safer Construction Environment [6]*, 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. Gil and Shragai then cover 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.

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 [6]. 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 [7] 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. [7]*

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".

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

This can cause issues in some cases because of the order in which constructors are run during the creation of an object. A top down approach is taken for creating an object where the constructor on the least derived type will execute first and then the constructor on each class down the hierarchy to the most derived type will execute. This means that if the `m()` method on `B` was dependent on some state set in `B`'s constructor, that state would not yet be initialised when the invocation was dispatched from the constructor of `A`.

2.2 Delegation

Henry Lieberman's 1986 paper *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems [12]* 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." [12]*

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.

The paper *Object Inheritance Without Classes [10]* by Tim Jones et al 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 characteristics of some Java programs make them 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 which follows a different model of object initialisation. Additionally, this information shows which patterns could be rewritten under other inheritance models without requiring much modification and, in some cases, more concisely.

2.3 JavaScript Analysis

Does JavaScript Software Embrace Classes? [19] 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 "Class Usage Ratio" metric 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 [19].

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 [4] [19]. Analysing code which was created after the addition of native class support could no longer be considered to be a representation of class usage in a language offering only delegation natively.

2.4 Java Analysis

Understanding the Shape of Java Software [2] 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 [5] 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. [5]*

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 [22] 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. Any virtually dispatched method on a non-final class can potentially perform a downcall at runtime.

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 [21]* 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.

2.5 C# Analysis

The interesting aspect of the C# programming language for the purposes of this study is the mandatory inclusion of the `virtual` keyword to allow for dynamic dispatch of a method to occur at runtime. In contrast, Java dispatches methods dynamically by default and requires the inclusion of a `final` modifier to declare a statically dispatched method. Due to optimisations in the JVM whereby it can still inline any virtual method for which no class with an overriding method has been loaded through the class loader, developers who omit the `final` keyword will likely also avoid the performance penalty associated with dynamic dispatch [18]. While there are some Java projects which make use of the `final` keyword where possible as a matter of convention, many do not, so relying on the keyword for metric calculation would not provide meaningful data.

As a result of the requirement of the `virtual` keyword, C# integrated developer environments are able to offer useful warnings about questionable practices involving the use

of virtually dispatched methods. One of these warnings discourages developers from making calls to virtual methods from constructors of non-sealed class. This rule is outlined in Microsoft Developer Network's usage warning *CA2214: Do not call overridable methods in constructors [16]* and is discussed by Eric Lippert in *Why Do Initializers Run In The Opposite Order As Constructors? [14]*.

As Lippert describes, *"Calling methods on derived types from constructors is dirty pool, but it is not illegal [14]."* These warnings exist because of the way object initialisation occurs in C#. Following the Uniform Identity model, an object's constructors are run from the top of the inheritance hierarchy to the bottom. This means that when a constructor calls to a virtual method which has been overridden on a more derived type, the constructor for that more derived type has not been invoked yet. This means the method is likely to be dispatched to a derived type's implementation which is dependent on vital state of that derived type which has not yet been initialised.

2.6 Software Metrics

Towards a Metrics Suite for Object Oriented Design [3] 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.

2.7 Analysing Corpora

The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies [20] 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.

The Use and Limitations of Static-Analysis Tools to Improve Software Quality [1] provides an outline of some of the limitations which will likely be encountered in this study as a result of analysing corpora rather than runtime execution of projects. Through analysis of source code alone it is not possible to gather information about the dynamic frequency, or frequency of execution, of particular patterns in the analysed code. Corpus analysis also restricts access to information which only becomes available after some number of compilation steps. This means that declarations found in libraries of pre-compiled code are inaccessible to static source code analysis.

Chapter 3

Code Patterns

To determine how developers are using delegation and inheritance, the first stage is to define a clear way of identifying the behaviour of each pattern. 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, calls to `this.f(...)` are passed to some `other.f(...)`, transferring any necessary information as call parameters.

In the following example, a `Square` object is forwarding responsibility for its area calculation to the `SquareAreaCalculator`. The `SquareAreaCalculator` could be shared by many `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`.

```
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, detecting this behaviour involves searching for patterns where an object contains method which does very little work besides forwarding the call to a 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 the forwardee needs to be capable of responding correctly to requests from other forwarders without influence from program 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 is often described as forwarding `this.f(...)` calls to `other.f(...)` **on behalf of this.** That is, dispatch the call to `other.f(...)` but have the self reference within that call point back to my self reference. This is explained in further detail in Section 2.1.

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 `SquareAreaCalculator` 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;}
}
```

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.

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` 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`.

```
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 Merged Identity, as used in C++, can also model the behaviour fairly closely. 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

Methodology

4.1 Selecting Languages

The languages explored in this study include Java, C#, JavaScript and Lua. These languages were were selected because they each have large enough open source communities to gather meaningful corpora of projects and, between them, they offer a wide range of native object inheritance model implementations. They also provide a helpful division between two languages with native support for classical inheritance object models and two languages with native support for delegation object models.

Java and C# both offer native implementations of classical inheritance so can be used to analyse the frequency of use of patterns which surround this classical inheritance model. These languages also allow an analysis of patterns which would behave differently under a delegation object model, representing cases which could be difficult to reimplement in another language with different native support.

JavaScript and Lua both offer delegation natively so can provide a measure of how often developers are making use of these delegation features. They also show how often developers in these languages are choosing to ignore the languages' native features in favour of classical inheritance models.

4.2 Assembling Corpora

To analyse the use of 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 [20]. Likewise, with JavaScript, we have adopted an existing corpus used by the team that developed JSClassFinder [19].

For Lua and C#, quality existing corpora could not be found so we had to build our own. For each of these languages, the top 25 open source projects were sourced from GitHub's "Trending this month" list as of 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 contain repositories which are currently active so likely make use of newer language features.

4.3 Static Analysis Tools

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:

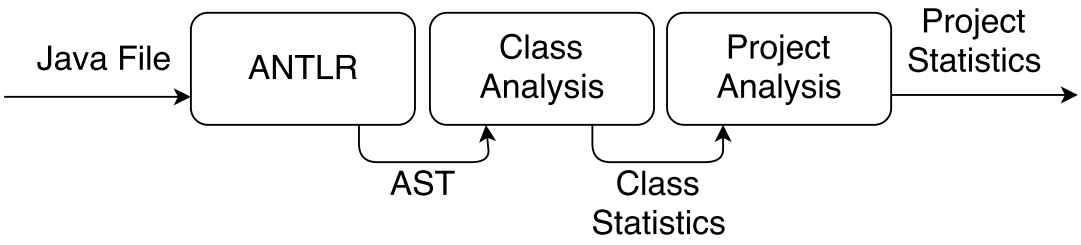
- `grep` is used to perform regular expression searches on files. This can detect some of the more simple patterns explored in this paper. More specifically, an implementation of `grep` known as PCREGrep (Perl Compatible Regular Expression `grep`) was used to allow multiline analysis which is a requirement of a source code corpus analysis.
- ANTLR (Another Tool For Language Recognition) is a tool which accepts a language grammar as input and produces a lexer and parser. This lexer and parser can then accept a file which conforms to the grammar definition and construct a syntax tree to represent that file.
- JSClassFinder is a tool which detects patterns indicative of class and method declarations in JavaScript projects. It accepts JSON representations of the syntax trees of JavaScript files as input and produces the Class Usage Ratio of the syntax tree as its output [19].
- Esprima accepts a JavaScript code file as input and produces a JSON representation of the syntax tree of that file as output. This JSON file can then be used as the input to the JSClassFinder tool.

4.4 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



4.5 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. Use ANTLR, along with a C# preprocessor grammar, to perform the first stage towards forming a syntax tree. This stage evaluates preprocessor directives in the program to ensure the remaining file can be transformed into a well formed syntax tree. Included in this stage is the removal of `#region` tags and conditional directives which exclude and include blocks of source code based on boolean arguments.

2. Use ANTLR, along with a C# program grammar, to create a syntax tree for each C# code file in the corpus and traverse it to find all class declaration subtrees. Collect these class declarations to be explored in later steps.

3. 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.

4. 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.

4.5.1 Outliers

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 [20] included copies of the source of their libraries, the .NET core 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.

An interesting outlier file was found during the C# analysis carried out in this study. A file titled `T_1247520.cs` is defined in the Roslyn compiler project and is used in testing scenarios. This file contains 10,020 class definitions, each with no superclass, no subclasses, and no method or constructor definitions. The file is found in a `Test` directory so the assumption is that it is used as a compiler stress test. This file has been left in the corpus for the results found in Table 5.3 because I believe removing it would be making the disingenuous claim that all the file in the other corpora had been investigated to ensure they had no similar outliers. Despite this, it is interesting to see the changes to results when these class definitions are removed, thereby reducing the number of classes found by 10,020 but leaving the remaining counts unchanged.

- The percentage of classes which extend another class increases from 26.87% to 31.24%.
- The percentage of classes which are extended by another class increases from 12.66% to 14.72%.
- The percentage of classes which make calls from local methods from a constructor increases from 2.47% to 2.87%.
- The percentage of classes with calls to local virtual, override, and abstract methods from their constructors increases proportionately with the change in all calls to local methods from constructors.

4.6 JavaScript Analysis

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 4.1: 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(...)`

As a result of the wide ranging methods of implementing classical inheritance in JavaScript, the effort required to do so accurately is great. Prior work in the field made the JavaScript analysis in this study possible, primarily the existence of tool JSClassFinder [19]. The JavaScript analysis in this study consisted mainly of a recreation of the JSClassFinder study. JSClassFinder is a tool created by a team of researchers to analyse the extent to which JavaScript developers use classes in their projects.

4.7 Lua Analysis

The Lua corpus was analysed with `grep` to identify code patterns and keywords associated with class usage. There exists a variety of patterns used to implement classical inheritance in Lua as described by the Lua-Users Wiki [15]. The analysis in this study attempts to uncover the proportion of the Lua corpus which is making use of object oriented paradigms and, to achieve this, analyses the code of each file to detect the particular patterns found in the object orientation tutorial.

The first of these patterns is the presence of `identifier = setmetatable()` in a Lua program. The `setmetatable()` function is the core of all suggested object orientation implementations so the detection of this pattern is vital. Unfortunately, while the use of this function is typically considered necessary for object orientation to exist in a Lua program, the pattern is often encapsulated in a function of a different name which makes the actual extent of

object orientation usage more difficult to measure. In response to the practice of encapsulation of these patterns, this study has also included measures of the presence of two function names which are typically used to wrap these classical inheritance behaviours. These are the `class()` and `new()` functions.

4.8 Evaluation

The accuracy of the results of this study can be measured in terms of false positives against true positives and false negatives against true negatives. These can each be measured in different ways, but the process of testing for false positives is easier and more reliable than testing false negatives.

4.8.1 False Positives

False positives occur when the algorithms used to detect patterns in the corpora claim to have found an instance of that pattern, but the code does not match the definition exactly. Detecting false positives in the data can be achieved by investigating the results returned as matches against the patterns in the analysis to determine whether the are, in fact, exhibiting the behaviour for which the pattern is searching. In cases where the algorithms used to detect the patterns could be modified to ignore the false positives, they were. Despite this, there were some cases remaining where the search algorithms could not be easily fixed. Examples of these are as follows.

- False positives may be returned for extendedness when the interface naming assumption explained in section 5.2.4 fails and an interface is named in a way that makes it indistinguishable from a class.
- False positives may be returned for extendedness when the unique class name assumption explained in section 5.1.2 fails as classes may be considered extended when another class of the identical name is extended.
- False positives for delegation can occur when developers use code patterns which look identical to those of delegation but are modelling different intents. This is explained in more detail in section 5.1.1.

4.8.2 False Negatives

False negatives occur when the algorithms overlook code segments in the corpora which fulfil the criteria to be defined as an instance of the patterns under investigation. Examples of these cases are as follows.

- There is room for false negatives in the C# statistics on calls to local methods from constructors. The accuracy of this data is limited by the problem of inaccessible external code and the `using static` pattern as explained in section 5.2.3. This is reflected in the final row of the table which measures the number of calls from constructors where the receiver of the call either could not be found or was another form of method call which was indistinguishable from a local method call.
- False negatives for subclassing in C# can exist if a class is defined with a name which fulfils the requirements of the naming convention for interfaces as discussed in section 5.2.4.

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){`
	`  return identifier[.identifier]*.name(anything);`
	`}`

Call Delegation	`Anything name (anything){`
	`  return identifier[.identifier]*.name(this);`
	`}`

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

We used two patterns to represent delegation in code samples 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. The second, constructor delegation, is where a delegatee is constructed specifically for the instance of the delegator by passing `this` as a constructor argument. This delegatee can then act on that constructor argument when its other methods are called.

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
Projects	112		
Classes	116427		
Extending Classes	71203	61.16%	
Extended Classes	20751	17.82%	
Classes with forwarding	7087	6.09%	
Classes with forwarding that extend another class	3381	2.90%	
Classes with local method calls in constructors	16101	13.83%	
Classes storing this in constructors	2392	2.05%	
Classes with local method calls or storing this in constructor	17099	14.69%	
Extended classes with local method calls 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

As discussed in Section 4.8.1, the imprecise definition of what it means to model delegation in Java will mean 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 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. For example, a instance of a `Square` class 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:

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

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

```
void addSquare(){
   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.

5.1.2 Unique Class Name Assumption

As the Java analysis in this study was performed on source code alone, there are cases of ambiguous class names which would only be disambiguated through full namespace analysis. Because of this, the results are dependent on classes in a package being named uniquely. In cases where this assumption does not hold there is a risk classes being considered to be extended when they are not. More specifically, the question "Is class A extended by another class" will evaluate to true when any class in the package with the name "A" is extended by another class. This limits the accuracy of the "Classes that are extended by another class" metric in Table 5.2

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		
Classes	71539		
Extending Classes	19222		26.87%
Extended Classes	9057		12.66%
Methods	222121		
Virtual Methods	10721	4.83%	
Override Methods	26760	12.05%	
Delegates	738		
Classes with calls to local methods in constructors	1764		2.47%
Classes with calls to local virtual methods in constructors	48		0.07%
Classes with calls to local override methods in constructors	55		0.08%
Classes with calls to local abstract methods in constructors	15		0.02%
Classes with calls to methods that could not be found in constructors	616		0.86%

The full C# corpus statistics are broken down for each project and have been included as appendix A.

5.2.1 Fewer Local Method Calls from Constructors

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, 2.47% for C# compared with 13.18% for Java. There are likely a variety of reasons for this drastic reduction, but of note is that the Microsoft Developer Network Blog strongly discourages this practice [14], and the widely used Visual Studio extension ReSharper provides IDE warnings against the practice [9] as discussed in Section 2.5.

5.2.2 Virtual Calls from Constructors are Rare

The valuable information gained from the C# analysis which, as discussed in Section 2.5, could not be retrieved in a meaningful form from 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 0.07% 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.

5.2.3 Methods that could not 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 table 5.3 where we find that 0.86% 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 the following code:

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

to be replaced with this implementation:

```
using static System.Console;
class Program : SuperProgram
{
   static void Main()
   {
```

```
        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.

While the number of methods which were not found is high at 616, making up 34.9% of all local method calls in constructors, the remaining 1148 methods which were found were largely without virtual, abstract or override modifiers. Of those 1148 methods only 118, or 10.3%, had one of these modifiers.

5.2.4 Superclasses and Interfaces

In Java, source code alone clearly identifies the distinction between extending a class and implementing an interface. An inheritance relationship is written as

```
class Square extends Rectangle { }
```

whereas indicating that a class implements a given interface is written as

```
class Square implements Shape { }
```

In C#, however, this distinction cannot be determined from source code alone as the syntax elements for both are identical.

```
class Square : Rectangle { }
class Square : IShape { }
```

To counteract this, the analysis is dependent on the members of the corpus following the C# naming conventions for interfaces as outlined by Microsoft [17]. This study considers any base type to be an interface when the name of that type is a capital "I" character followed by any other uppercase character.

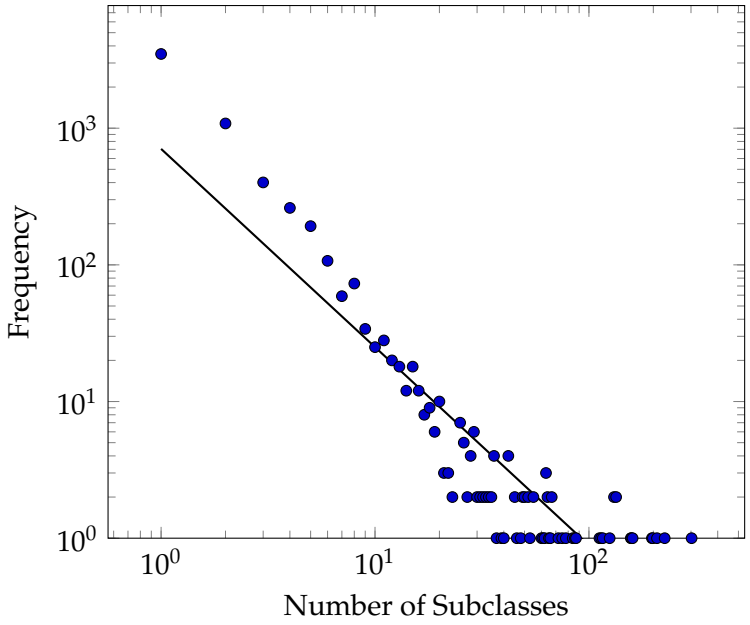
5.2.5 Unique Class Name Assumption

The C# results are dependent on the same assumption of unique class names as Java. The reasons for this match those explained in section 5.1.2.

5.3 Distribution of Number of Subclasses

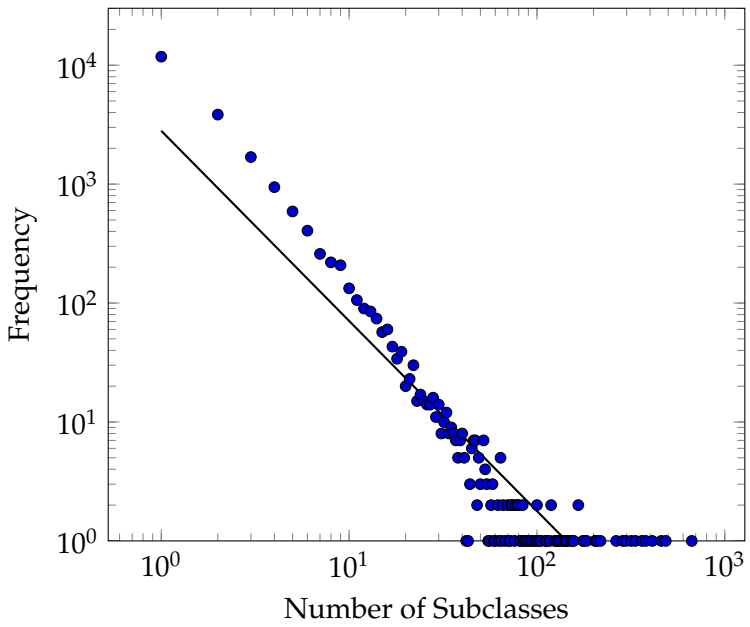
The C# analysis conducted as part of this study resulted in an interesting finding relating to the distribution of the number of subclasses across the classes in the corpus. The analysis revealed that the number of subclasses of extended C# classes approximates a power law distribution, which can be seen in the straight line when the data is rendered onto a log scale. This distribution can be seen in the following chart:

Figure 5.1: Distribution of Number of Subclasses in C# (Log Scale)



The C# subclassing distribution in 5.2 matches the distribution discovered through the investigation in *Understanding the Shape of Java Software* [2] and reproduced in this study. The distribution found in the analysis of the Java corpus can be found in the following chart:

Figure 5.2: Distribution of Number of Subclasses in Java (Log Scale)



.

5.4 JavaScript

The JavaScript analysis in this study makes extensive use of Silva et al's work in developing the JSClassFinder application [19]. 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? [19]*. 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

This indicates that, in the median project in the JavaScript corpus, 15% of all functions are modelling some form of class or method behaviour. A value of this magnitude shows that the use of classical inheritance in JavaScript is highly prevalent despite the language's lack of native support for its implementation.

5.5 Lua

Table 5.4 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.

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 5.4: 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%

These results show that the proportion of Lua developers making use of classical inheritance patterns is high, with around 17% of all Lua files in the examined corpus presenting class-like behaviour of some kind.

Chapter 6

Conclusion

We found that differences between the native support offered by each language and the patterns used by developers of projects in that language were fairly frequent:

- In Java, where classical inheritance is natively supported, 6.09% of classes used forwarding patterns and 4.45% of classes used delegation patterns.
- The C# analysis showed that the vast majority of classes could be reimplemented in a delegation language with minimal need for modification. Just 0.07% of classes contained constructor patterns which would exhibit unexpected behaviour without Uniform Identity.
- In JavaScript, where delegation is natively supported, 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 willing to use classical inheritance structures, thus ignoring native delegation. It is difficult to determine whether this is because classical inheritance is necessary for aspects of the projects or if its use is simply more common because developers find it more comfortable.

The further findings of this study involve a measurement of the difficulty involved in reimplementing projects built for classical inheritance into a language built for delegation. The main issues for this reimplementation are constructor patterns which are dependent on uniform identity. It was found, through the Java corpus analysis, that 13.83% of classes made local calls from constructors and 2.05% stored `this` from a constructor. Through the C# corpus analysis, it was found that only around 10.3% of calls to local methods from constructors are dispatched virtually at runtime. In C#, only 0.17% of classes contained a call from a constructor to a dynamically dispatched method.

6.1 Limitations

There are some innate limitations to the data which can be gathered through pattern matching against 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) [1].

6.1.1 Inaccessible External Code

When only analysing source files, it is difficult to collect information about pre-compiled units which are used by those source files at runtime [1]. 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 code file defines a class `A` which extends a class `B` where `B` is defined in a library with precompiled or otherwise inaccessible source code, we cannot see the details of the methods defined in `B`. This becomes an issue because it is no longer possible to determine whether a local call in `A` which targets a method defined on `B` will be dispatched statically or virtually because we cannot see the method declaration in `B`. 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.

6.1.2 Static vs. Dynamic Frequency

It is difficult, and in some cases impossible, 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 frequently at runtime than others. For example, we might prefer give a different weighting 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 need to undergo structural changes after they are written. The are also expected to be executed less frequently than other core functionality in general operation of the program.

6.1.3 Incomplete Source Files

There were cases of files in the JavaScript corpus which could not be parsed in isolation to form a valid syntax tree because they were not syntactically valid source. One example of this was a file which consisted of the majority of a valid JavaScript file but stopped short of the end, with a few other files in the same directory offering several different options for the final part of the code. The assumption here was that the files would be opened and appended at runtime but this was not possible to replicate in a study which operates on source code alone. This study ignored files which could not be compiled on their own, but it is possible that data was lost by ignoring files which expect to be concatenated to create valid programs.

6.2 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.

6.2.1 Analysis of Compiled Units

Analysis of compiled units would help to mitigate the issues explained in section 6.1.1 where precompiled libraries are inaccessable to the analysis. Java and C# both provide intermediate representations in the form of bytecode languages. For Java this is the Java Bytecode Language [13] and for C# this is Microsoft's Common Intermediate Language [8]. 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.

6.2.2 Analysis of Dynamic Frequency

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 commonly used libraries associated with classical inheritance implementations to add counters which record how often classes are created, modified or instantiated. In Java, a program called JVM Monitor [11] could be used to determine the invocation counts of methods which are of interest. This would allow code which is core to the functionality of the program to be weighted more heavily and code which is run relatively infrequently to be weighted more lightly.

6.2.3 Measuring Intent

Ideally, an empirical study of this nature would come with measures of the precision and recall of the algorithm. For this study, these measures would be dependent on comparing the patterns detected by the algorithms against the intent of the developer. Unfortunately, there is no easy way to accurately differentiate between a developer's intent and the patterns they use in their code. This issue is discussed in *Does JavaScript Software Embrace Classes?* which describes the creation of the JSClassFinder tool [19]. Even when manually inspecting files and analysis results, it is often not possible to determine whether the result of the analysis is truly indicative of the developer's intent. For this reason, it is only possible to approximate the values of precision and recall based on some assumptions about whether certain patterns are, in fact, representative of the behaviours under analysis. Despite this, having estimated ranges of these precision and recall values could help to improve confidence in the research so would be valuable future work.

Bibliography

[1] ANDERSON, P. The Use and Limitations of Static-Analysis Tools to Improve Software Quality. *CrossTalk: The Journal of Defense Software Engineering 21*, 6 (June 2008), 18–21.

[2] BAXTER, G., FREAN, M., NOBLE, J., RICKERBY, M., SMITH, H., VISSER, M., MELTON, H., AND TEMPERO, E. Understanding the Shape of Java Software. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 397–412.

[3] CHIDAMBER, S. R., AND KEMERER, C. F. Towards a Metrics Suite for Object Oriented Design. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications* (New York, NY, USA, 1991), OOPSLA '91, ACM, pp. 197–211.

[4] ECMA INTERNATIONAL. *ECMAScript 2015 Language Specification, Standard ECMA-262*, 6th ed. ECMA International, 2015.

[5] GIL, J. Y., AND MAMAN, I. Micro Patterns in Java Code. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2005), OOPSLA '05, ACM, pp. 97–116.

[6] GIL, J. Y., AND SHRAGAI, T. Are We Ready for a Safer Construction Environment? In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming* (Berlin, Heidelberg, 2009), Genoa, Springer-Verlag, pp. 495–519.

[7] GOSLING, J., JOY, B., STEELE, JR., G. L., BRACHA, G., AND BUCKLEY, A. *The Java Language Specification, Java SE 7 Edition*, 1st ed. Addison-Wesley Professional, 2013.

[8] HAMILTON, J. Language Integration in the Common Language Runtime. *SIGPLAN 38*, 2 (Feb. 2003), 19–28.

[9] JETBRAINS. Resharper 2016.1 help :: Code inspection: Virtual member call in constructor.

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

[11] JVM MONITOR PROJECT. The Java Virtual Machine Monitor Specification.

[12] 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.

[13] LINDHOLM, T., YELLIN, F., BRACHA, G., AND BUCKLEY, A. *The Java Virtual Machine Specification, Java SE 7 Edition*, 1st ed. Addison-Wesley Professional, 2013.

[14] LIPPERT, E. Why Do Initializers Run In The Opposite Order As Constructors? Part Two. *Fabulous Adventures In Coding* (2008).

[15] LUA USERS ORGANISATION. Lua-Users Wiki: Object Orientation Tutorial.

[16] MICROSOFT DEVELOPER NETWORK. CA2214: Do not call overridable methods in constructors.

[17] MICROSOFT DEVELOPER NETWORK. Names of Classes, Structs, and Interfaces.

[18] ORACLE CORPORATION. The Java HotSpot Performance Engine Architecture.

[19] 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.

[20] 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. In *2010 Asia Pacific Software Engineering Conference* (Nov 2010), pp. 336–345.

[21] TEMPERO, E., NOBLE, J., AND MELTON, H. How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming* (Berlin, Heidelberg, 2008), ECOOP '08, Springer-Verlag, pp. 667–691.

[22] 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.

Appendix A

C# Analysis by Project

Project	AutoMa	CodeHu	corefx	dapper-	EntityFr	ILSpy	mono	MonoG	msbuild	Mvc	Nancy	NecroB	Newton
Classes	2260	444	23292	281	5108	846	1387	1260	1310	2545	1042	147	1114
Extending Classes	654	288	5580	68	1271	336	250	564	424	781	323	8	500
	28.9%	64.9%	24.0%	24.2%	24.9%	39.7%	18.0%	44.8%	32.4%	30.7%	31.0%	5.4%	44.9%
Extended Classes	642	43	5243	7	439	74	167	93	98	163	41	0	79
	28.4%	9.7%	22.5%	2.5%	8.6%	8.7%	12.0%	7.4%	7.5%	6.4%	3.9%	0.0%	7.1%
Methods	2098	1023	75298	686	13159	3107	3377	4550	8426	8331	4338	300	3180
Virtual Methods	8	39	3110	26	4491	51	60	67	49	360	131	0	69
	0.38%	3.81%	4.13%	3.79%	34.13%	1.64%	1.78%	1.47%	0.58%	4.32%	3.02%	0.00%	2.17%
Override Methods	349	383	8321	35	2800	576	78	914	354	411	369	16	418
	16.63%	37.44%	11.05%	5.10%	21.28%	18.54%	2.31%	20.09%	4.20%	4.93%	8.51%	5.33%	13.14%
Delegates	5	0	241	2	0	1	107	136	27	5	0	1	2
Classes with calls to local methods in constructors	22	41	374	4	52	46	13	93	66	39	147	4	14
	0.97%	9.23%	1.61%	1.42%	1.02%	5.44%	0.94%	7.38%	5.04%	1.53%	14.11%	2.72%	1.26%
Classes with calls to local virtual methods in constructors	1	2	10	0	6	0	0	2	0	0	5	0	0
	0.04%	0.45%	0.04%	0.00%	0.12%	0.00%	0.00%	0.16%	0.00%	0.00%	0.48%	0.00%	0.00%
Classes with calls to local override methods in constructors	0	0	24	0	9	0	0	6	0	0	0	0	0
	0.00%	0.00%	0.10%	0.00%	0.18%	0.00%	0.00%	0.48%	0.00%	0.00%	0.00%	0.00%	0.00%
Classes with calls to local abstract methods in constructors	0	1	0	0	1	0	0	1	0	1	0	0	0
	0.00%	0.23%	0.00%	0.00%	0.02%	0.00%	0.00%	0.08%	0.00%	0.04%	0.00%	0.00%	0.00%
Classes with calls to methods that could not be found in constructors	2	30	98	0	31	21	0	35	15	11	78	1	11
	0.09%	6.76%	0.42%	0.00%	0.61%	2.48%	0.00%	2.78%	1.15%	0.43%	7.49%	0.68%	0.99%
Project	OpenR	Opserve	PowerS	Psychso	PushSh	RestSh	roslyn	Service	ShareX	SignalR	Sparkle	Wox	
Classes	1710	655	2540	7	101	203	19475	4528	261	759	99	165	
Extending Classes	621	157	1362	0	27	31	4069	1422	94	277	55	60	
	36.3%	24.0%	53.6%	0.0%	26.7%	15.3%	20.9%	31.4%	36.0%	36.5%	55.6%	36.4%	
Extended Classes	74	68	257	0	3	8	1229	238	14	54	13	10	
	4.3%	10.4%	10.1%	0.0%	3.0%	3.9%	6.3%	5.3%	5.4%	7.1%	13.1%	6.1%	
Methods	5059	827	7919	43	133	682	64385	10887	850	2642	392	429	
Virtual Methods	160	14	216	0	1	33	1020	617	41	146	6	6	
	3.16%	1.69%	2.73%	0.00%	0.75%	4.84%	1.58%	5.67%	4.82%	5.53%	1.53%	1.40%	
Override Methods	589	103	1372	0	8	22	7814	1239	186	296	67	40	
	11.64%	12.45%	17.33%	0.00%	6.02%	3.23%	12.14%	11.38%	21.88%	11.20%	17.09%	9.32%	
Delegates	72	0	43	0	3	0	34	6	2	3	44	4	
Classes with calls to local methods in constructors	132	5	99	0	4	4	364	95	44	43	33	26	
	7.72%	0.76%	3.90%	0.00%	3.96%	1.97%	1.87%	2.10%	16.86%	5.67%	33.33%	15.76%	
Classes with calls to local virtual methods in constructors	2	0	1	0	0	0	3	8	8	0	0	0	
	0.12%	0.00%	0.04%	0.00%	0.00%	0.00%	0.02%	0.18%	3.07%	0.00%	0.00%	0.00%	
Classes with calls to local override methods in constructors	0	0	0	0	0	0	4	5	2	1	4	0	
	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.02%	0.11%	0.77%	0.13%	4.04%	0.00%	
Classes with calls to local abstract methods in constructors	0	0	1	0	0	0	7	2	0	0	1	0	
	0.00%	0.00%	0.04%	0.00%	0.00%	0.00%	0.04%	0.04%	0.00%	0.00%	1.01%	0.00%	
Classes with calls to methods that could not be found in constructors	33	2	45	0	0	0	99	25	24	11	24	20	
	1.93%	0.31%	1.77%	0.00%	0.00%	0.00%	0.51%	0.55%	9.20%	1.45%	24.24%	12.12%	