What Programmers do with Inheritance in Java and C#

B.L. Brekelmans

Master's Thesis

20-10-2014

Master Software Engineering

Universiteit van Amsterdam

Supervisor: dr. Tijs van der Storm

Centrum voor Wiskunde en Informatica

Abstract

Inheritance is a widely used concept in modern object oriented software engineering. Previous studies show that inheritance is widely used in practice yet empirical data about how it is used in practice is scarce. An empirical study into this subject has been done by Tempero, Yang and Noble titled “What Programmers do with Inheritance in Java” [1]. This study replicates and extends the study by Tempero et al through inclusion of C# and explanation of the differences and similarities between the languages with respect to practical use of inheritance. It contributes towards the validation and broadening of original conclusions. This study presents a comparative analysis of 169 open source C# and Java systems totalling around 23 million lines of code. Interesting findings are presented on the potential effects of forbidding implicit dynamic binding and inferring types for local variables on the practical use of inheritance amongst C# and Java open-source systems.

Acknowledgement

I would like to thank dr. Tijs van der Storm at the Centrum for Wiskunde & Informatica (CWI) for his excellent supervision and advice. Additionally, I would like to thank Ewan D. Tempero, Hong Yul Yang and James Noble for their interesting study on the use of inheritance, used as the basis and source of inspiration for this Master's thesis. I am also grateful to Cigdem Aytekin, who helped in validating and understanding the subject matter for this study, as well as reviewing this document. Lastly, I would like to thank Laurens Knoll for his work in reviewing this document.

Contents

1 Introduction

Inheritance is widely supported by general-purpose languages such as C# and Java. How it is used in practice however remains an open question. Tempero, Yang and Noble present a model for empirical research on practical use of inheritance in their study titled “What Programmers do with Inheritance in Java”. They apply their model to an empirical investigation of 93 Java open source systems, supplemented by a longitudinal analysis of 43 versions of two systems. Their findings indicate that subtyping is the dominant use of inheritance, while code reuse is also prominent. This study aims to investigate their findings for the purposes of verification and application to the C# language. 86 open source Java systems and 83 open source C# systems are investigated in this study using quantitative source code analysis.

The structure of this study is inspired by the model for replication studies proposed by Carver [2]. Section 2 discusses the motivation and relevance of this replication study. A concise summary of the original study’s motivation, research questions, study details, results and conclusions is presented in section 3. Note that the model the original study uses to report findings is also used in this replication study and has slightly different parameters, therefore this model is discussed in a later section (6.1). The discussion related to the original study is integrated with the discussion of this study, presented in section 9. Because this study also investigates C#, differences in programming language between C# and Java are discussed in detail in section 4. This section does not cover all language differences, only those relevant to the purposes of investigating inheritance usage. Section 5 discusses the replication study in more detail by defining the research questions and presenting detailed information and discussion related to the changes made to the original study. In section 6, the research method used for this study is discussed. Since the main purpose of investigation remains the same, the research method is very similar to the original study. The technical implementation and systems investigated are different however.

A presentation and analysis of results are detailed in a comparative fashion in section 7, following the reporting structure of the original study but including results from the C# and Java replication. Section 8 analyses the similarities and differences found, and further investigates some of these differences. The research method, results and analysis are discussed in section 9, where numerous threats to validity are presented. Section 10 presents the conclusions related to the research questions. This study shows reduced usage of inheritance patterns investigated in this study for the C# systems, while the Java replication shows generally similar results as the original study. Section 11 wraps up this study by discussing possible avenues for future work.

2 Problem statement & context

Inheritance is an important concept in object-oriented software engineering. A significant portion of educational material teaching object-orientation covers the concepts of inheritance, books such as *Head First C#* [4]*, Head First Java* [5] *and Learning Object Oriented Programming in C#* [6] each contain multiple chapters devoted to explaining the concepts of inheritance. An empirical study by Tempero et al [7] shows that inheritance is widely used in practice, around three quarters of the classes in the Java open source systems they investigated participate in an inheritance tree.
To determine if using inheritance is 'a good thing', the effect of inheritance on the maintainability and extensibility of a system has been investigated by previous studies. Several empirical studies were done on the effect of inheritance on the maintainability of systems. Harrison et al, [7] Daly et al, [8] and a replication study by Cartwright and Shepperd [10] each investigated the effect of inheritance on modifiability through controlled experiments. Students were tasked with making changes to small (400-1200 lines of code) C++ systems or answering questions about how the code works. The study by Daly et al [8] reports that systems using inheritance require less time to modify, while the study by Cartwright and Shepperd [10] reports the opposite. Cartwright and Shepperd further conclude that inheritance usage makes code harder to modify, but that using inheritance makes changes more compact. Harrison et al [7] report that code without inheritance is easier to modify and understand. Two controlled experiments by Prechelt et al [11] found that programs containing higher levels of inheritance took longer to maintain than programs with lower levels of inheritance. Cartwright and Shepperd [12] investigated a single system of 133.000 lines of C++ code, suggesting increased defect density for code that uses inheritance. However, they report an average of 3500 lines of code per class (the highest found in this study is 250), leading one to wonder about the relevance of these results in current code.

Having determined that inheritance appears to be a critical part of object-oriented programming with widespread use in practice, it would be interesting to investigate *how* it is used. Several metrics have been defined related to the use of inheritance. For example, the *Depth of Inheritance Tree* (DIT) and *Number of Children* (NOC) metrics defined by Chidamber & Kemerer [13] have been used extensively in empirical research. The Specification Ratio (SR) and Reuse Ratio (RR) metrics devised by Henderson-Sellers [14] provide insights into the nature of the inheritance tree. However, these metrics merely count classes and the inheritance relationships among them, providing no information about the specific kinds of inheritance actually used. Taivalsaari [15] and Meyer [16] present a taxonomy of different kinds of usage of inheritance, identifying specific features like subtyping, late binding and code reuse. However, empirical work demonstrating the amount of usage across each category is scarce. An empirical study by Lämmel et al [16] investigated reuse characteristics of the .NET Framework related to inheritance and defined a model for analysing frameworks. Their static and dynamic program analysis found significant use of inheritance for the purposes of code reuse and customization through late binding.

Tempero, Yang and Noble [1] investigated different types of usage of inheritance by defining a conceptual model for measuring inheritance use, based on the taxonomies provided by Meyer and Taivalsaari. They apply this model on an empirical investigation of 93 open-source systems. Tempero et al found significant usage of inheritance for the purposes of late binding to customize the behaviour of superclasses. Additionally, they found that Java developers use inheritance mostly for subtyping, and that around a quarter of inheritance usage could be replaced by composition. There are other uses of inheritance, but they are generally insignificant.

This study aims to corroborate the results by Tempero et al. for a different set of Java systems and through a different technical approach. In addition, it broadens the applicable kinds of systems by analysing a comparable set of C# systems.

3 Original Study

Tempero, Yang and Noble empirically investigated the use of Java inheritance in practice in their study *What Programmers do with Inheritance in Java* [1]. They looked at purposes of use of inheritance; to provide subtyping, reuse of code, allow subclasses to customize superclasses' behaviour, or categorizing objects. They created a model for different categories of usage of inheritance by defining attributes on relationships between types. Their model is also used in this replication study, therefore a detailed description is provided in section 6.1.

3.1 Research Questions

This section describes the four research questions defined in the original study and their motivation. The authors mainly base their research questions on two reports of how inheritance could be used in practice. A study done by Meyer [16] provides a taxonomy of inheritance, defining 12 possible types of inheritance use. A similar study by Taivalsaari [15] concludes that inheritance in general can be defined as *an incremental modification mechanism in the presence of late-bound self-reference*. *Late-bound self-reference* is defined as an object calling a method on itself (in Java and C# a call to *this*), where that method will be bound to a different method at runtime. In Java and C# this would mean the called method has been *overridden*. This definition has not been backed by empirical evidence and was authored in 1996. Taivalsaari defining late-bound self-reference as the most profound benefit of inheritance leads Tempero et al. to further investigate the actual use of late-bound self-reference.

RQ1: To what extent is late-bound self-reference relied on in the designs of Java systems?

A second form of inheritance use is subtyping, being able to replace one type with another when an inheritance relation exists between those types. For example, in Java and C#, a method accepting a *Mammal* as a parameter gladly accepts a *Giraffe* given an inheritance relation between *Giraffe* and *Mammal*. Taivalsaari indirectly implies that the subtype relationship is "rarely" used. Other work does not seem to agree; in his book *Effective Java* [17]*(p85)*, Bloch claims the only appropriate use of inheritance is where the subclass is a subtype of the superclass. Empirically investigating actual use of subtyping would therefore be a valuable contribution in validating this.

RQ2: To what extent is inheritance used in Java in order to express a subtype relationship that is necessary to the design?
They define 'necessary' as the requirement for an inheritance relationship to exist for the code to compile. Considering the previous Mammal and Giraffe example, the code

would not compile correctly if an inheritance relationship between Mammal and Giraffe did not exist.

Gamma et al instruct readers to “Favor composition over inheritance” [19] as later supported by Bloch [17], suggesting that some forms of inheritance can and should be replaced by composition. Given that prominent authors have strong opinions against unnecessary uses of inheritance, Tempero et al hypothesise that little room for replacing inheritance with composition exists. This motivates the third research question:

RQ3: To what extent can inheritance be replaced by composition?

While late-bound self-reference, subtyping and replacement of inheritance by composition are investigated, other inheritance uses remain open. To look at other significant uses of inheritance, they add a final open-ended research question:

RQ4: What other inheritance idioms are in common use in Java systems?

3.2 Definitions

While this section presents only a summary, some terms need to be defined for the purpose of brevity. The authors view a software system as a directed graph in their study results. The *nodes* in this graph represent types (classes or interfaces). The *edges* represent *inheritance relationships* between types For example, when a *class-class relationship* is mentioned, this is defined as a class *extending* another class. When a *class-interface relationship making use of subtyping* is mentioned, this is defined as a class (child) implementing an interface (parent), for which some occurrence of code has been seen where the child class was provided, but the parent class was expected. This is an indication of *substitution*.

3.3 Study details

The original study covered 93 open-source Java systems from the Qualitas Corpus [20]. The corpus provides a diverse set of systems for the purpose of analysis, varying greatly in size and application. In addition, they included a longitudinal analysis of the version history of two systems, *freecol* and *ant*.

Their tools statically analyses systems’ bytecode to find results. They describe some minor limitations caused by using bytecode instead of source code for analysis, these and other considerations are discussed in section 5.2.2.

In order to answer the first research question, late-bound self-reference must be investigated. To quantify the use of late-bound self-reference, all invocations are investigated. Given the definition of *call site* as the place where the invocation takes place and *invocation target* as the type on which the invocation is done, if the call site is the same type as the invocation target type, a *downcall* attribute is assigned to all types overriding the method being called. This assumes that the downcall can actually take place, which may not be true for all cases as explained further in section 9.5.

```
class T
class S extends T
//example of substitution:
T t = new S();
```

Code Sample 1: Example of substitution

```
class P {
  void M() { D(); }
  void D() { };
}
class C extends P {
  void D() { }
}
```

Code Sample 2: Example of a late-bound self-call. When method P.M calls P.D, the method C.D will be invoked instead.

For the second research question, subtype usage has to be found. To determine subtyping, they look at specific places where substitution can occur. They name passing a parameter, returning a value, assignment and cast. An example of the assignment case is shown in Code Sample 1. Each time one of these expressions or statements occur, the static type of the target is compared to the static type of the provided argument. When these types are different, some form of subtyping must be present. Specific details of these cases are listed in section 6.1.

To determine the amount of code reuse, they define two metrics: *internal reuse* and *external reuse*. Internal reuse occurs when a method in a child class makes use of code defined in a parent type. Similarly, external reuse occurs when code outside of the inheritance hierarchy makes use of code defined in a parent type, through a reference to a child type. To measure internal and external reuse, all occurrences of member access are analysed. Member access consists of accessing/assigning a field or invoking a method. If the type that declares the member is in is an ancestor of the type where the access takes place, internal reuse is counted. Otherwise external reuse is counted.

The study ignores exception and annotation types, this decision is detailed in section 5.2.4.

3.4 Results and conclusions

This section briefly covers the results reported by Tempero et al. Section 7 will cover these in more detail through a comparative presentation of results.

For their first research question related to late-bound self-reference, they measured downcall potential among class-class (CC) relationships. This indicates an inheritance relationship between two classes within the system under investigation exists, where the parent class calls a method on itself, and another class overrides that method. They found significant use, around a third of CC relationships make use of downcall. They report high variance among systems with no apparent relation to system size. Two systems did not have any downcall relationships while the maximum was 86%. The median value is 34%.

Their second research question relates to *necessary* inheritance. This is defined as edges that rely on subtype use in order for the code to compile: the proportion of inheritance relationships making use of subtyping. They report highly common use of subtyping – it seems to dominate the overall use of inheritance. For relationships between classes (class-class edges) there is high variation, comparable to downcall edges, but a significantly higher median of proportion 76%. The lowest proportion of subtype edges reported was 11%. They reported two systems with 100% subtype use. For class-interface relationships they report a median of 69% with one system having zero subtype use and four systems at 100%. Interface-interface edges are less common; 23 out of 93 systems do not have any and a further 51 systems have less than 10 interface-interface pairs. A median use of 63% is reported. They summarize that at least two thirds of relationships are used as subtypes in the systems they investigated, conflicting with Talvasaari's implication that using inheritance for subtyping is a rare occurrence [15].

Their third research question relates to the possibility of replacing inheritance with composition. A mechanical procedure of doing this was introduced by Bloch in his book *Effective Java* [17]. They report that around 22% of class-class relationships are potential candidates for refactoring inheritance into a compositional design.

For the fourth and last research question Tempero et al investigated other uses of inheritance. While around 87% of relationships between types have already been explained by previously discussed matter, there is still some other use of inheritance visible. These will be further detailed in section 7.4.

4 Language differences

This study investigates both Java and C# systems. In order to extend the research with the C# language, the differences between Java and C# need to be discussed. This section discusses differences in syntax and behaviour of language features that may influence the practical use of inheritance when compared to Java. It forms the source of hypotheses made for the usage of inheritance in C#, which are discussed in section 5.1. Börger and Stärk [21] provide a formal approach to comparing Java and C#, aiding in the completeness of this section, however their research originates from 2004 and much of both Java and C# has changed since then.

The list of differences is assumed to be exhaustive within the scope of this research, language differences not mentioned should not have impact on the metrics used. This section considers Java 7 and C# 5.0.

4.1.1 Overriding methods

The first research question of the original research investigates the use of late-bound self-reference. An important difference between Java and C# exists in the behaviour of method overriding. Java implements all methods as overridable by default. A programmer in C# has to specify the `virtual` keyword to make a method overridable. Both Java and C# make it possible to prevent overriding a method explicitly by using the `final` and `sealed` keywords respectively. Section 5.1 discusses how this key difference in explicitness is expected to influence the results for late-bound self-reference.

4.1.2 Implicitly typed local variables

C# supports declaring local variables that have the implicit type `var` [22]. The compiler then infers the type based on the expression that initializes the variable. Based on results, it appears there is a significant impact on subtyping and potentially on external reuse. By inferring the type of a variable, substitution cannot occur from the initialization of a variable, while this form of substitution is quite common among both Java and C# systems investigated in this study, as shown in section 8.3. This was not originally hypothesized and the impact of usage of `var` is further detailed in section 9.6.

4.1.3 'as' operator

C# introduces a second type of cast expression: the `as` operator. It evaluates to `null` when a cast fails. The `as` operator is treated in the same manner as a normal (direct) cast when considering subtyping relationships.

4.1.4 Value types

C# and the Common Language Runtime (CLR) support value types through the `struct` keyword. They are not allocated on the heap unless wrapped by its corresponding *object type* (boxed [23]) and provide bitwise `HashCode` and `Equals` implementations. Value types can only implement interfaces as far as inheritance goes. A value type is considered to be a class for the purposes of analysing inheritance patterns.

4.1.5 Properties

C# has a syntax for the commonly occurring pattern of getters and setters called *Properties*. These properties contain a getter and/or a setter method called *accessors*. Accessors can be overridden like normal methods. A special form of property called an *indexer* is also present, containing an arbitrary number of parameters accessed through a square bracket syntax, as shown in Code Sample 3. Given their method-like nature, property accessors are treated as methods for the purposes of determining facts related to inheritance usage, in this case subtyping, code reuse and late-bound self-reference.

```
class MyList {
    public int Length {
        get { ... }
        set { ... }
    }

    object this[int i] {
        get { ... }
        set { ... }
    }
}
MyList a;
int i = a.Length;
object item = a[3];
```

Code Sample 3: Example of property and indexer declaration and usage in C#

4.1.6 Constants interface

A Java interface can contain fields with a constant value that is implicitly `static final` [24]. One of the patterns investigated is an interface and its parents containing solely constants, with no methods declared. An example of usage for this is a *tokens* interface used by parsers and tokenizers.

In C#, declaring fields within an interface is not possible, although the Common Language Infrastructure (CLI) and the Visual C++ language support it through marking it as `literal` [25], exposed as `static` read-only properties in C#. Since relationships analysed in this study are only between types defined in the system under investigation, reuse of constants cannot occur for a relationship defined in C#, unless the parent of that relationship is a class.

4.1.7 Dynamically typed variables

C# supports the `dynamic` keyword and the Dynamic Language Runtime (DLR) since version 4. Any method calls or field access is done using dynamic binding, the appropriate method/overload is resolved at runtime. This means the type of a variable of type `dynamic` should be considered as the type of object it currently holds as far as polymorphism and subtype relationships go. This requires looking at the latest assignment. No runtime analysis is done in this research so this cannot be determined, however the impact of not including dynamically typed variables is estimated in section 9.6.

4.1.8 Foreach

In Java, the *foreach* statement allows iteration over any collection through the syntax `T item : collection`. The type of elements in the `collection` must be `T` or a type less specific than `T`. A compiler error is generated when this is not the case. C# has a similar syntax of `S item in collection`, but does not have the constraint that type S must be the same type as the elements in the collection. It instead inserts a cast from the element type to `S` [26]*(p264)*. This means that both a *downcast* and *upcast* may occur when using the foreach statement in C#, while its Java counterpart only allows for *upcasts*. This possibly indicates a higher number of subtyping occurring from *foreach* statements in C#.

4.1.9 Extension methods

The notion of extension methods allows a static method to be called as if it were a member of a type directly if the first parameter of the static method matches is assignable to its type and the parameter uses the `this` keyword. This is illustrated in Code Sample 4. This feature is solely syntactic sugar for static methods, extension methods are considered to be conventional static methods.

```
class A { }

static class Extension {
    public static void M(this A a) { }
}

A a = new A();
a.M(); //is the equivalent of:
Extension.M(a);
```

Code Sample 4: Extension method usage in C#

4.1.10 Enumerations

In Java, an enumeration is a class that can implement an interface, override or declare methods and declare fields. Each enumeration value is an instance of the class. In contrast, a C# `enum` is a wrapper around one of the primitive integer types (8-64 bit signed or unsigned integers). It assigns names to one or more of the values that can be represented by the primitive type. Methods cannot be added to enumerations unless extension methods are used. Enumerations cannot be extended in Java or C#, and do not participate in any inheritance relation covered in this research. They are excluded completely.

4.1.11 Events

C# allows for so called *multicast delegates.* These are comparable to a normal reference to a method/function, but allow for multiple functions be registered in its *invocation*

```
//define a method signature for the event
//handler using a delegate type
delegate void ButtonClick
           (Button clickedButton);
class Button {
    //button defines the 'click event'
    public event ButtonClick Click;
    void SomeInternalLogic() {
        //trigger the event
        Click(this);
    }
}
class Other {
    void AddClickHandler(Button b) {
        //add a method to the invocation
        //list, subscribing to the event
        b.Click += OnClick;
    }
    void OnClick(Button clickedbutton) {
        //...
    }
}
```

Code Sample 5: Example of basic *event* usage in C#.

list. When invoking a multicast delegate, all items in the invocation list are called. *Events* are a special kind of multicast delegates. Events only publicly expose the `add` and `remove` operations (called with the `+=` and `-=` operators). These operations can be overridden in derived classes but rarely are, it would be a surprise to encounter such a pattern. Invoking the delegate (raising the event) is only possible in the declaring class, often a method is exposed to invoke the delegate. For the purpose of this study, the `add` and `remove` operations are considered to be methods. Code Sample 5 shows a basic example of *events* and how they are used in C#. Note that the *delegate* type defines a method signature, used by the event invocation and event handler code. These methods usually return `void`, when a return value is specified, the return value of the *last* handler is used.

4.1.12 Anonymous methods, classes, closures

C# allows defining anonymous methods. Their type can be determined at compile time and they should be considered as any other type. Function types are excluded from this study. Anonymous methods may capture local variables from the outer scope using *closure containers*. These are implemented using anonymous classes in C#. Anonymous classes in C# cannot implement interfaces or inherit from other classes. Anonymous classes in Java implement an interface or extend an abstract class. These classes can participate in a class-interface or class-class relation in the context of this research.

4.1.13 Explicit interface implementations

C# allows declaring methods as being specific implementations of interfaces. This adds complexity to the method binding used by the CLR as illustrated in Code Sample 6. When invoking a method from an interface on an object, the method binding rules are as follows:

1. Call the first explicit interface implementation matching the signature searching the inheritance graph upwards starting from the called object's type.
2. If no explicit implementation was found, call the first method matching the signature searching the inheritance graph upwards starting from the called object's type.

```
interface I { void O(); }

class B : I {
    public virtual void O() { Console.Write("B.O"); }
    void I.O()              { Console.Write("(I)B.O"); }
}

class D : B, I {
    public override void O() { Console.Write("D.O"); }
    void I.O()               { Console.Write("(I)D.O"); }
}

B ctest = new B();
I itest = ctest;
ctest.O();        //B.O
itest.O();        //(I)B.O
ctest = new D();
ctest.O();        //D.O;
itest = ctest;
itest.O();        //(I)D.O;
```

Code Sample 6: explicit interface implementations in C#

Explicit interface implementations affect the way code reuse is measured. When a call to an explicitly implemented interface method implementation is encountered, external reuse will occur between the type declaring that method and the implemented interface.

4.1.14 Operator overloading and sideways type conversions

C# supports overloading some operators and implicit type conversions, these are static and therefore cannot be overridden. These conversions might expose usage of subtyping as seen in Code Sample 7. In the context of this study, overloaded operators are viewed as static methods. Implicit and explicit conversions are also viewed as static methods.

```
class A {
    public static A operator +(A left, A right) { return new A(); }
    public static implicit operator B(A item)   { return new B(); }
    public static explicit operator C(A item)   { return new C(); }
}
class D : A { }
class B { }
class C { }

A a = new D() + new A(); //+ operator, subtype between D and A
B b1 = a;                //ok
B b2 = (B) a;            //ok
B b3 = new D();          //ok
C c1 = a;                //invalid: cannot implicitly convert
C c2 = (C)a;             //ok
```

Code Sample 7: Sample of operator overloading and type conversions in C#

4.1.15 Generics

The way generic types are implemented is profoundly different when comparing Java and C#. Java implements generics using Type Erasure [27]. C# and the Common Language Specification implements generics in the MSIL bytecode [25] (p. 128). Identifying a type in C# means using its fully qualified name in conjunction with the number of type parameters, since inheritance relations can and do exist between types with the same name but with a varying number of type parameters. Using the number of type parameters identifies types as defined by the programmer; a programmer may use different closed generic types e.g. `List<string>` and `List<int>` but only writes a single `List<T>`.

4.1.16 Covariance and contravariance

Analysing C# means introducing the complication of generic covariance and contravariance. This feature extends polymorphism, allowing type arguments to participate as well. Using the `out` and `in` specification on type parameters declares them to be covariant and contravariant respectively. Code Sample 8 illustrates this; if a value of type parameter `T` is only used as output (through return values) the value may be replaced by a type *less specific* than `T` without breaking type safety. Conversely, if a value of type parameter `T` is only used as input, through parameter values, the value may be replaced by a type *more specific* than `T`.

For example, the `IEnumerable<T>` interface (the equivalent of `Iterable<T>` in Java) is declared covariantly: an `IEnumerable<Giraffe>` may be implicitly cast to an `IEnumerable<Mammal>` without breaking type safety given an inheritance relation between `Giraffe` and `Mammal`. Implicitly or explicitly casting a covariant or contravariant type along an inheritance relation indicates a subtype relationship between those types; the relation between `Giraffe` and `Mammal` is required for the code to compile.

```
interface ICovariant<out T> {
    T GetT();
}
void Covariance() {
    ICovariant<Giraffe> giraffes;
    ICovariant<Mammal> mammals;
    mammals = giraffes; //ok
    giraffes = mammals; //error
}
interface IContravariant<in T> {
    void AcceptT(T value);
}
void Contravariance() {
    IContravariant<Giraffe> giraffes;
    IContravariant<Mammal> mammals;
    mammals = giraffes; //error
    giraffes = mammals; //ok
}
```

Code Sample 8: Example of covariant and contravariant interface declarations

4.1.17 Bounded quantification

The example below illustrates a subtype relationship occurring from usage of generic type constraints in C#: a subtype relationship exists because any implementation of `IH<T>` means that in `IG<T>` an instance of type `A` or derived is expected but an instance of `B` or a derivative thereof is supplied. If code exists that does not close type parameters in covariant or contravariant definitions, subtype relationships might be missed that could be inferred from type parameters. The original study makes no reference to this pattern. To maintain consistency with the original research, subtype relations inferred from these constructs are not considered. However, generic variance discussed in the previous section is included.

```
interface IG<in T> where T : A {
    void DoSomethingWithT(T obj);
}

interface IH<in T> : IG<T> where T : B {
    // Calling DoSomething from a reference of this type automatically
    // constitutes a subtype relationship.
}
```

Code Sample 9: Contravariant type parameter indicating a subtype relationship without closing an open generic type

4.1.18 Null coalescing operator

In C#, the expression `A ?? B` is the equivalent of writing the ternary expression syntax `A == null ? B : A`. This potentially leads to an occurrence of subtype usage, as the types of A and B may not match.

4.1.19 Asynchronous methods

C# supports language integrated continuations through the `async` and `await` keywords. This introduces a form of asynchronous programming that appears to be a synchronous invocation as seen in Code Sample 10. For the purposes of determining subtype relations, any occurrences of the structure `x = await t` where `t` is of type `Task<U>` is substituted by `x = s` where `s` is of type `U`. This effectively erases the `Task`, exposing the actual parameter type for the asynchronous method's continuation callback.

```
class P { }
class C : P {  }
class Other {
    public Task<C> GetChildAsync() { ... }
    public async void DoSomethingAsync()
    {
        P p = await GetChildAsync();
        //subtype between C and P
    }
}
```

Code Sample 10: asynchronous method invocation in C# 5.0

5 Replication Study

This section describes the research questions for the replication study, the rationale and hypotheses. The specific changes made to the original study are listed in section 5.2.

5.1 Research questions

The main purpose of this replication study is the validation of the results presented by Tempero et al. It verifies the original research by repeating it using a different set of tools and systems. Additionally, this study broadens the scope of the original study by introducing C# as a second programming language.

The original research uses static bytecode analysis on 93 open source systems from the Qualitas Corpus [20]. The replication study analyses 86 open source systems from the Qualitas.class corpus [28] through source code analysis. Section 5.2 discusses these differences in more details. This study hypothesizes these differences in technical research method and systems will not cause different results when compared to the original study. This motivates the first research question.

RQ1. *Are the conclusions from the study 'What programmers do with Inheritance in Java' by Tempero et. al. [1] valid when source code analysis is used for a similar but different set of systems?*

As discussed in section 4.1.1, C# methods must be made overridable explicitly through usage of the *virtual* keyword. This invites one to think that late-bound self-referencing in C# occurs less frequenly than in Java systems, because the programmer has to be explicit about making a method polymorphic. While this study does not qualitatively investigate the programmers' decision making in this regard, the expectation exists that implicitly making a method polymorphic could cause some calls be made unintentionally by the programmer creating the class in which the calls occur (the superclass). No empirical investigation has been done to determine unintended overriding, but there must be cases where this happens. Searching the issues database in GitHub [29] for 'unintentional override' yields many relevant results, educational material such as the books by Deitel [30] [p386], Bloch et al [31][Puzzle 58] and the language specification [32][section 13.5.6] mention unintentional overriding as a potential pitfall.

If there is no difference, we may consider it plausible that when a method is overridden, the author of the superclass intended for the possibility of overriding that method. This motivates the second research question:

***RQ2**. Does late-bound self-reference occur less often in C# systems when compared to Java systems?*

Considering the differences explained in section 4, for the remaining aspects of the original study: subtyping, reuse and other uses of inheritance this study expects similar results for C# and Java. There are some minor considerations such as implicit casts in foreach statements, generic covariance and contravariance and other types of accessors such as events and properties. No empirical evidence is known of how these features relate to the inheritance usage of C# systems; the impact is unknown. The hypothesis is that these language features do not impact actual inheritance use for the important metrics this study uses to measures it: subtyping and reuse between classes. This motivates the third research question.

***RQ3**. Are the conclusions from the study 'What programmers do with Inheritance in Java' by Tempero et. al. [1] related to code reuse, subtyping and other common idioms valid for open source C# systems?*
Note that 'code reuse, subtyping and other common idioms' refers to the second, third and fourth research question of the original study, as described in section 3.1.

5.2 Changes to the original study

This section details the changes made to the original study. This study adds the C# language as a source of information, section 5.2.1 describes how this requires some adaptation to the model and a comparable set of systems. The replication study employs static source code analysis instead of bytecode analysis. The motivation behind this and the potential implications are described in section 5.2.2. For the Java analysis, a different set of systems, although with large overlap, has been chosen. This is described in section 5.2.3. A final and minor change to the original study was done, including annotation and exception types for analysis, detailed in section 5.2.4.

5.2.1 Addition of the C# language

For the purpose of broadening the result set a secondary equivalent analysis on systems developed in the C# language was done. The model of inheritance used in the original research as explained in section is also applicable to the C# language.
A set of 83 open-source systems containing around 11,5 million code lines was compiled with the aid of Ohloh [33], a database of open source projects. This set contains diverse projects, including but not limited to the 'Roslyn' C# compiler, content-management systems, object-relational mapping frameworks, dependency injection frameworks and build tools. The systems used in the original study and the Java and C# replication are compared with respect to size, domain and number of inheritance relationships in section 6.2. The specific set of analysed C# systems are listed in Appendix B.

5.2.2 Source code instead of bytecode

A study by Logozzo et al [34] discusses the challenges faced by bytecode analysers for the purposes of program verification, when compared to source code analysis. They show through a formalized approach that bytecode analysis tools can only obtain completeness for trivial cases such as the *nop* operation. This illustrates problems related to bytecode analysis, however the question remains how much this affects the study of inheritance use. This section discusses the advantages and pitfalls of using bytecode analysis versus source code analysis. Specific details of bytecode implementations are discussed where relevant, but this section focuses on the general notion of analysing bytecode versus source code in the context of this study.

One advantage of using bytecode is the possibility of analysing closed source systems. Java and C# both use a JIT compiler in most cases (tools such as NGEN [35] and Excelsior JET [36] allow for native compilation), indicating the binary format for systems written in these languages are generally available for analysis. However legal constraints will often prevent analysis of closed-source systems.

Another advantage of using bytecode is that any system written in a language compiling to JVM or MSIL bytecode could be analysed, allowing for example VB.NET, F#, Scala and Clojure to be analysed as well. However, this study only focuses on Java and C#.

A disadvantage of using bytecode is that some compilers do small optimizations when compiling from source code to bytecode. This can include and might not be limited to replacing virtual dispatch with instance dispatch and not emitting code for unreachable paths [37] [38]. In addition to being optimized, bytecode might be obfuscated, adding bogus methods and classes possibly interfering with results. At least one system in C# (OrmBattle.NET) uses a post-build bytecode injector (PostSharp [39]) that could severely change emitted code. Additionally, at least 10 C# open-source systems use ILMerge [40], a tool that merges output of different binaries into a single binary, possibly removing the ability to make a distinction between system code and external code when dependencies are merged into the system binaries.

Arguments for using original source code is maintaining full integrity of semantics and intent, for example an explicit call to the default constructor of a parent class can be distinguished from a compiler-injected call. Code that is not deployed to the resulting application, like unit test code, is maintained. This may yield a better picture of the programmers' way of working. Appropriate tools are available (Rascal MPL language and NRefactory), which support extraction of all information required for the data in this research through source-code based analysis using abstract syntax trees (ASTs). Because of the availability of tools that support the analysis of source code directly and the possible loss of information when investigating systems using bytecode, this study uses source code for fact extraction.

5.2.3 Qualitas.class corpus

The original research analysed systems in the Qualitas Corpus [20], a collection of software systems selected for the purpose of empirical research. It aids in the reproducibility of studies by providing a consistent and diverse set of Java systems for investigation. While this dataset is certainly valuable, analyses such as this one require resolution of external dependencies. Large systems may have numerous external dependencies that can be tedious to resolve. The Qualitas.class [28] corpus addresses this problem by providing compiled Eclipse projects for the systems in the Qualitas Corpus. This results in a large overlap between systems analysed in this study and the original study, but also introduces other versions of systems and different systems. Section 6.2 shows how the set of systems is comparable in size, distribution and architecture to the set of C# systems and the set used in the original study. The specific set of systems analysed is listed in Appendix C.

5.2.4 Inclusion of annotation and exception types

The original study excludes annotation and exception types. The authors motivate this decision by reasoning that exception types are always defined through use of inheritance, and that this use is mandatory. Hence, the programmer cannot decide not to use inheritance for exception types. Their reasoning with respects to excluding annotation and exception types is valid, using inheritance for these types is certainly not a decision that can be made by the developer. However, the results and conclusions are based solely on relations between types *inside* the system of investigation. This means that any edge between two types that ultimately derive from (for example) *java.lang.Throwable* is an explicit decision by the programmer to use inheritance, because the edge between the user-defined exception or annotation type and the external type is not included in any measurement. This study assumes the notion that if the developer does not use inheritance for exceptions types, all exception types would derive only from external types, and no relationships would be visible in the results of this study.

6 Research method

This section discusses the method of quantitative analysis employed by this study. Since this is a replication study, much has been borrowed from the original study. Section 6.1 describes in detail the method used by the original study to model the inheritance usage characteristics. It mentions variations and additional patterns that appear through the addition of C#. Section 6.2 compares the systems investigated for the original study, the Java replication and the C# replication. The specific tools used to analyse source code (Rascal MPL and NRefactory) are described in section 6.3, followed by a brief overview of the technical implementation of the analysis in section 6.4.

6.1 Modeling inheritance

Tempero et al define a conceptual model used to analyse the inheritance usage patterns of Java systems. This section describes their model in detail, complemented by code examples explaining the specific patterns in source code that are measured in order to quantify the usage of inheritance. Their model consists of a directed graph where vertices portray the classes and interfaces within a Java system and the edges represent inheritance relations between these types. This section uses specific terminology for brevity; 'an edge between type A and B' means there is a class or interface A that directly or indirectly inherits from type B in some form, 'edge A->B has the downcall attribute' means that type A inherits from type B, and some code pattern was found that constitutes a downcall relationship between type A and B. This section conceptually describes attributes on these edges supplemented with source code patterns that constitute assignment of a specific attribute to an edge. These attributes are the source of metrics used in both the original and the replication study.

CC, CI, II: An edge will have one of these attributes if it represents an edge between a Class-Class, Class-Interface or Interface-Interface respectively.

External Reuse: An edge from type S (child) to T (parent) has the external reuse attribute if another external class accesses a field or invokes a method using a reference of type S when the field or method is declared by type T. The definition does not assume a class-class relation, however mainly class-class relations are discussed with respect to external reuse. Code Sample 11 illustrates the

```
class T {
    void m() { }
    int f;
}
class S extends T { }
class Other {
    void method() {
        S s = new S();
        //external reuse S->T x3:
        s.m();
        s.f = 3;
        int a = s.f;
    }
```

Code Sample 11: External reuse between two classes.

patterns of code leading to an edge receiving this attribute. Note that accessing a property or event in C# also counts towards external reuse.

Internal Reuse: An edge from class S (child) to T (parent) has the internal reuse attribute if a method declared in S invokes a method or accesses a field declared in T. Note that usage of `this` or `super` as a qualifier is not distinguished from other qualifiers as illustrated in Code Sample 13.

```
class T {
    void m() { }
    int f;
}

class S extends T {
    void method()
    {
        this.m();      //internal reuse through this
        super.m();     //or super (base in C#)
        S anotherS = new S();
        anotherS.m(); //internal reuse through
                      //another instance
    }
}
```

Code Sample 13: Different forms of internal reuse between two classes.

Subtype: An edge from type S (child) to T (parent) has the subtype attribute when some occurrence of an expression exists where T is expected and S is provided. This includes assigning a value, passing a parameter, upcasting or downcasting, using the ternary

```
class T
class S extends T
class E {
    void m(T t);
    T subtypes() {
        T t = new S(); //assignment
        m(new S()); //passing a parameter
        t = (T) new S(); //casting
        t = 3 > 4 ? new S() : new T(); //ternary operator
        List<S> listOfS;
        for (T item : listOfS) { } //foreach statement
        return new S(); //return value
    }
}
// in class T
void subtype()
{
    new E().m(this); //subtype through 'this changing type'
}
```

Code Sample 12: Examples of expressions resulting in a subtype attribute assigned to an edge.

operator or declaring a different variable type in a `for` statement. Examples of the types of expressions resulting in a subtype attribute are shown in Code Sample 12. Note the occurrence of *this changing type*. When the pseudo-variable `this` is used and an edge to a child type exists, it is possible that `this` changes type when it is used, implying a subtype relation between that child type and the parent.

Another case resulting in the assignment of the subtype attribute is a *sideways cast* as illustrated in Code Sample 14 . For this cast to succeed, the two interfaces must share a common child type. Note that this is not limited to class-interface relationships, either `I1` or `I2` could be a class, but not both.

```
interface I1
interface I2
class Child implements I1, I2
void M(I1 item) {
    I2 i2 = (I2)item;
}
```

Code Sample 14: Example of a sideways cast

Downcall: An edge from class C (child) to class P (parent) is assigned the downcall attribute when a method defined in P calls a method m() defined in P and m() is overridden in C. The object on which this invocation takes place must be constructed from the child type or one of its descendants. Code Sample 15 illustrates the occurrence of a downcall through a method call. The downcall attribute represents late-bound self-reference.

```
class P {
    void q() {
        m(); //downcall
    }
    void m();
}
class C extends P {
    void m();
}
```

Code Sample 15: Occurrence of a downcall edge between C and P.

The definitions that follow occur less frequently, and will be reported under 'other common idioms of inheritance'.

Framework: An edge from types P to Q that does not have external or internal reuse, subtype or downcall receives the framework attribute if Q descends from a third party type.

Constants: An edge from types P to Q receives the constants attribute if type Q and all of its parents do not define any members with the exception of constant fields (`static final` in Java, `const` or `static readonly` in C#). Code Sample 16 illustrates an occurrence of an edge with the constants attribute.

Marker: An edge from type G to interface H has the marker attribute if H does not declare any members and all of its parents also have the marker attribute.

```
interface Tokens {
    int EOF = 0;
    int BOOL = 1;
    ...
}
```

Code Sample 16: The *tokens interface* is a common pattern used in parsers and tokenizers.

Super: If a constructor in class C (child) invokes a constructor defined in class P (parent) explicitly, the edge from C to P receives the super attribute.

Category: An edge from type C (child) to type P (parent) will get the *category* attribute if there has been no subtype use seen for it, but a sibling type with respect to P has shown subtype usage.

Generic: An edge from type R to type S has the generic attribute if there has been a cast from Object to S and there is an edge from R to some (non-Object) type T. In practice, this usually indicates that some object has been put into a non-generic container and has been cast to a different type upon its removal. This indicates some relation exists between those two types.

6.2 Systems investigated

This study investigates both Java and C# code and replicates a previous study. To be able to compare results among data sets, an indication with respect to the investigated systems' size should be presented. Figure 2 lists a few high-level metrics for the two data sets studied. For the metrics related to inheritance relationships between types, only those between system types are counted. As can be seen, the two data sets for the replication study are comparable in size, with the Java systems making slightly more use of inheritance per line of code on average.

The variance between systems for all metrics is higher among the Java systems used in the replication study, indicating that the set is more diverse in terms of system size. The original study reported no relation to system size for any metric used. The same results are found in this study, both the C# and Java results indicate no apparent relation to system size. This study therefore assumes that the reduced diversity in system size for C# systems does not have a meaningful impact on the results.

The specific set of systems used for C# and Java are listed in Appendix B and Appendix C respectively. A rough categorization of system domains is listed in Figure 1. Note that the similarity between the replication study for Java and the original study is caused by the large overlap of systems investigated. 52 systems from the original study were also used in the replication study, and a further 20 were included with a different version.

Metric		Replication C#	Replication Java	Original Java
#Systems		83	86	93
KLOC[1]	Sum	11.673	11.176	13.869
	Avg	141	128	149
	Std Dev	171	232	239
CC Edges	Sum	41.234	49.358	39.973
	Avg	496	573	429
	Std Dev	650	976	741
CI Edges	Sum	20.750	25.996	24.889
	Avg	250	302	267
	Std Dev	316	549	562
II Edges	Sum	2.731	3.707	2657
	Avg	32	43	28
	Std Dev	56	147	91

Figure 2: Comparison of system size for C# and Java systems used in this study and the original study.

System Domain	Replication C#	Replication Java	Original Java
middleware	15	14	13
testing	11	10	12
SDK	14	6	6
parsers/generators/make	4	9	9
diagram/data visualization	1	8	8
3D/graphics/media	5	5	6
database	3	6	6
IDE	3	3	3
games	1	3	3
persistence object mapper	4	1	1
programming language	3	1	2
tool/other	19	20	24

Figure 1: Rough categorization of system domains for the systems used in this study and the original study.

6.3 Tools used

For the analysis of Java code, the Rascal Metaprogramming Language (Rascal MPL) [41] was used. This language has first-class support for the representation of ASTs and its standard libraries implement AST structures for the Java language, creating them from Java code, and integration with the Eclipse IDE. Visiting tree structures is also a language feature, allowing a clear and concise representation of the analysis, as illustrated in Code Sample 17, where all local variables declared in an Eclipse project's Java code are printed. In addition to providing ASTs, the Rascal MPL libraries support the creation of an *M3* model. The M3 model contains information about inheritance relationships, method calls, types, etc. When the ASTs and M3 model are used in conjunction, a powerful method of Java code analysis is available. The Rascal MPL has some limitations as described in section 9.3.

```
asts = createAstsFromEclipseProject(|project://fitjava-1.1/|, true);
for (ast <- asts) {
 visit (ast) {
  case Expression variable: \variable(str name, int extraDimensions): {
   println("Encountered variable <name>");
  }
 }
}
```

Code Sample 17: Example of printing all local variables declared in the code of an Eclipse project using the Rascal MPL language.

[1] This is the number of physical code lines that were actually analysed, in thousands. For the original study, lines of code were taken from the metadata on the Qualitas Corpus [8]. For more details about the systems used in the original study see http://qualitascorpus.com/docs/metadata/attributes.html

```
public class VariableNamePrinter : DepthFirstAstVisitor {
 public override int VisitVariableInitializer(
          VariableInitializer variableInitializer) {
  Console.WriteLine("Encountered Variable: {0}",
                           variableInitializer.Name);
 }
}
```

Code Sample 18: Example of printing local variables using an AST visitor in NRefactory.

For analysing C# code, the NRefactory [42] .NET library was used. This is a C# compiler front-end used by the SharpDevelop and MonoDevelop IDEs. It contains a type resolver, AST data structures and when used in conjunction with .NET build tools, makes it possible to generate ASTs for C# systems. Visiting ASTs is supported by abstract Visitor classes as illustrated in Code Sample 18. The type resolver uncovers relations between types outside of the system boundary, leading to a complete picture of types within the system under investigation and any dependencies it has. As described in section 9.2 however, relationships existing within external systems may still not be uncovered because ASTs cannot be generated from MSIL bytecode using NRefactory.

6.4 Overview of technical implementation

This brief overview explains the methods and tools used to investigate the source code in C# and Java for the purpose of extracting information related to inheritance use.

The Java and C# source code are analysed using different tools written in different programming languages (Rascal and C# respectively). Facts extracted from code are written to CSV files in a uniform format containing definitions of types and edges and their attributes. Each system investigated produces eight CSV files, listing types, edges, subtype relations, internal reuse, external reuse, downcalls, generic attributes and super constructor calls. For C#, two more CSV files are produced, one reporting the use of 'dynamic' and 'var' and the other measuring lines of code. The dynamic type and type inference do not occur in Java systems, and information relating to the lines of code is available through the Qualitas.class corpus

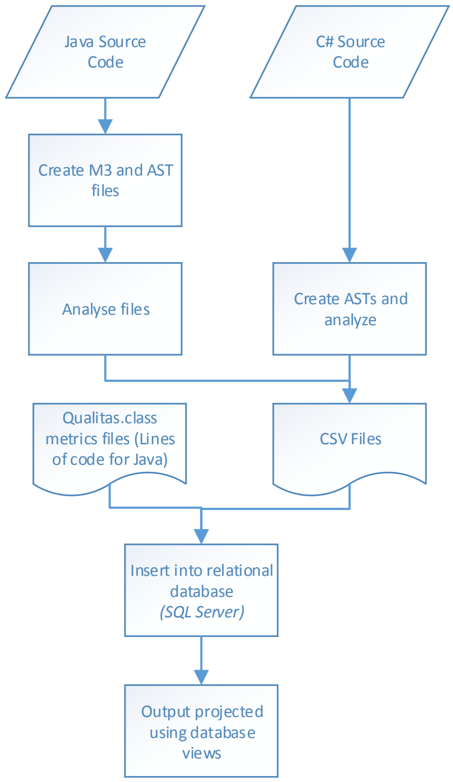


Figure 3: Visualisation of data flow through the various tools used in the analysis.

metrics data.

CSV files are loaded into a relational database, where data is summarized for the different measurements. The full integrity of details is maintained up to and including the relational database, enabling drilling down to specific pieces of source code that result in the assignment of one of the attributes. It also opens the possibility of excluding certain occurrences for the purpose of investigating the impact of decisions made in relation to the inheritance model. For example, the patterns resulting in a subtype assignment are categorized, allowing for the investigation of the effect of including *this changing type* for subtype relations as detailed in section 9.4.

7 Results

This section describes results found from the quantitative analysis of C# and Java open source systems. The original research has four research questions related to the investigation of late-bound self-reference, subtyping, code reuse and other cases respectively. This replication study defines three research questions, the comparison of the original study with the Java replication, the comparison of Java and C# related to downcalls (late-bound self-reference) and the comparison of Java and C# in general. Answering the research questions in this study requires a comparative report of the results done in the original research with results from this study, and requires a question-by-question analysis and interpretation. This leads to the structure of this section following the reporting model used in the original research, discussing each subject (downcall, subtyping, reuse and others) individually in a comparative report.

The analysis of results found in this section is presented separately, in section 8. That section contains a more in-depth investigation for interesting findings found in the results.

The original study reported results on a per-system basis using bar charts with system size on the x-axis. Due to the volume of data involved (comparing 262 systems in three categories: original study, Java replication, C# replication), the reporting visualizations used by the original study cannot be repeated, however the data for each metric is provided in the same level of detail in Appendix E. Note that no apparent relation was found between system size and any of the metrics reported, therefore it is considered appropriate to omit the information related to system size. This study instead opts to report using charts that show aggregated/averaged data per category. When the distribution among systems is shown, a boxplot is used. The boxplot utilizes the so called 'five number summary'. This method visualizes the distribution of a value set and makes no assumptions regarding the (normal) distribution of values. As illustrated by Figure 4, the raw values are summarized by retrieving the minimum, median, maximum and 25th and 75th percentile of values. When no exact value is available due to the number of values, the value is interpolated between the upper and lower bound. I.e. in Figure 4, the 75th percentile consists of the point between the values 8 and 9, this results in a value of 8.5. In the results, both values will be reported when applicable.

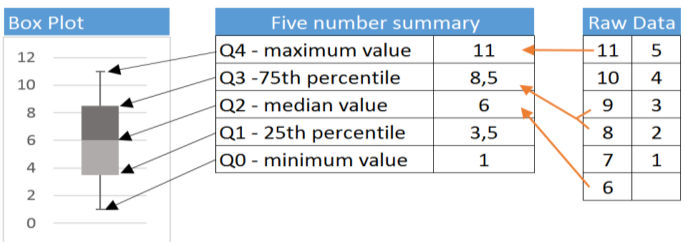


Figure 4: Illustration of how raw data is visualized in a box plot chart.

7.1 Downcalls (late-bound self-reference)

The original research reports on downcall edges by means of the proportion of system-defined class-class (CC) edges that have the potential for late-bound self-reference. This means a method in a parent class calls a method on itself, and that method is overridden in a child class. As summarized in section 3.2, Tempero et al report around a third of edges having the downcall attribute, with large variance among systems. A median of 34% of CC edges make use of downcalls. Appendix E contains more detailed data regarding downcalls, reporting on a system by system basis for the replication study and the original study.

7.1.1 Java replication

When comparing results of the replicated study on Java open source systems with the original study, less downcalls are found while the variance remains similar to the original study. As illustrated by Figure 5, this study reports a median proportion of 28% compared to the original 34%. All quartiles reported have lower proportions. Even for systems included in both studies with the same versions, consistently lower downcall proportions are found. Examples of such systems are *hsqldb* with 45% and 58% and *struts* with 26% and 37% for the replication study and original study respectively. The system for which the highest proportion of downcall CC edges is found is *displaytag*, having 85% out of its 178 CC edges making potential use of downcall. Both the original study and the replication study report three systems with zero potential for downcalls.

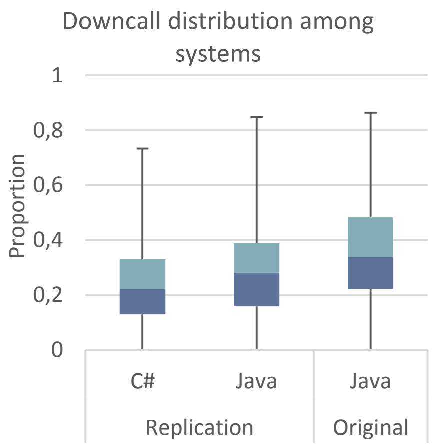


Figure 5: Box plot of downcall proportions among all systems, grouped by language and study.

7.1.2 C# systems

For the C# systems investigated, even lower downcall proportions are found when comparing to both the original study and the Java replication. A median proportion of 22% of CC edges are reported to have downcall occurrences, while all quartiles reported in Figure 5 have lower values than both the replication study for Java systems and the original study. The system with the highest proportion of downcalls is AForge.NET, having 73% of its 150 CC edges making potential use of downcall. Two systems were found having zero potential use of downcalls.

7.2 Subtyping

Class-Class (CC), Class-Interface (CI) and Interface-Interface (II) edges can all show usage of subtyping. Each type of edge is investigated separately and the results reported by Tempero et al are compared with results from this study, reporting data from C# and Java systems separately.

This study follows the subtype reporting model of the original study, CC subtype edges are shown as the proportion among edges that have occurrences of external reuse, internal reuse and/or subtype. This is related to and further described in section 7.3, where the potential for replacing inheritance by composition is investigated. In the results of the original study, as described in section 3.4, it seems that subtype use dominates the overall use of inheritance: at least two thirds of edges have some form of subtype usage reported. Appendix E contains more detailed information, on a system-by-system basis for subtype proportions among CC, CI and II edges.

7.2.1 Java replication

For CC edges in the Java systems, this study reports similar findings, as visualized in Figure 6 and Figure 8, with a median proportion of 75.5% compared to 75.8% for the original study. The variance however is slightly higher among Java systems in the replicated study. The original study reported two systems with 100% subtype use. The replication reports four systems with 100% subtype use, although three of those are small (61 or less CC edges). No systems were reported without subtype usage, the replicated study reports a minimum proportion of 7% compared to 11% for the original.

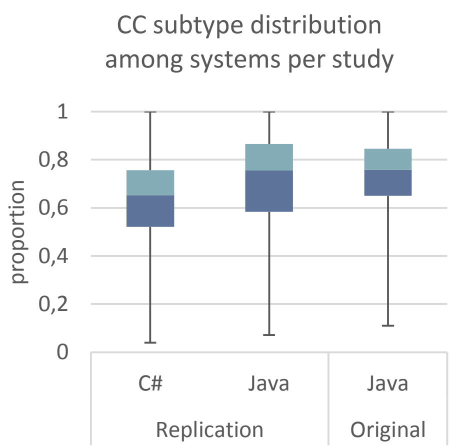


Figure 6: Box plot showing the proportion of CC subtype edges per study.

For the class-interface (CI) edges, results are relatively similar to CC with respect to the distribution among systems, illustrated in Figure 7. The original study and replicated Java study both contain a single system without CI edges. The Java systems investigated in the replication study contain two systems with no subtype occurrences, while the original study reports a single system. Three systems from the original study have 100% subtype use for CI edges, the replication study reports a single system. The median value is 69% in both studies.

II edges are less common, they make up around 4% of the 211.000 total edges, consistent among C#, Java and the original study. Out of the 86 investigated Java

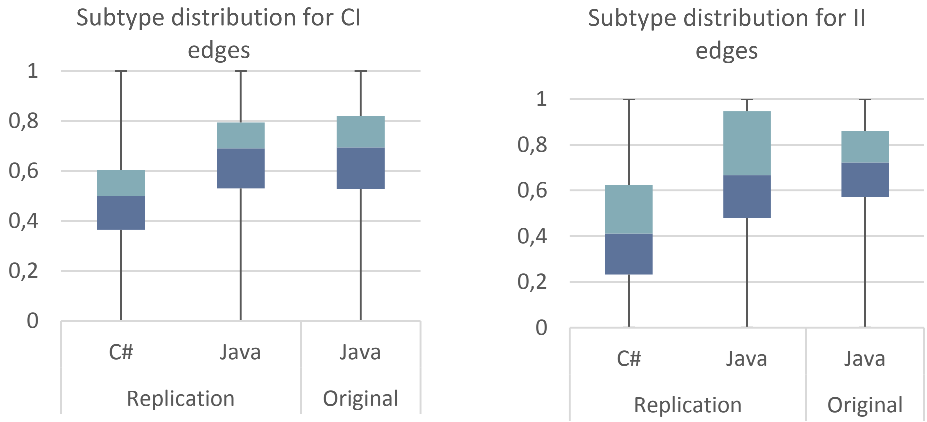


Figure 7: Subtype distribution among systems for CI and II edges. The y-axis represents the proportion of edges having the subtype attribute. Systems without the respective edges are omitted.

systems, 16 systems have no II edges. For the original study, 23 out of 93 systems have no II edges. The original study reports a median of 63%, however systems without II edges are counted as having a 0% subtype proportion. This study finds median values of 72% for the original study and 67% for the replication study. Two systems show zero subtype usage among II edges compared to four systems in the original study. 16 systems in the replication study have 100% subtype usage, compared to 13 systems in the original study.

7.2.2 C# systems

For C#, lower use of subtyping among CC edges is found when compared to the original study and the Java replication. The median system has a proportion of 65,3%. The relatively lower proportions are consistent, with all quartiles having lower values when compared to both the Java replication study and the original study. The lowest subtyping proportion found among the C# systems investigated is 4% for CC edges. One system has 100% subtype use.

Results for CI edges show similar findings, again all quartiles have lower values, with a median proportion of 50%. All C# systems investigated contain CI edges, three systems have 100% subtype use. Two systems report zero subtype use.

10 out of 83 systems do not have II edges, and a further 10 show zero subtype usage. A median value of 41% is reported among II edges, with five systems having 100% subtype use.

7.3 Replacing inheritance by composition

For the third research question presented by Tempero et al, the potential of replacing inheritance by composition is investigated. This potential is defined according to the mechanical procedure proposed by Joshua Bloch in his book Effective Java [17]. In order to apply this procedure, there must be a class-class edge that shows internal or external reuse, but makes no use of subtyping. As discussed in section 3.2, Tempero et al report on this by first identifying all CC edges that have either reuse or subtyping. They then count all subtype edges, external reuse edges without subtyping, and mark the remaining edges as internal reuse only. Figure 8 illustrates the averaged values for subtype (**ST**), external reuse but not subtype (**EX-ST**) and internal reuse only (**INO**) edges. Figure 9 shows the distribution among systems for the **EX-ST** and **INO** edges.

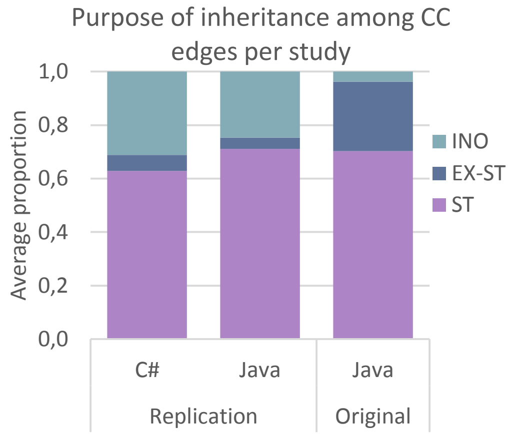


Figure 8: Mean values for the purpose of CC edges across all systems, showing subtype edges (**ST**), external reuse but not subtype (**EX-ST**) and internal reuse only (**INO**). The distribution is shown in Figure 9.

7.3.1 Java replication

The original study reports an average of 26% (median 22%) of CC edges for the external reuse but not subtype (**EX-ST**) category, while this study reports 4% (3% median). This study reports a maximum of 22%, compared to 88% for the original study. 12 out of 86 systems in the replication study show zero external reuse edges that do not have subtype, while the original study reports two systems.

An average of 25% (median 19%) of edges found in the replication study are reported to have internal reuse only, compared to 4% (median 2%) for the original study. The highest proportion found in the replication study is 90%, compared to 30% for the original study. 7 out of 86 systems in the replication study have zero internal reuse only edges, compared to 24 systems in the original study.

When comparing the possibility of replacing inheritance with composition as a whole, disregarding the kind (internal/external) of reuse, this study finds nearly equal potential. A median of around 22% of system-defined CC edges could be redesigned to use composition instead of inheritance, compared to a similar proportion of 24% reported by the original study.

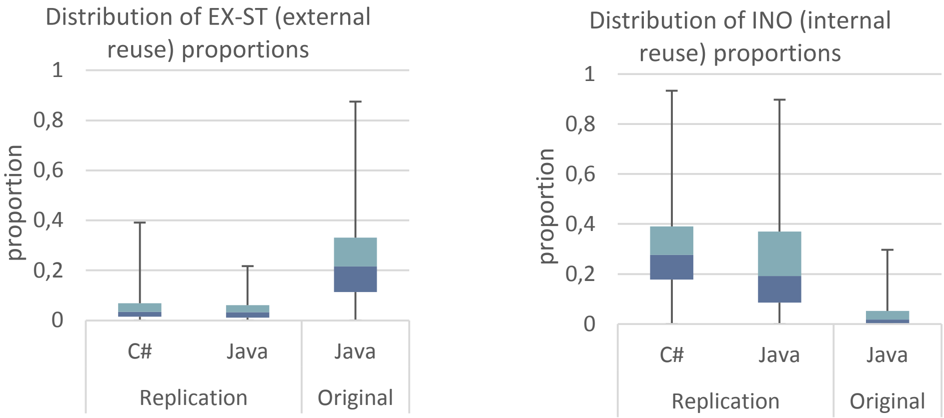


Figure 9: Distribution of internal reuse and external reuse among systems investigated. Note that the external reuse (**EX-ST**) are edges that have shown signs of external reuse but not subtyping, and the internal reuse (**INO**) do not have subtype or external reuse occurrences. Complements Figure 8.

Note the method of counting external reuse and internal reuse: all class-class edges having subtype use, external reuse and/or internal reuse are counted. Subtype proportions are shown as the proportion of edges among those with internal reuse, external reuse or subtype. Those without subtype, but showing external reuse are shown as external reuse (**EX-ST**). Edges without external reuse or subtyping, but showing internal reuse are shown as internal reuse (**INO**) edges. This means that the edges reported to have external reuse in the original study may also have internal reuse. This possibly explains the interesting contradiction shown in Figure 9, and is further discussed in sections 8.1 and 9.4.

7.3.2 C# systems

For C# systems, an average proportion of 6% (median 4%) of edges show external reuse but not subtype. A maximum of 39% is reported, while 8 out of 83 investigated systems show zero external reuse but no subtype usage.

An average proportion of 31% (median 28%) of edges do not show signs of subtype use or external reuse, but only internal reuse. The highest proportion reported is 93%. Two systems show zero signs of internal reuse only.

Section 7.2.2 has shown how C# systems investigated in this study contained lower proportions of subtypes. This directly results in higher reuse proportions due to the reporting model used. On average, 37% of CC edges show potential of replacing inheritance with composition.

7.4 Other uses of inheritance

The fourth research question for the original study investigates other common uses of inheritance. These are edges for which no external reuse, internal reuse or subtype use has been found. The remainder of this section only considers those edges.

Section 6.1 defines the notion of an interface or class solely defining constants. For C#, this study reports zero use of constants-only types among all systems for all types of edges. For CC edges, the original study reports 13 out of 93 systems containing constants classes. Of these, 5 systems had a proportion greater than 1%, the largest being *fitlibraryforfitnesse* with 13% out of 259 edges. The replication study for Java reports three systems with constants CC edges, the highest being 5% for *colt* out of 196 edges. 48 systems in the original study have CI edges with constants occurrences, and 18 had more than 10%. For the Java replication study, 26 systems report constants CI edges, with a maximum of 8%.

Another secondary use of inheritance is the *marker* interfaces, those which have no members defined and all parents are also marker interfaces. The original study finds 32 systems with interfaces solely used as markers. The largest proportion among CI edges found was 47% (*jext* with 43 edges). For the Java replication study, 37 out of 86 systems are found containing marker CI edges. The largest proportion was found for *cobertura*, where 44% of 34 CI edges were marker edges. The C# replication reports similar values, 44 out of 83 systems contain marker edges, with large proportions of 61% for *sandcastle* – 33 edges and *StructureMap* – 55% of 422 CI edges.

Due to analysis limitations discussed in section 9.2, some edges were subjectively suspected of having subtype use from inside external frameworks. These edges receive the *framework* attribute. Another limitation is the use of generics through casting, these may constitute a subtype relationship when cast to a different type after removal from a generic container. These two types of edges are reported as *suspected subtype* (**SUS**) in Figure 10.
For these edges, the original study reports 35 out of 93 systems having generic or framework CC edges. 16 out of these 35 systems were reported as having less than 1%, with a maximum of 17%. For C#, 45 out of 83 systems investigated had use of framework or generic for CC edges, 17 of which had less than 1%. The highest value reported was *ServiceStack* with 23% out of 723 edges. For the Java replication, 47

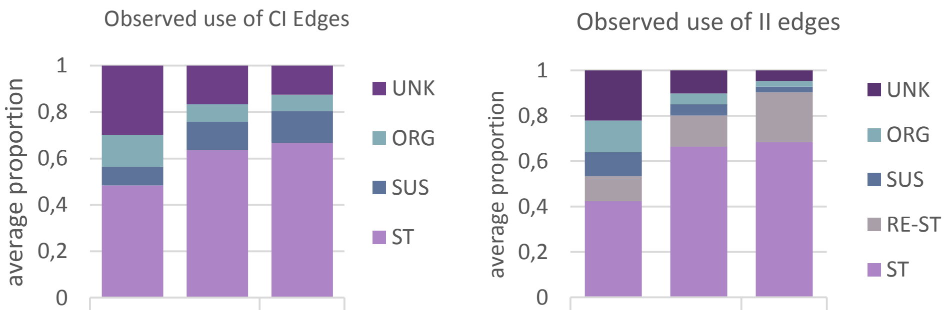


Figure 10: Averaged use of subtype (**ST**), reuse but no subtype (**RE-ST**), suspected subtype (**SUS**), organisational (**ORG**) and unknown purpose (**UNK**) for CI and II edges among studies.

systems contain CC edges with generic or framework, 17 have less than 1% and the maximum reported was 16% for *rssowl* with 370 edges.

For CI edges, the original study reports 55 systems having framework or generic edges, 8 with more than 10% and a maximum of 58%. The C# replication shows 51 systems having CI framework or generic edges, 4 with more than 10% and a maximum of 67%, although this system had only 3 CI edges (*openbastard*). The Java replication reports 62 systems with CI framework or generic edges, 7 having more than 10% and a maximum of 30%.

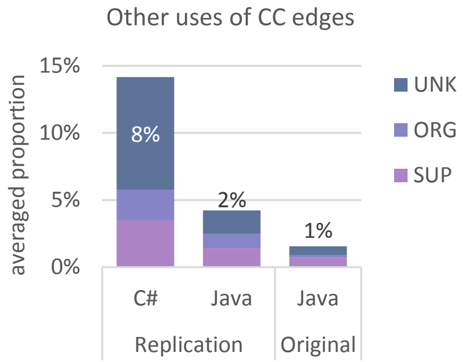


Figure 11: Uses of CC edges that are not subtype, external reuse or internal reuse edges. Super constructor (**SUP**), organisational (**ORG**) or unknown purpose (**UNK**).

The original study reports only a single system with occurrences of framework/generic II edges, *jmeter* with 5% of 20 edges. The C# replication shows 19 systems, 6 with more than 10% and a maximum of 40% for *opentk* with 5 edges. The Java replication reports 14 systems, 5 having more than 10% and a maximum of 41% for *xerces* with 85 edges.

For the remaining edges, the original study reports CC edges where the only use of the relationship is the invocation of a super constructor. Another pattern they found was an (CC, CI, II) edge appearing to have no purpose, but a sibling was used for subtype, internal or external reuse. Those edges receive the *category* attribute. They reason that the parent of such a relationship was playing an organisational role within the implementation. The super constructor is reported as *super* (**SUP**) in Figure 11. The *category* edges are reported as *organisational* (**ORG**) in Figure 10 and Figure 11.

In summary, many uses of inheritance may exist that are not documented in this study, although they are negligible in Java, and are relatively uncommon in C#, with an average of 8%.

8 Analysis

The previous section presented results for the replication study, structured by following the research questions asked in the original study (from section 3.1). This section presents an analysis of these results based on the research questions defined in this study (section 5.1).

Section 8.1 covers the similarities and differences reported for the Java replication study, in order to answer the first research question. For the second research question, section 8.2 discusses results for late-bound self-reference usage in the context of the C# analysis. Section 8.3 investigates the other types of inheritance usage for C# systems, and how they relate to Java inheritance use.

8.1 RQ1: Java replication study

The original study concluded that late-bound self-reference (downcalls) is a feature showing significant practical use, around one third of inheritance relationships employ it. Java developers use inheritance mostly for the purpose of subtyping, with more than two thirds of relationships using some form of subtyping. They found significant opportunity to replace inheritance with composition, at around 22% of relationships. Other uses of inheritance were deemed insignificant, since 99% of inheritance relationships were explained by the previously mentioned usage.

For late-bound self-reference, this replication study has revealed a small discrepancy between results. Section 7.1 shows consistently lower proportions of downcall edges reported for the replication study, when compared to the original study. Two possible causes were found after further investigation.

Appendix D lists an example of a case where downcalls reported by the original study were not found in the source code. This could be caused by having different versions or configurations of source code (even though the system and version information matches). Another possible explanation is that the original study uses bytecode whereas this study employs source code analysis. Employing bytecode analysis may skew results, as discussed in section 5.2.2, however its effect on downcalls has not been determined. Valuable interactions by Cigdem Aytekin, who performed a similar replication study, with Tempero et al confirmed that some of their downcall edges could not be explained.

Unfortunately, method-level data is unavailable from the original study, therefore a definitive explanation of actual causes remains absent. Section 9.5 further discusses the notion of late-bound self-referencing in the context of this study.

With regards to usage subtyping by programmers of Java systems, the results presented in section 7.2 are highly similar to those presented in the original study. 52 out of 86 systems investigated in the replication study are also used in the original study with the same system version, and a further 20 systems are included with a different version. This indicates similar results are to be expected. This study corroborates the finding that subtyping is the dominant type of usage among Java open source systems. At least two thirds of inheritance relationships in Java show some sign of subtyping.

As for the potential of replacing inheritance with composition through the mechanical procedure proposed by Bloch [17], for which an inheritance relationship is required that reuses code from a parent class, but shows no use of subtyping. This study again corroborates the findings by Tempero et al. This is to be expected as the reporting model used in the original study and section 7.2 and 7.3 directly binds subtyping and code reuse together.
An interesting discrepancy was found among internal and external reuse. Further investigation revealed unexplained external reuse edges in the original study, a few examples are available in Appendix D. The number of occurrences of code patterns leading to internal reuse from this study in both Java and C# is around three times the number of external reuse occurrences, as shown in Appendix A. The reported difference cannot be considered critical for the overall conclusion as the aim is to find edges with either internal or external reuse, but no subtyping.

Other uses of inheritance are generally found to be insignificant in this replication study, only 2% of class-class edges are unexplained by previously mentioned kinds of inheritance use. The original study reported 1% of class-class edges to be unexplained.

To summarize in answering this study’s first research question, using source code analysis instead of bytecode analysis is suspected to have a small impact when looking at the inheritance usage metrics defined by Tempero et al. Results from this study are however very similar to those reported in the original study, even though a different set of source systems (although with large overlap) was used. Late-bound self-reference is the only exception of significance, where a median proportional usage of 28% was found compared to 34% for the original study.

8.2 RQ2: Late-bound self-reference in C#

For this study the hypothesis for the second research question indicates fewer downcalls are to be expected for C# systems, methods have to be explicitly marked *virtual*, while this behaviour is implicit in Java, as described in section 4.1.1. The results reported in this study support that hypothesis: a median of 22% is found for the downcall proportion among CC edges for C# systems investigated compared to 28% for the Java replication study and 34% for the original study. These results are consistent among the open source systems investigated, all quartiles show lower downcall proportions for C# systems. Unfortunately, determining if these calls actually happened without being intended by the person that wrote the parent class is not possible without more information about the decision making process underlying the creation of these methods. Qualitatively investigating the effect of language features on the notion of unintended overriding is left for future work, discussed in section 11.

8.3 RQ3: Comparing Java and C#

For the third research question defined in section 5.1, this study investigated the other kinds of inheritance usage defined by Tempero et al. For usage of subtyping, consistently lower values were found amongst the C# open source system investigated, when compared to both the Java replication study and the original study. The median value reported for CC edges is 65%, compared to 76% in both the original study and the replication study.

The lower values for C# systems warrant further investigation. To do this, the causes for subtype edges are investigated. For each CC, CI and II edge, the specific kinds of occurrences (described in section 6.1) that lead to the subtype attribute are measured. These are then aggregated across all systems and grouped by programming language. Note that this information is available only for the replication studies. Figure 12 illustrates these proportions, showing mostly similar values across all kinds of subtype occurrences for Java and C#, with the exception of the variable initializer statement. For this type, the proportion of subtype occurrences caused by variable initializer statements is almost twice as high in Java when compared to C#. A strong suspicion exists that this discrepancy is partially caused by implicitly typed local variables, this is further investigated and its implications discussed in section 9.6.

For the possibility of replacing inheritance with composition in the C# systems investigated, higher values are consistently found when compared to Java. Both the

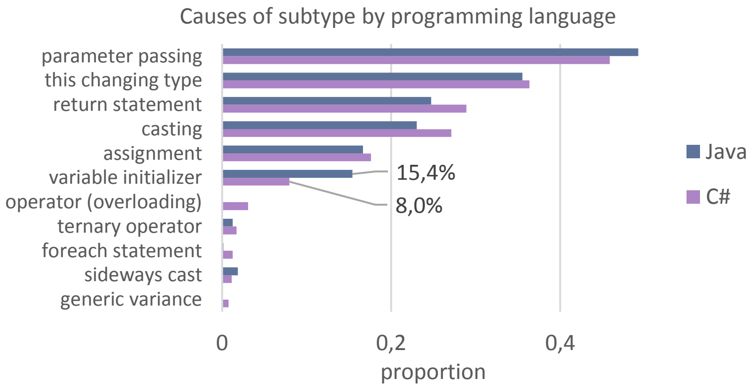


Figure 12: Proportion of subtype edges that have occurrences per kind of expression causing it. Aggregated from 82.000 CC, CI and II edges. Information from the original study is not available in this context.

internal reuse and external reuse measures seem to yield higher proportions when compared to the Java replication study. An average of 37% was found for CC edges, compared to around 30% for the Java systems. The difference between Java and C# in the replication study can be (partially) explained by the reduced amount of subtype usage seen for C# systems, as reduced subtype usage directly implies a larger potential for replacing inheritance with composition.

The relative proportions among internal and external reuse are very similar for the Java and C# systems investigated in the replication study, and the absolute number of occurrences that lead to reuse are similar proportional to the number of CC edges (see Appendix A). This indicates that the amount of reuse seems similar for C# and Java, but the increased amount of subtype usage in Java results in a relatively smaller potential for replacing inheritance by composition in C# systems.

As section 7.4 has illustrated, a general issue related to the C# analysis is found. While the Java original and replication studies report an average of 1% and 2% unknown CC edges, the C# study reports 8% of CC edges that cannot be explained by the kinds of usage defined by Tempero et al. A similar difference is found for CI and II edges. This raises the impression that due to language, programmer culture or other reasons, some other forms of inheritance usage exist that are not contained in the model.

Limited manual inspection was done, investigating the purpose of some of these unexplained inheritance relationships

In *ASP.Net*, entire class hierarchies are found that do not use any form of inheritance, but are only used to test reflective properties of the type hierarchy. For example, five classes named `SubClass…Controller` with different names on the ellipsis inherit from a superclass `BaseClassController`. These are used to test automatically generated API documentation based on the methods defined in these classes.

Another example in C# is the *AutoMapper* project. This is an API allowing developers to map objects' property values across different types. The implementation of this system uses reflection and code generation to map values, showing no apparent use of inheritance from a static perspective in their unit test code, and types are named according to their position in the inheritance tree (`BaseClass`, `DerivedClass`, etc). For this system, 17% of 438 CC edges are unexplained.

In *Math.NET Numerics*, an interesting pattern appears for II edges. The interface `ILinearAlgebraProvider` derives from a generic counterpart `ILinearAlgebraProvider<T>` four times, each with a different type argument. The generic version of this interface defines a large number of operations on different combinations of arrays of T. It seems like this is a form of *method declaration reuse*; a way of creating overloads for all of the operations declared in the generic version of the interface for each of the four type arguments.

In summary, lower subtype usage is found for the C# systems investigated in this study. These could be explained by language features, programmer culture, framework usage or other causes. The ‘var’ feature is likely to have an impact on this, but how much impact and what else it affects remains an open question. The lower subtyping usage directly increases the potential for replacing inheritance with composition. The other potential uses of inheritance appear relatively more significant, and more room is left for the investigation of different kinds of inheritance usage that are not contained in the model defined by Tempero et al.

9 Threats to validity

This section covers the threats to validity for this study. While the impact of some issues has been investigated, others remain open. The threats to validity reported by the original study and later found in the original study are discussed in section 9.1. The *framework problem* is an important threat to validity that is present in both the original study and the replication study as discussed in section 9.2. Important comments can be made on the research method, how the method of reporting may not yield a correct picture of the programmer's way of working. These are discussed in section 9.4 and 9.5. The results show a reduced number of subtype edges for C# systems, which could be partly caused by the language feature 'var'. This is discussed in section 9.6. Section 9.7 discusses the *dynamic language runtime* of .NET and its potential impact on the results of this study. The generalizability of results is discussed in section 9.8. Other minor points of discussion are presented in section 9.9.

9.1 Original study

In section 4.3 of the original study, the authors show an example of potential issues resulting from the analysis of bytecode. As previously discussed in the results section, Appendix D contains a few examples where edges reported by the original study could not be explained. These are the result of manual inspection of source code and emitted bytecode. The impact of these oddities cannot be quantified for the purposes of determining an error margin, therefore this remains a problem with unknown impact.

9.2 Framework problem

The framework problem as Tempero et al describe in section 4.3 of their study exists for both the C# and Java components of the analysis done in this study. Without detailed knowledge of the implementation of external systems, not all relationship attributes can be uncovered. At the time of the study done, neither tool used in this study was capable of creating the required abstract syntax trees from bytecode in external systems. Therefore subtype and reuse edges are still underreported for those that only have occurrences outside of the system boundaries. The impact of this is unknown, however 98% of edges have been explained for the Java analysis, indicating very low impact. For the C# analysis, the gap is larger, since only 92% of edges has been explained. There could be higher framework usage for C#, or other types of inheritance usage that are unknown to this study's research method.

9.3 M3 model and Java ASTs

The Rascal MPL defines a code metadata (M3) model and is able to construct Java syntax trees. At the time of doing this study however, the M3 model does not look outside the boundaries of the system under investigation. For example, if a system class

S extends an external class E, and the external class E extends another external class F, the relationship between S and E is visible, but the relationship between E and F is not. This may introduce false negatives for the subtype metric, because the whole graph may not be uncovered.

For generic types declared in external code, the information related to type arguments is not completely available due to a tool limitation. The type arguments are provided in the form of a list of types, without their corresponding names. Consider the `List<E>` interface in the Java standard library. The type of parameter for the method `List.add(E)` is not available in the AST. Manual inspection leads to an indication that this limitation introduces false negatives for the subtype metric, most profoundly on the commonly used Java interface `Map<K,V>`. Elements added to the `Map` using the `put` method are not reported as a subtype. In an attempt to reduce the amount of false negatives, a heuristic was applied: if a type contains only a single type parameter, the single corresponding type argument is assumed to be the value of that type parameter (`E` in the above case). When no arguments were specified, the `List` was declared as-is, the value of all type parameters is assumed to be `Object`.

9.4 Inheritance Model

The way the model is implemented in both the original and the replication study may not accurately reflect the intentions of the programmer. The subtype and external reuse attributes are assigned to all intermediate edges when an occurrence is found for types that are not directly related. Consider a system containing three classes A, B and C and inheritance edges `A->B` and `B->C`. If subtype or reuse is found for `A->C`, both `A->B` and `B->C` will be attributed, even though the programmer did not define either of the two. When looking solely at the indirect relation between A and C, type B could be removed completely if a direct edge between A and C is created, allowing the code to compile. This implies that there may have been no intent by the programmer to express a subtype relationship for `A->B` or `B->C`. The indirect edge `A->C` is not used in the analysis; only direct relations between types are reported. This enables simplified reporting of the results, since there is no overlap between edges, but

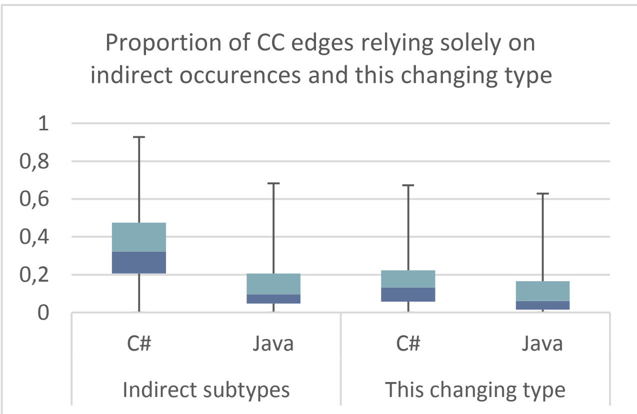


Figure 13: Proportion of CC edges relying solely on *indirect* subtype occurrences (left) and edges relying solely on occurrences of *this changing type* (right).

may reduce accuracy. This limitation is further strengthened by the way results are presented: when a subtype relationship is found, whether direct or indirect, the edge is considered explained and will not be considered for further attributes. The effect of attributing indirect subtype edges is illustrated in Figure 13. While one C# system does not rely on indirect subtype edges at all, some systems rely heavily on indirect edges. Notable are *DashCommerce* with 93% at 243 CC edges and OpenSimulator with 80% at 736 edges. The median system is *NMock* at 32%. For Java, less impact of indirect subtype edges is found, with 14 systems not relying on indirect subtype edges at all, notable systems with high values are *compiere* at 68% (1096 edges) and *jgraphpad* at 64% (246 edges). Future work should address this limitation by refining the conceptual model of the inheritance graph in order to more accurately reflect actual programmers' intention of creating a subtype relationship.

A second point of discussion in relation to subtype edges is the notion of *this changing type*. This is the only measure for which the static type of a variable is not used to determine a subtype relation. While it is true that the variable `this` possibly has a different runtime type, other variables may also show the same behaviour. Consider Code Sample 19, the variable `p` may hold any type assignable to type `P` at runtime. In this case however, the variable assignment does not result in a subtype attribute.

```
class P { }
class C : P { }
P getP();
void M() {
    P p = getP();
}
```

Code Sample 19: The notion of this changing type may apply to other variables.

As shown in Figure 13, the number of edges that solely rely on *this changing type* varies greatly per system, but is significant. For C#, 10 systems do not rely on *this changing type* for occurrences of subtype, while 5 systems report proportions of 50% or above with a maximum of 67% for *FubuMVC* (out of 342 CC edges). The Java replication reports slightly lower proportions, 16 systems do not rely on *this changing type*, with three systems above 50% up to a maximum of 63% for *jOggPlayer* (out of 49 CC edges).

A third potential issue related to the reporting model used lies in the method of counting metrics. All relationship attributes are counted in boolean form, hence it does not matter if a certain relationship has 1 or 100 occurrences of some (downcall, reuse, subtype, etc.) metric. This might skew results if certain kinds of inheritance use are significantly more frequent per relationship than others. This has also been discussed in the original study, but requires significant changes to the reporting model, which are considered of out of scope of this replication study.

Another issue related to the way results are reported is the subjectivity of some of the metrics used. The measures related to the *framework* and *category* attributes are somewhat subjective. The framework attribute is assigned to relationships between types for which the parent type is a descendant of a third party type. The framework attribute helps explain some of the relationships that would otherwise have an unknown purpose. This assumes some use of inheritance inside an external framework, but this is not a guarantee. The same notion applies to the category attribute. If a relationship between two types does not show signs of subtyping or code reuse, but another relationship with the same parent type makes use of subtyping, the relationship is assumed to play some kind of organisational role within the inheritance graph.

For the class-class relationships investigated in the replication study, the proportion of edges that cannot be explained by occurrences of code reuse or subtyping is significantly lower in the replication study than in the original study. Figure 14 illustrates the proportions of explained edges. In the study by Tempero et al, almost all edges could be explained by either external reuse, internal reuse or subtyping, with a median of 99% proportion. For the replication study on the Java open source systems, a lower median of 90% is reported. Results and analysis have indicated that C# programmers may make other use of inheritance relatively more prominently. This is also visible in the proportion of edges explained by subtyping or reuse, a median of 82% is reported.

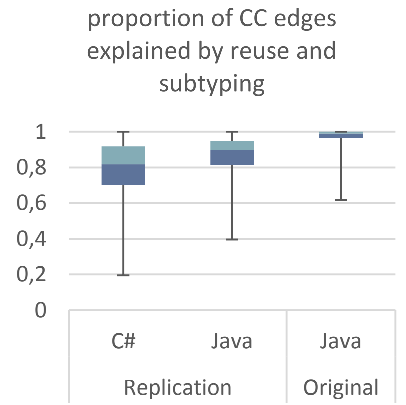


Figure 14: Proportion of CC edges that could be explained by either external reuse, internal reuse or subtyping

The lower proportion of explained inheritance relationships lowers the confidence in the results reported for subtyping use. The first research question presented by Tempero et al. investigated the proportion of subtyping among inheritance relationships. They reported subtyping usage for class-class relationships as a proportion of relationships that could be explained by either code reuse or subtyping. For the original study this is a valid proposition, as virtually all of these edges have been explained. For the replication study these results are less reliable however, as not all edges have been explained by code reuse or subtyping. If subtyping usage for class-class relationships would instead be reported as a proportion of *all* edges, different results are

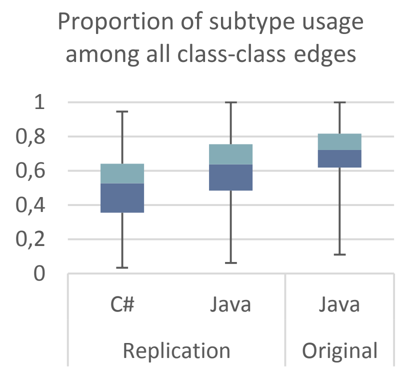


Figure 15: Proportion of subtype usage among *all* class-class relationships, including those not attributed with external or internal reuse.

found. Figure 15 illustrates this issue, distinctly lower proportions of subtyping are reported for the replication study when counting towards *all* subtype edges instead of only those reported having subtype or reuse occurrences.

9.5 Downcall edges

The original study makes the assumption that any overridden method creates a downcall edge when a late-bound self-reference occurs. Code Sample 20 shows a situation where this is not the case. According to the definition of downcall, when the method `target` is invoked by source, the edges `ChildA -> Parent`, `ChildB -> Parent` and `ChildC -> ChildB` receive the downcall attribute. In the class `ChildB` however, the source method is also overridden and it does not invoke the parent method using `super`, removing the possibility of a downcall to `ChildB.target` from `Parent.source`. This extends to down to the class `ChildC` as well, because the method `ChildB.source` is inherited there. This may lead to overreporting actual downcall edges. In addition to the previous constraint, there should be internal or external reuse for the method `source`, since its call to target will never be a downcall unless invoked by an object of a type that derives from `P`.

```
class Parent {
    void source() { target(); }
    void target() { }
}
class ChildA extends Parent {
    void source() {
        super.source();
    }
    void target() { }
}
class ChildB extends Parent {
    void source() {}
    void target() {}
}
class ChildC extends ChildB {
    void target() { }
}
```

Code Sample 20: Example of a situation where false-positive downcall reporting may take place.

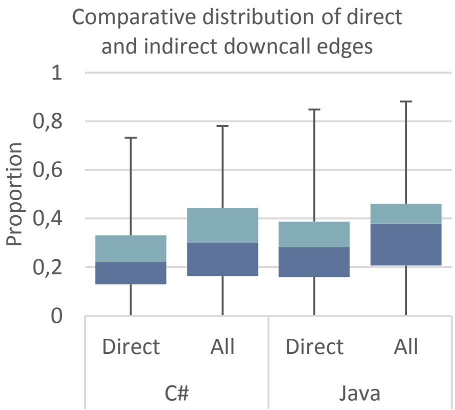
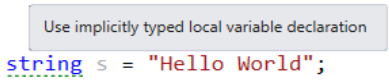


Figure 16: Comparison of downcall attribute when counting intermediate (*indirect*) downcall edges.

Manual inspection of the results of the original study in relation to downcall edges indicates that intermediate edges are not reported for downcalls like the subtype and reuse metrics explained in the previous section. Effectively, only direct edges are reported as downcall edges. When considering Code Sample 20, this would result in only the edges `ChildA -> Parent` and `ChildB -> Parent` being reported. The edge `ChildC -> Parent` is attributed with downcall but omitted from the result set because it is not a direct edge. Figure 16 shows how including intermediate edges in a fashion similar to the reuse and subtype measures, as described in section 9.4, has a significant impact on reporting downcalls, raising the median value for the C# replication from 22% to 30% and the Java replication from 28% to 37%. When reasoning from a programmer's point of view about downcalls, it may be considered that indirect downcalls can be equally intentional as direct downcalls.

9.6 Potential consequences of implicitly typed local variables

Implicitly typed local variables were introduced with C# 3.0 in 2007. They are highly common among C# systems investigated in this study, although with high variance as illustrated in Figure 18. While 8 systems do not employ the syntax feature at all, a quarter of systems have 75% or above of variable declarations using 'var'. The common usage is to be expected as they provide syntactic convenience, IDEs can be configured to enforce their usage and are even required for anonymous types as illustrated in Figure 17. The high variance is also to be expected, the introduction of the *var* keyword in C# spawned extensive discussions relating to whether it improves or reduces code quality [43] [44].




```
var animal = new {
    Name = "Giraffe",
    Height = 6.0
};
```

Figure 17: IDE-assisted implicit local variable declaration and anonymous object creation expression.

Section 8.3 has shown that a significantly reduced amount of subtype relationships occur from the definition of a local variable in C# when compared to Java. Confirming this is (partly) caused by usage of *var* is outside the scope of this study, but one could reason that if a system completely relies on implicitly typed variables, subtyping from variable initializers is zero. As illustrated by Code Sample 21 the effect of implicitly typing a local variable may stretch further than just the initializer statement. It could reduce subtype values for parameter passing, assignment statements and generic

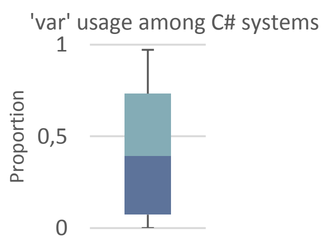


Figure 18: 'var' usage among C# systems investigated. The Y axis represents the proportion of all declarations that use 'var'.

```
class P {
    void ParentMethod() { }
}
class C : P { }

class O {
    static void Method(P p) { }
}

P MethodWithoutVarUsage() {
   P p = new C();     //subtype
   p.ParentMethod(); //no reuse
   O.Method(p);      //no subtype
   return p;         //no subtype
}
P MethodWithoutVarUsage() {
   var p = new C();  //no subtype
   p.ParentMethod(); //reuse
   O.Method(p);      //subtype
   return p;         //subtype
}
```

Code Sample 21: Potential implications of the *var* keyword related to subtype and external reuse.

variance. It may also increase or decrease subtype occurrences from return statements as illustrated in Code Sample 21. Section 11 presents a recommendation for further investigation. This requires tracking individual variables as statements occur, determining if the identifier used was declared with the 'var' keyword.

9.7 Dynamic language runtime

The runtime behaviour of dynamically typed variables in C#, as explained in section 4.1.7, has not been measured for this research. However, the impact has been measured: a count was done on the total number of references to static types versus the dynamic type. These references potentially lead to a subtype, reuse, category or other assignment to an edge. Out of 83 open-source C# systems, 60 systems do not use dynamically typed variables. A further 15 systems have less than 0,01% usage of dynamic variables. The highest usages were found on *Nancy* (1,09%), *Orchard* (0,57%), *Dapper ORM* (0,59%) and *RavenDB* (0,39%). The average proportion referred on all open source C# systems is 0,04%. Therefore the use of `dynamic` in C# does not seem to have a significant impact on the outcome of this study for the systems analysed.

9.8 Generalizability of results

While a considerable number of systems have been investigated in this study, some concerns arise when speaking about the generalizability of results. Firstly, all systems investigated were open source, even though not all systems match the criteria for open source software defined by the Open Source Initiative [45]. An attempt was made to include proprietary software written in C#, however the number of systems (29) acquired and the total size (350 KLOC) was not deemed sufficient for the purposes of studying the usage of inheritance among these systems.

Secondly, systems selected for C# are among the most prominent systems found on Ohloh [33], in terms of usage popularity as well as developer activity. One could speculate that this must have a generally positive effect on the quality of these systems, since more usage and developer support would increase the proportion of faults being detected and solved. A similar notion applies to the Java open-source systems from the Qualitas Corpus [20]; most systems are very large and could not have been built without significant community and user support, or the help of a large corporation.

9.9 Other discussion

Due to time constraints, some occurrences of subtype relations with respect to the use of bounded quantification (type parameter constraints) were not uncovered. Consider the generic interface `I<T>` where type parameter T is constrained by `T extends P`. Any declaration of a variable, parameter or super type definition with a type argument E that is not P requires a subtype relationship to exist between E and P for the code to compile. The original study does not utilize bounded quantification as a means of determining subtype, however future work should include this type of subtyping.

Results have shown consistent lower values across all metrics related to the use of inheritance in C# when compared to Java. One could speculate that this is related to programmer culture, system architecture or other reasons. Whatever the reason, results suggest that programmers in C# use inheritance for relatively more purposes that could not be explained by the model defined by Tempero et al, indicating that the model tailored towards analysing Java code may not be entirely suitable for C# code.

Another important observation can be made with regards to *how much* inheritance is used. For this, the lines of code per system are considered. While lines of code as a metric is subject to many threats to validity, and lines of C# code may not correspond to lines of Java code for various reasons, this study reports an average of 269 lines of code per inheritance edge in C#, versus 186 lines of Java code per inheritance edge. This yields some high level indication that Java programmers use more inheritance than C# programmers.

Creating two implementations of the same analysis tool opens the possibility to compare them. For this purpose a small test library was built in both Java and C#, containing the patterns of code used for the analysis in this research. The two systems are equivalent in terms of types and inheritance relation attributes, although language specific exceptions such as the constants interface are present. This aided in the detection of errors and inconsistencies between the two analysis tools.
Numerous validation sessions with Cigdem Aytekin, who performed the same replication study using Rascal MPL at the Centrum Wiskunde & Informatica (CWI), greatly aided in the verification of results and finding corner cases of relationship attributes. Her interactions with Ewan Tempero provided valuable information with regards to the intent and implementation details of various parts of the original research.

10 Conclusions

The general aim of this study is the validation and extension of the results and conclusions presented in the replicated study. This study presents an investigation of 169 open source Java and C# systems into how inheritance is used by its developers.

To corroborate the results presented in the original study, this study investigated a similar, but different, set of open source Java systems. This study found that slightly less than one third of subclasses (28%) rely on late-bound self-reference (downcalls) to customize the behaviour of superclasses, while the study by Tempero et al reports 34%. Section 9.5 discussed possible reasons for this, such as errors in (interpretation) related to the metrics from the original study, leading to both false positives and false negatives.

For **RQ1**, this study supports the conclusion from the original study in the sense that it indicates late-bound self-reference plays a significant role in the use of inheritance.
For subtyping, this study reports values highly similar to those reported in the original study. It is the dominant use of inheritance, around two thirds of inheritance relationships utilize some form of inheritance. This study also coincides with the original study with respect to replacing inheritance with composition, while the original study reported a median of 24%, this study indicates 22% of edges are candidate for replacing inheritance with composition. Tempero et al conclude that other uses of inheritance are generally insignificant, this study seems to support that conclusion, with around 98% of inheritance usage explained.

For **RQ2**, this study hypothesised that C# programmers should show relatively less usage of late-bound self-reference (downcalls). This was motivated by the fact that unintended overrides appear to exist in Java systems, and C# requires the explicit definition of an overridable method. While this study does report significantly lower values for late-bound self-reference (22%), causality cannot be determined without further qualitative investigation left for future work.

For **RQ3**, results indicate that the proportion of subtyping usage is around 10% lower in the C# systems investigated in this study than those reported for the Java systems. A higher proportion of edges are reported as a candidate for replacing inheritance with composition, at around a third of edges. For other uses of inheritance, results are generally similar to Java, with the exception of edges that could not be explained. 8% of edges could not be explained using the model defined by Tempero et al, compared to 1-2% for Java. This indicates potential other uses of inheritance that are not present in Java systems.

11 Recommendations for future work

An important point of discussion for this study is what Tempero et al call the *framework problem* as described in section 9.2, code declared in external systems is not investigated in the same level of detail as code declared in the system of interest. Future work should address this issue by including the analysis of code within external dependencies. This may introduce higher values for the subtype and reuse related metrics.

As seen in section 9.4, the inheritance model proposed by Tempero et al may not accurately reflect the intentions of the programmer with respect to intermediate edges being attributed. Future work could refine this model by shifting the focus from edge attributes to individual *explicit* occurrences of inheritance use, possibly giving a more accurate insight into the degree and nature of inheritance use. The notion of subtype occurring from *this changing type* should also be carefully evaluated, it is inconsistent in the sense that it is the only type of occurrence that does not rely on differences in static types of variables. Measuring bounded quantification as a subtype occurrence, as explained in section 9.6, should also be considered for future work.

A related issue is reporting actual downcall occurrences instead of potential downcall as explained in section 9.5. Future work should address this issue by ensuring the downcall could actually take place before assigning the attribute. In addition, indirect downcall edges should also be reported to maintain consistency with the subtype and reuse measurements.

One of the most prominent issues related to the collection of data for this study remains the definition of the most appropriate method of empirically investigating systems using quantitative methods. For languages utilizing portable binary code subject to just-in-time compilation, it is evident that loss or obfuscation of information occurs when compiling source code to intermediate bytecode. The analysis of source code has its own issues, including conditional compilation and the difficulty of analysing code from external dependencies: the source code of these dependencies must be obtained in order to generate a unified model of the system under investigation and all code affecting it.

The Qualitas Corpus [20] and the derived Qualitas.class Corpus [28] go a long way in aiding the reproducibility of empirical investigation of software systems, but future work may be able to refine this further. Compiling a corpus of persisted Rascal MPL [41] M3 models and abstract syntax trees would address many issues regarding uncertain or erroneous reporting, while maintaining full traceability to original source code and enabling relatively simple, high-volume and reliable quantitative analyses of empirical data about software systems. Extending the Rascal M3 and AST models to include more programming languages could also be a valuable contribution, allowing simplified comparative studies among languages. Generating full AST and M3 model information from JAR files would also be a valuable contribution to future work. This would address the framework problem for this study while retaining a single non-ambiguous model for future studies.

Results of this study indicate that possibly less use of late-bound self-reference occurs in C# systems when compared to Java systems. Assuming this is true, future work using qualitative methods could investigate if downcalls occur without being intended by the software engineer that created the superclass. Unintended method overriding could be a source of bugs in Java software, for example accidentally defining a method with the same signature or forgetting to invoke the parent method when required.
This study shows significant use of type inference for local variables and illustrates its relation to subtyping and code reuse related to inheritance. An interesting avenue of future research could be the investigation of effects of type inference on inheritance usage. This requires a more in-depth analysis of the behaviour of local variables; tracking them as reuse and subtypes occur in order to determine the actual effect of type inference on inheritance. For example, if a class-interface edge exists solely for the purpose of external reuse, type inference would allow the removal of the inheritance relation and the interface from the system completely and the code would still compile.

Incorporating other C# language features such as extension methods, delegation and anonymous methods into a conceptual model for investigating use of inheritance could be an interesting avenue for future research. This would yield valuable data about how inheritance is used in relation to other patterns.

More generally, replicating "What Programmers Do With Inheritance in Java" on closed-source systems and in other programming languages, considering previous recommendations in this section, would also be a valuable contribution to this field of research, increasing confidence in results and gaining valuable insights into programmers' decision-making with regards to inheritance usage.

12 References

[1] E. Tempero, H. Yul Yang and J. Noble, "What Programmers do with Inheritance in Java," *ECOOP,* vol. 7920, pp. 577-601, 2013.

[2] J. C. Carver, "Towards Reporting Guidelines for Experimental Replications: A Proposal," in *1st International Workshop on Replication in Empirical Software Engineering Research (RESER)*, Cape Town, 2010.

[3] A. S. Jennifer Greene, Head First C#, 3rd Edition, O'Reilly Media, 2013.

[4] B. B. Kathy Sierra, Head First Java, 2nd Edition, O'Reilly Media, 2005.

[5] B. M. Harwani, Learning Object-Oriented Programming in C# 5.0, Cengage Learning, 2014.

[6] E. Tempero, J. Noble and H. Melton, "How Do Java Programs Use Inheritance? An Empirical Study of Inheritance in Java Software," *Lecture Notes in Computer Science,* vol. 5142, pp. 667-691, 2008.

[7] R. Harrison, S. Counsell and R. Nithi, "Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems," *Journal of Systems and Software,* vol. 52, no. 2-3, pp. 173-179, 2000.

[8] J. Daly, A. Brooks, J. Miller, M. Roper and M. Wood, "The effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study," in *International Conference on Software Maintenance*, Opio, 1995.

[9] M. Cartwright and M. Shepperd, "An Empirical View of Inheritance," 1998.

[10] L. Prechelt, B. Unger, M. Philippsen and W. Tichy, "A controlled experiment on inheritance depth as a cost factor for code maintenance," *The Journal of Systems & Software,* vol. 65, no. 2, pp. 115-126, 2003.

[11] M. Cartwright and M. Shepperd, "An Empirical Investigation of an Object-Oriented Software System," *IEEE Transactions on Software Engineering,* vol. 26, no. 8, pp. 876-796, 2000.

[12] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software ,* vol. 20, no. 6, pp. 467-493, 1994.

[13] B. Henderson-Sellers, Object-oriented metrics: measures of complexity, Prentice-Hall, Inc, 1995.

[14] A. Taivalsaari, "On the Notion of Inheritance," *ACM Computing Surveys,* vol. 28, no. 3, 1996.

[15] B. Meyer, “The many faces of inheritance: A taxonomy of taxonomy,” *IEEE Computer,* vol. 29, no. 5, pp. 105-108, 1996.

[16] R. Lämmel, R. Linke, E. Pek and A. Varanovich, “A Framework Profile of .NET,” *20th Working Conference on Reverse Engineering,* pp. 141-150, 2011.

[17] J. Bloch, Effective Java, Addison-Wesley, 2008.

[18] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns, Reading: Addison Wesley Publishing Company, 1994.

[19] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton and J. Noble, “Qualitas corpus: A curated collection of Java code for empirical studies,” *Asia Pacific Software Engineering Conference,* pp. 336-345, 2010.

[20] E. Börger and R. F. Stärk, “Exploiting Abstraction for Specification Reuse. The Java/C# Case Study,” *Lecture Notes in Computer Science,* vol. 3188, pp. 42-76, 2004.

[21] Microsoft, “var (C# Reference),” [Online]. Available: http://msdn.microsoft.com/en-us/library/bb383973.aspx. [Accessed 2 October 2014].

[22] Microsoft, “Boxing and Unboxing (C# Programming Guide),” [Online]. Available: http://msdn.microsoft.com/en-us/library/yz2be5wk.aspx.

[23] J. Gosling, B. Joy, G. Steele, G. Bracha and A. Buckley, “The Java® Language Specification,” 28 February 2013. [Online]. Available: http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf. [Accessed 10 February 2014].

[24] ECMA International, “Common Language Infrastructure (CLI) Partitions I to VI,” June 2012. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf. [Accessed Feburary 2014].

[25] Microsoft, “C# Language Specification 5.0,” 7 June 2013. [Online]. Available: http://www.microsoft.com/en-us/download/details.aspx?id=7029.

[26] Oracle, “Type Erasure,” [Online]. Available: http://docs.oracle.com/javase/tutorial/java/generics/erasure.html.

[27] R. Terra, L. F. Miranda, M. T. Valente and R. S. Bigonha, “Qualitas.class Corpus: A Compiled Version of the Qualitas Corpus,” *Software Engineering Notes,* vol. 38, no. 5, pp. 1-4, 2013.

[28] GitHub Inc., “GitHub,” [Online]. Available: https://github.com/.

[29] H. Deitel and P. Deitel, Java: How to program, Fonenix inc., 2011.

[30] J. Bloch and N. Gafter, Java Puzzlers: Traps, Pitfalls and Corner Cases, Pearson Education, 2005.

[31] J. Gosling, B. Joy, L. G. Steele, G. Bracha and A. Buckley, The Java Language Specification, Java SE 8 Edition, Addison-Wesley Professional, 2014.

[32] Ohloh, "Ohloh, the open source network," [Online]. Available: http://www.ohloh.net/.

[33] F. Logozzo and M. Fähndrich, "On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis," in *Compiler Construction: 17th International Conference*, Budapest, 2008.

[34] Microsoft, "Ngen.exe (Native Image Generator)," [Online]. Available: http://msdn.microsoft.com/en-us/library/6t9t5wcf(v=vs.110).aspx. [Accessed 6 October 2014].

[35] Excelsior Jet, [Online]. Available: http://www.excelsiorjet.com/. [Accessed 6 October 2014].

[36] Proguard, "ProGuard - Frequently Asked Questions," [Online]. Available: http://proguard.sourceforge.net/index.html#FAQ.html. [Accessed 24 February 2014].

[37] E. Lippert, "What does the optimize switch do?," 11 06 2009. [Online]. Available: http://blogs.msdn.com/b/ericlippert/archive/2009/06/11/what-does-the-optimize-switch-do.aspx. [Accessed 24 February 2014].

[38] PostSharp, "PostSharp.net," [Online]. Available: http://www.postsharp.net/. [Accessed 22 September 2014].

[39] Microsoft Research, "ILMerge," [Online]. Available: http://research.microsoft.com/en-us/people/mbarnett/ilmerge.aspx. [Accessed 2014 September 22].

[40] CWI, SWAT group, "Rascal Metaprogramming language," [Online]. Available: http://www.rascal-mpl.org. [Accessed 2014].

[41] D. Grunwald, "Using NRefactory for analyzing C# code," 11 August 2012. [Online]. Available: http://www.codeproject.com/Articles/408663/Using-NRefactory-for-analyzing-Csharp-code.

[42] Community, "Use of 'var' keyword in C#," [Online]. Available: http://stackoverflow.com/questions/41479/use-of-var-keyword-in-c-sharp. [Accessed 3 October 2014].

[43] J. Allen, "C# Debate: When should you use var?," [Online]. Available: http://www.infoq.com/news/2008/05/CSharp-var. [Accessed 3 October 2014].

[44] Open Source Initiative, "The Open Source Definition (annotated) version 1.9," [Online]. Available: http://opensource.org/osd-annotated.

[45] E. Tempero, "Inheritance Use Data," [Online]. Available: https://www.cs.auckland.ac.nz/~ewan/qualitas/studies/inheritance/. [Accessed 25 September 2014].

[46] G. Booch, Object-Oriented Analysis and Design, 2nd edition, Santa Clara, California: Addison-Wesley, 1998.

Appendix A. Analysis statistics

This appendix gives an indication of the amount of data processed to obtain results presented in this study. It also briefly summarizes the size of the tools used in the analysis. All source code and data is available at the following url:
https://github.com/basbrekelmans/inheritance-msc

	Measure	Java	C#
Input	Number of source/binary files	713.951	444.781
	Size of source/binary files	12,0GB	26,8GB
	Number of systems	86	83
	Lines of source code (thousands)	11.673	11.176
	Number of Eclipse/Visual Studio projects	904	1.898
	Size of compressed M3 and AST files (binary format)	3,6GB	
Analysis	Analysis tool language	Rascal	C#
	Analysis tool lines of code	1.475	3.252
	Time to compute AST and M3 files	70h	
	Number of visitation passes over ASTs[2]	1	3
	Running time to analyse all projects[3]	±9h	±45m
	Number of CSV files in output[4]	6.192	830
	Size of CSV files	528MB	288MB
Database	Database size	1,4GB	
	CSV -> database tool language	C#	
	CSV -> database tool lines of code	1279	
	Time to process & insert data	±10m	
	Time to calculate all metrics	±4s	
	Number of tables	11	
	Number of views	29	
	SQL Lines of Code to create database schema	2224	
	Downcall occurrences	80.704	81.614
	Number of edges (CC, CI, II)	79.061	64.715
	Subtype occurrences	642.719	502.398
	Internal reuse occurrences	624.870	551.150
	External reuse occurrences	237.367	172.145

[2] Due to the M3 model provided by Rascal MPL, the full context of types, dependencies and declared methods is already present. This has to be built up in C# before being able to determine edge attributes.
[3] C# analysis runs single threaded on optimized code without a debugger attached, Java analysis runs on 3 threads with precomputed AST and M3 files. Both analyses were run on a PC with 8GB memory, an SSD and a Core i7-4500U CPU that is otherwise idle. Author has no experience optimizing Rascal code. This is not valid as a benchmark.
[4] Java CSV files are per Eclipse project, C# files per system

Appendix B. List of open source C# systems analysed

All systems analysed were pulled from the main branch (usually *master*) and updated on September 19, 2014. All edges are between types within the system under investigation and are direct relations. This list was compiled with the help of Ohloh [33], an online repository of open-source systems. Note that some systems were developed by companies that published the source code, including but not limited to DB4O – Versant; ASP.Net, EntityFramework and Roslyn – Microsoft, MindTouch Dream & Deki – MindTouch. There may be a question of definition of *open-source*, since some of these systems do not allow contributions from any member of the public. This may conflict with the definition as presented by the Open Source Initiative [45].

Name	KLOC[5]	CC Edges	CI Edges	II Edges
Accord.NET	144	175	309	24
AForge.NET	47	150	112	1
ASP.NET	309	1140	510	14
Autofac	36	149	222	11
AutoFixture	31	21	139	6
AutoMapper	26	438	151	18
Axiom	217	593	288	1
Banshee	106	359	255	28
BLToolkit	360	909	272	24
Boo	117	448	165	62
BugNET	74	217	72	0
Caliburn	25	149	131	33
Castle.Core	51	286	271	112
Castle.Windsor	68	426	489	51
CruiseControl.NET	141	375	450	15
CSLA	185	91	152	71
Dapper	21	12	18	1
dashCommerce	68	243	10	1
DB4O	194	1705	1870	123
DotNetKicks	24	164	1	0
DotNetNuke	186	470	242	25
DotNetOpenAuth	57	268	188	44
Elmah	8	28	10	0
EntityFramework	553	3105	325	23
FileHelpers	46	305	35	0

[5] Small parts of some systems could not be loaded due to missing dependencies or build errors. This is the number of physical code lines that were actually analysed, in thousands.

Name	KLOC[5]	CC Edges	CI Edges	II Edges
FlashDevelop	179	167	124	7
F-Spot	73	153	73	3
FubuMVC	74	342	514	14
Gallio	219	894	487	35
gendarme	64	621	276	11
GitExtensions	97	224	66	6
GMap.NET	78	193	195	14
ikvm9	172	409	39	1
ILSpy	265	1357	565	57
Lucene.Net	229	1808	134	26
MassTransit	62	423	652	144
MathNet.Numerics	601	223	96	4
SignalR	70	197	95	3
Migrator.NET	5	43	18	2
MindTouchDeki	137	444	72	2
MindTouchDream	52	92	98	0
MonoCsharp	70	439	99	8
MonoDevelop	548	1779	793	113
Moq	28	15	42	53
MVCContrib	27	99	76	12
n2cms	155	1290	643	62
Nancy	67	237	203	9
NAnt	55	324	25	3
Newtonsoft.Json	84	288	50	4
NGenerics	52	330	32	4
NHibernate.Everything	353	2218	888	177
NLog.netfx45	39	301	31	0
NMock2	10	66	56	14
NSubstitute	15	216	96	3
NuGet	129	151	235	31
openbastard	1	11	3	0
OpenSim	330	736	642	21
OpenTK	588	124	80	5
Orchard	135	727	921	360
ORMBattle.NET	33	51	2	0
Proto	149	165	117	7
Quartz	37	97	137	9
RavenDB	366	2368	284	17
Reflexil	137	598	179	29
Rhino Mocks	21	60	51	6
Roslyn	1001	1889	385	141
Sandcastle	76	283	33	2
ScrewTurnWiki	61	86	56	9
SdlDotNet	34	87	13	0

Name	KLOC[5]	CC Edges	CI Edges	II Edges
ServiceStack	127	723	677	113
SharpDevelop	540	2889	1235	170
SharpOS	71	160	38	2
SolrNet	23	103	152	10
Spring.Net	215	847	1034	96
StructureMap	33	184	422	15
SubSonic.Linq	36	132	77	4
tasque	17	29	46	19
Textile	5	36	4	0
TweetSharp	68	19	32	2
WatiN	29	246	42	3
WorldWind	203	344	78	3
xunit	58	304	194	93

Appendix C. List of open source Java systems analysed

All systems studied were downloaded from the Qualitas.class [28] corpus. Out of 111 systems, only 86 were usable due to compiler errors, memory limitations on the tools used or missing source code.

Name	In original?[6]	KLOC[7]	CC Edges	CI Edges	II Edges
ant-1.8.2	DV	128	937	332	21
antlr-3.4	DV	47	181	52	3
aoi-2.8.1	Yes	110	221	142	2
axion-1.0-M2	Yes	24	132	108	13
c_jdbc-2.0.2	Yes	96	463	36	0
castor-1.3.3	DV	263	1213	272	24
cayenne-3.0.1	Yes	192	1926	561	12
checkstyle-5.6	DV	37	349	29	2
cobertura-1.9.4.1	Yes	55	17	34	0
collections-3.2.1	No	55	366	237	11
colt-1.2.0	Yes	36	196	285	3
columba-1.0	Yes	92	143	96	8
compiere-330	No	401	1650	387	2
derby-10.9.1.0	DV	651	1556	685	131
displaytag-1.2	Yes	20	178	39	2
emma-2.0.5312	Yes	21	76	79	16
exoportal-v1.0.2	Yes	96	1050	296	41
findbugs-1.3.9	Yes	111	458	418	28
fitjava-1.1	Yes	3	66	0	0
fitlibraryforfitnesse	DV	47	508	265	24
freecol-0.10.3	DV	106	555	130	1
freecs-1.3.20100406	Yes	23	61	25	0
galleon-2.3.0	Yes	61	232	78	0
ganttproject-2.1.1	DV	49	293	256	17

[6] **DV** indicates a different version of this system was used in the replication study, **Yes** indicates the same version was used, **No** indicates the system was not included in the original study.

[7] Small parts of some systems could not be loaded due to missing dependencies or build errors. This is the number of physical code lines that were actually analysed, in thousands.

Name	In original?[6]	KLOC[7]	CC Edges	CI Edges	II Edges
geotools-9.2	No	684	2464	1210	529
hadoop-1.1.2	No	320	1293	935	54
hsqldb-2.0.0	Yes	144	205	98	9
htmlunit-2.8	Yes	101	705	89	0
informa-0.7.0-alpha2	Yes	14	40	64	46
iReport-3.7.5	Yes	218	713	110	0
itext-5.0.3	Yes	78	191	99	4
ivatagroupware-0.11.3	No	29	21	25	0
jag-6.1	No	16	25	20	0
jasml-0.10	Yes	6	22	2	0
jasperreports-3.7.4	No	170	780	606	273
javacc-5.0	No	15	60	8	0
jboss-5.1.0	No	85	157	189	18
jchempaint-3.0.1	Yes	213	1197	598	55
jedit-4.3.2	Yes	110	245	177	0
jext-5.0	Yes	60	350	94	0
jFin_DateMath-R1.0.1	Yes	9	20	2	0
jfreechart-1.0.13	Yes	143	294	286	35
jgraph-5.13.0.0	Yes	32	120	76	3
jgraphpad-5.10.0.2	Yes	24	236	28	0
jgrapht-0.8.1	Yes	17	103	117	6
Jgroups-2.10.0	Yes	96	328	229	10
jhotdraw-7.5.1	Yes	80	348	145	14
jmeter-2.5.1	DV	95	460	323	3
jmoney-0.4.4	Yes	8	21	16	0
jOggPlayer-1.1.4s	Yes	30	49	25	0
jpf-1.5.1	DV	13	39	44	19
jrefactory-2.9.19	Yes	123	816	134	1
Jruby-1.7.3	DV	244	1203	2029	29
JSPWiki-2.8	Yes	60	173	94	6
jsXe-04	Yes	18	37	15	0
jtopen-7.1	Yes	342	807	306	13
log4j-2.0-beta	DV	33	154	103	10
lucene-4.2.0	DV	413	3684	426	44
marauroa-3.8.1	Yes	18	75	31	0
maven-3.0.5	DV	66	95	230	11
megamek-0.35.18	Yes	243	1283	213	10
mvnforum-1.2.2-ga	Yes	105	107	289	2

Name	In original?[6]	KLOC[7]	CC Edges	CI Edges	II Edges
nakedobjects-4.0.0	Yes	134	1595	841	349
nekohtml-1.9.14	Yes	8	14	9	0
netbeans-7.3	No	1928	7504	4317	1204
openjms-0.7.7-beta-1	Yes	39	232	143	9
oscache-2.3	DV	8	27	10	4
pmd-4.2.x	DV	61	484	131	3
poi-3.6	Yes	203	842	327	30
proguard-4.9	DV	63	310	644	4
quartz-1.8.3	No	29	67	139	5
quickserver-1.4.7	No	18	20	59	0
quilt-0.6-a-5	Yes	8	20	35	0
rssowl-2.0.5	No	101	370	230	72
sablecc-3.2	DV	28	174	33	1
springframework-3.0.5	DV	234	1577	938	82
struts-2.2.1	Yes	143	1096	454	22
sunflow-0.07.2	Yes	22	16	110	8
tapestry-5.1.0.5	Yes	97	398	868	65
tomcat-7.0.2	Yes	181	679	362	43
velocity-1.6.4	Yes	27	203	110	13
wct-1.5.2	No	48	137	97	14
webmail-0.7.10	Yes	10	36	51	2
weka-3-6-9	DV	273	871	1265	9
xalan-2.7.1	Yes	184	568	236	118
xerces-2.10.0	Yes	126	371	260	85

Appendix D. Cases of unexplained attribute assignments

This section defines a few interesting cases where the original study reported attributes for relationships that could not be explained. For each case, all source code potentially leading to the assignment of an attribute is included. Attributes marked with **bold red** could not be found in the source code.

System	marauroa-3.8.1
Child type	marauroa.server.game.messagehandler.OutOfSyncHandler
Parent type	marauroa.server.game.messagehandler.MessageHandler
Relationship	Class-Class
Attributes	Category, Internal Reuse (method & field), **External Reuse (Method Call)**, Subtype

All code referencing OutOfSyncHandler

src/marauroa/server/game/messagehandler/OutOfSyncHandler.java

```
class OutOfSyncHandler extends MessageHandler {
     ...
       @Override
       public void process(Message message) {
        ...
        //Internal Reuse (field access)
        PlayerEntry entry = playerContainer.get(clientid);
        //Internal Reuse (method call)
        if (!isValidEvent(msg, entry, ClientState.GAME_BEGIN)) {
            ...
        }
        ...
       }
}
```

src/marauroa/server/game/messagehandler/MessageDispatcher.java

```
public class MessageDispatcher {
    private Map<MessageType, MessageHandler> handlers ...
    ...
       private void initMap() {
        ...
        //subtype
        handlers.put(C2S_OUTOFSYNC, new OutOfSyncHandler());
        ...
       }
    ...
}
```

System	marauroa-3.8.1
Child type	marauroa.server.db.adapter.H2DatabaseAdapter
Parent type	marauroa.server.db.adapter.AbstractDatabaseAdapter
Relationship	Class-Class
Attributes	Category, Internal Reuse (method & field), **External Reuse (Method Call), Subtype**, Super

All code referencing H2DatabaseAdapter:

src/marauroa/server/db/adapter/H2DatabaseAdapter.java

```
public class H2DatabaseAdapter extends AbstractDatabaseAdapter {
    ...
       public H2DatabaseAdapter(...) {
         super(connInfo); //super
       }
       @Override
    protected Connection createConnection(...) {
        //internal reuse (method)
        Connection con = super.createConnection(connInfo);
        ...
    }
    ...
       @Override
       public boolean doesTableExist(...) {
         //internal reuse (field)
         DatabaseMetaData meta = connection.getMetaData();
        ...
       }
}
```

src/marauroa/server/db/adapter/H2DatabaseAdapterTest.java

```
public class H2DatabaseAdapterTest {
    ...
    public void testRewriteSql() {
       H2DatabaseAdapter adapter = new H2DatabaseAdapter();
       //rewriteSql is overridden by H2DatabaseAdapter
       assertThat(adapter.rewriteSql(""), equalTo(""));
       ...
    }
}
```

Note that this class is instantiated by means of reflection, depending on the system configuration. No other static references exist.

System	cobertura-1.9.4.1
Child type	net.sourceforge.cobertura.javancss.parser.java15.Token.GTToken
Parent type	net.sourceforge.cobertura.javancss.parser.java15.Token
Relationship	Class-Class (child is also an inner class of the parent)
Attributes	Cast, **Category**, Single, **Downcall**, **External Reuse (Method Call)**, Subtype

Note: this edge reports both **Category** and **Single**. These attributes should be mutually exclusive by definition (see 0 for the list of definitions by Tempero et al.)

All code referencing GTToken:

src/net/sourceforge/cobertura/javancss/parser/java15/Token.java

```
public class Token {
  ...
  public static final Token newToken(int ofKind)
  {
     switch(ofKind)
     {
       ...
       //subtype through return
       case JavaParser15Constants.GT:
          return new GTToken();
     }
  }

  public static class GTToken extends Token
  {
     int realKind = JavaParser15Constants.GT;
  }
}
```

src/net/sourceforge/cobertura/javancss/parser/java15/JavaParser15TokenManager.java

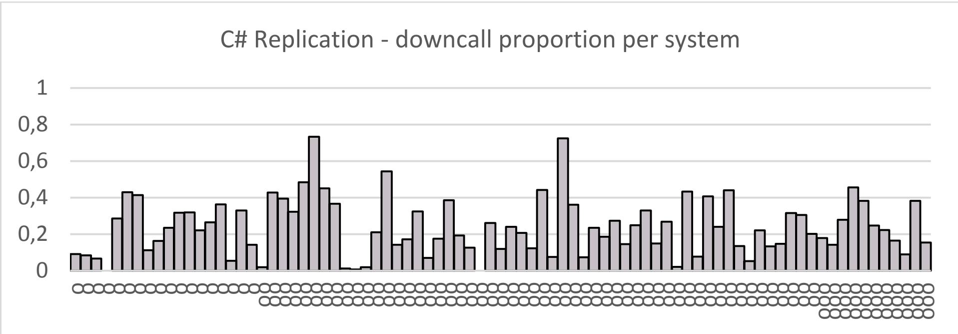
```
void TokenLexicalActions(Token matchedToken)
{
     ...
     //cast, four other similar cases omitted
     ((Token.GTToken)matchedToken).realKind = RUNSIGNEDSHIFT;
     ...
}
```

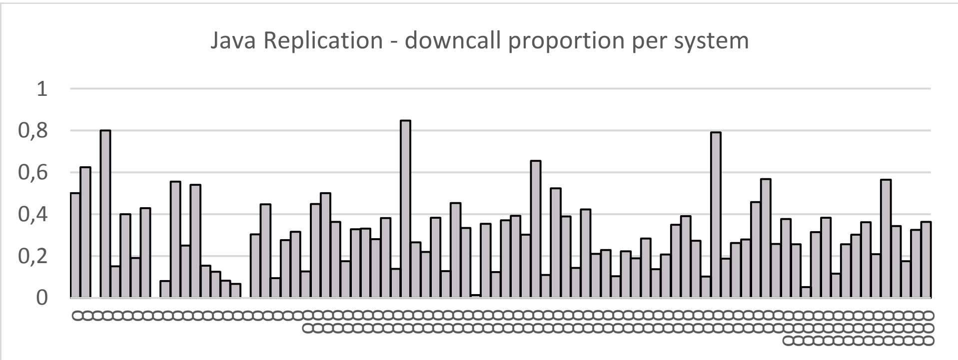
Appendix E. Detailed data

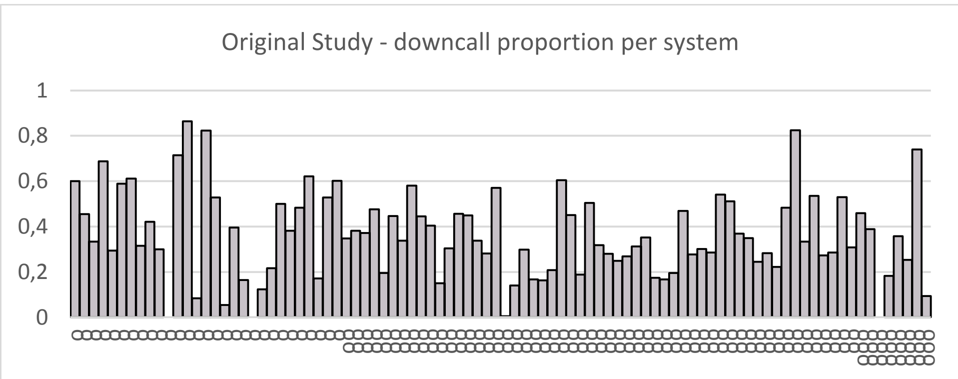
This appendix presents system-by-system data for metrics used in the results section. Some charts have system size defined on the x-axis in the form of “o”, “oo” and “ooo”. This indicates the order of magnitude of size, as the number of edges. A single “o” means the system has less than 100 edges, “oo” means less than 1000 and “ooo” means less than 10.000. See https://github.com/basbrekelmans/inheritance-msc for all data.

Downcall proportions

Shows downcall distribution among systems, related to Figure 5 on page 28.

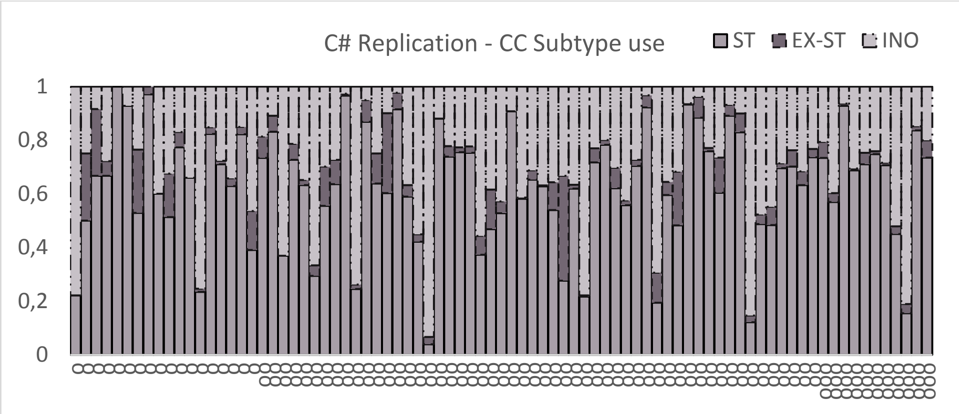


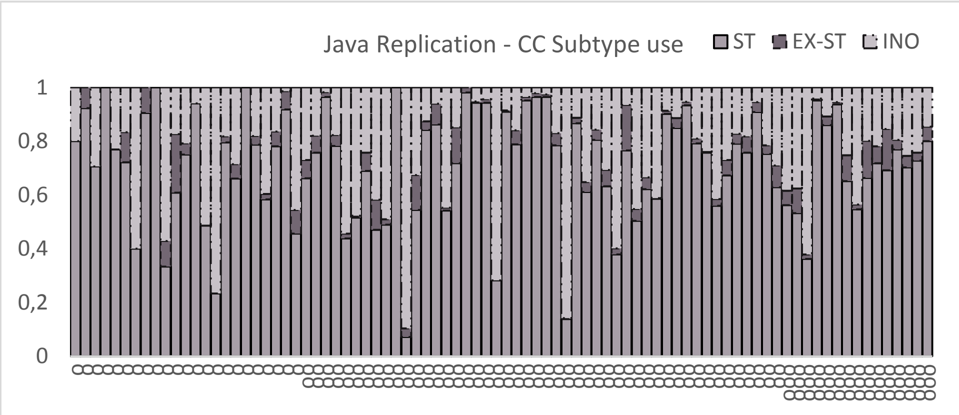


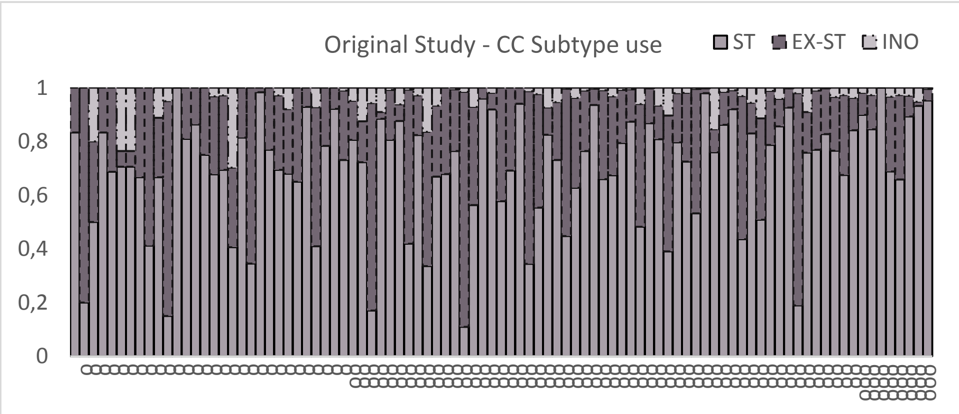


Subtype/Reuse for CC edges

Shows the relative proportions of subtype (**ST**), external reuse but not subtype (**EX-ST**) and internal reuse only (**INO**) among systems, ordered by size. Data shown here is presented in Figure 6 (page 29) and Figure 8 (page 31).

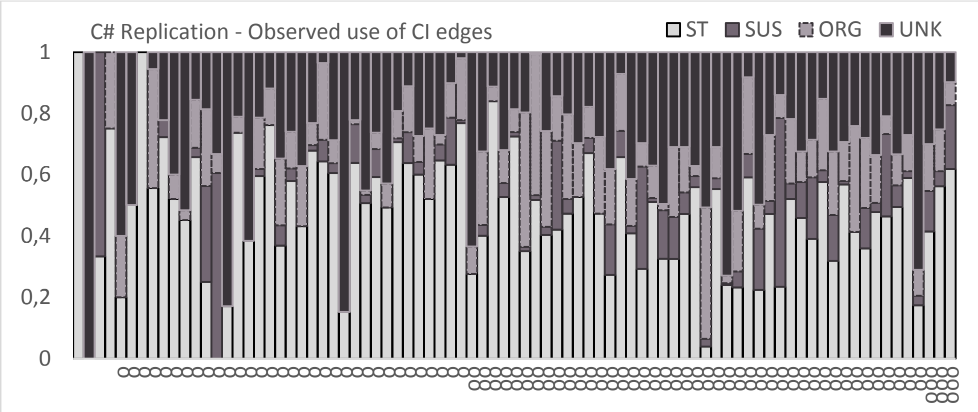


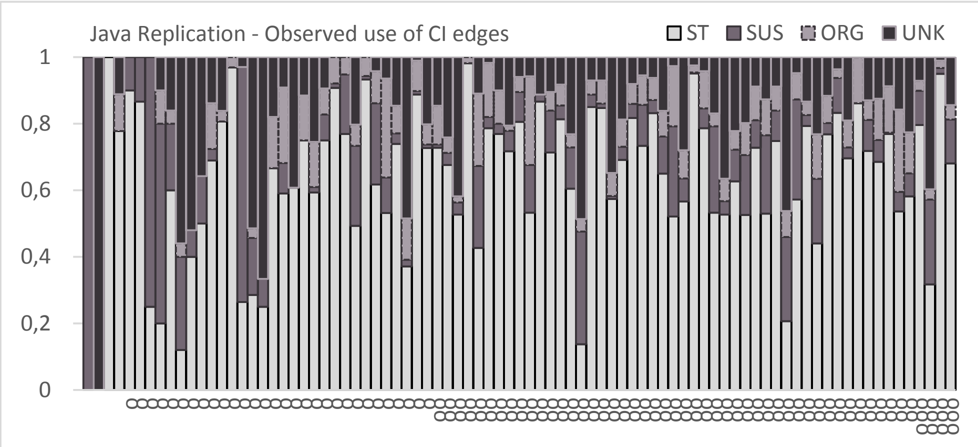


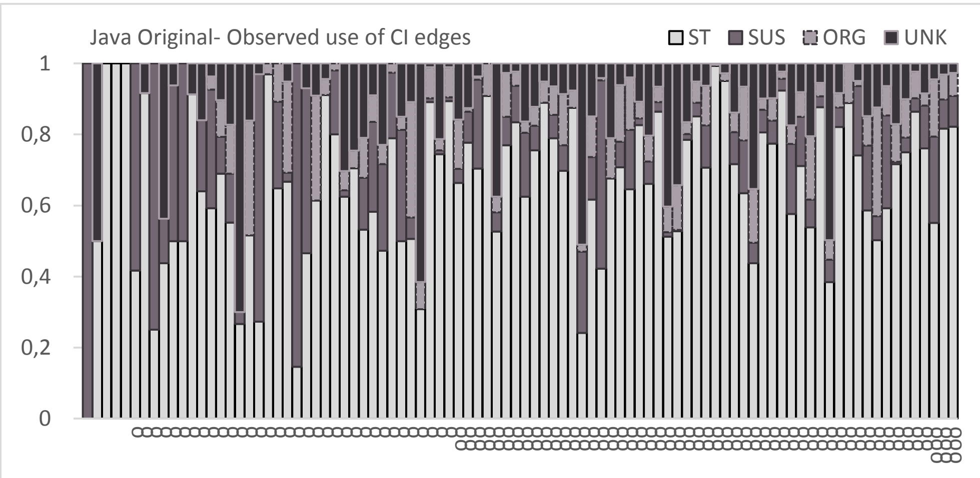


Usage of CI Edges

Shows use of subtype (**ST**), suspected subtype (**SUS**), organisational (**ORG**) and unknown purpose (**UNK**) for CI edges. Complements Figure 7 (page 30) and Figure 10 (page 33).

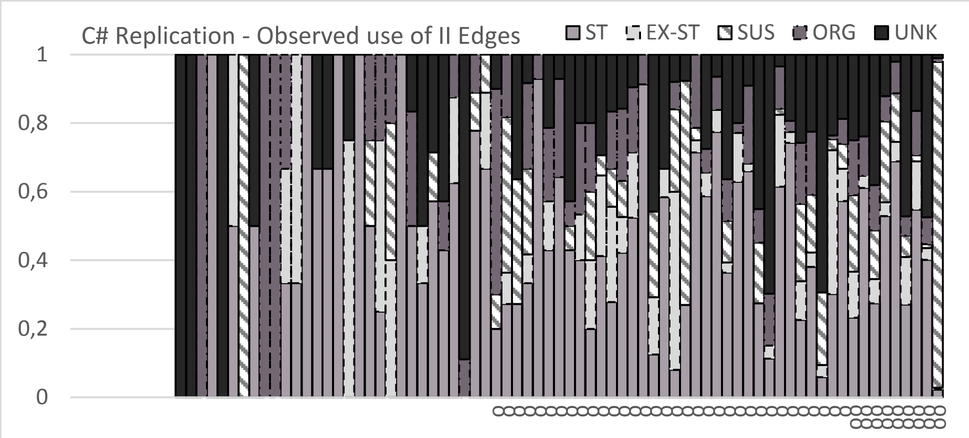


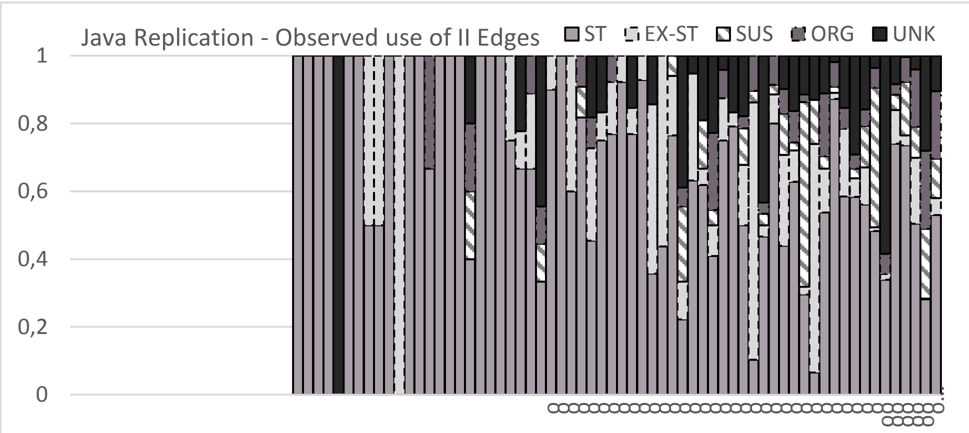


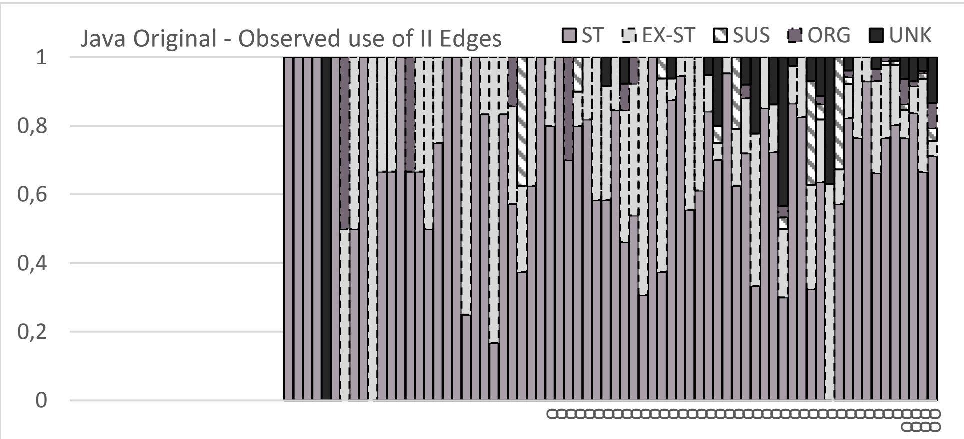


Usage of II edges

Shows use of subtype (**ST**), external reuse but not subtype (**EX-ST**), suspected subtype (**SUS**), organisational (**ORG**) and unknown purpose (**UNK**) for CI edges. Complements Figure 7 (page 30) and Figure 10 (page 33).

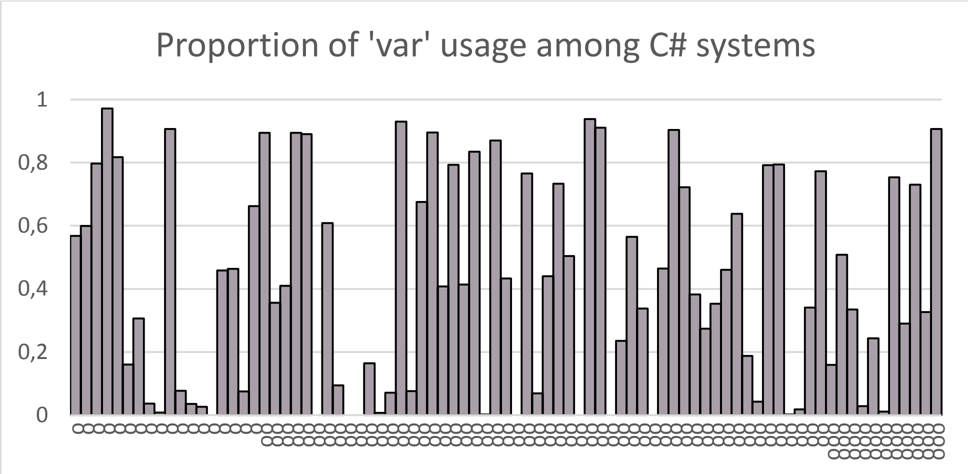






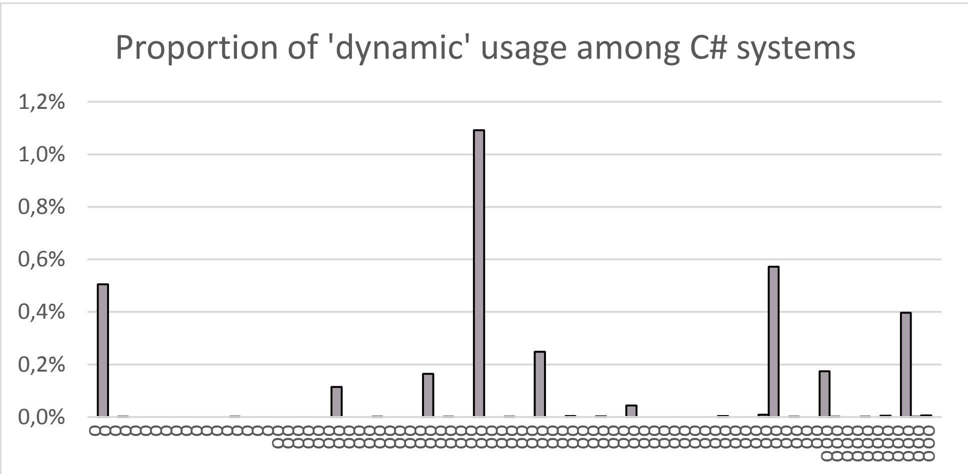
Var keyword

Show the proportion of variables declared using 'var' compared to the total variable declarations per C# system. The x-axis represents the number of CC edges. This represents the data presented in section 9.6 - Figure 18 (page 45).



Dynamic use

Show the proportion of times a reference was made to the dynamic type versus any static type per C# system. The x-axis represents the number of CC edges. Note that this is only relevant for C# systems. This represents the summary presented in section 9.7. The y-axis has a different scale than previous data, the maximum is 1.2% instead of 100%.



Appendix F. Summary of metrics

The table below reports median values for the metrics used in the results of this study. For a more detailed list of descriptions related to these values see the page by Tempero at the following url: https://www.cs.auckland.ac.nz/~ewan/qualitas/studies/inheritance/docs.html

Metric Name	Description	Replication C#	Replication Java	Original Java
nExplicitCC	Number of system defined CC edges	268	241	228
pCCUsed	CC edges used (subtype + external and internal reuse)	0,82	0,90	0,99
pCCDC	CC edges with downcalls	0,22	0,28	0,34
pCCSubtype	CC edges with subtyping as the proportion of pCCUsed	0,65	0,75	0,76
pCCExreuseNoSubtype	CC edges with external reuse and without subtyping as the proportion of pCCUsed	0,03	0,03	0,22
pCCUsedOnlyInRe	CC edges used only in internal reuse as the proportion of pCCUsed	0,28	0,19	0,02
pCCUnexplSuper	CC edges that are not used, but show super constructor use	0,01	0,00	0,00
pCCUnexplCategory	CC edges that do not show super constructor use, but have the *Category attribute*	0,01	0,00	0,00
pCCUnknown	CC edges not explained by above metrics	0,05	0,01	0,00
nExplicitCI	Number of system defined CI edges	134	133	127
pOnlyCISubtype	CI edges having subtype use	0,50	0,69	0,69
pExplainedCI	CI edges not having subtype but have one of *Framework, Generic, Marker or Constants* attributes	0,03	0,07	0,07
pCategoryExplCI	CI edges having the *Category* attribute, but none of the above	0,12	0,07	0,05
pUnexplainedCI	CI edges not explained by above metrics	0,27	0,12	0,08
nExplicitII	Number of system defined II edges	11	8	6
pIISubtype	II edges having subtype use	0,41	0,67	0,72
pOnlyIIReuse	II edges showing external reuse, but not subtyping	0,04	0,06	0,17
pExplainedII	II edges not having subtype or reuse but have one of *Framework, Generic, Marker or Constants* attributes	0,00	0,00	0,00
pCategoryExplII	II edges having the *Category* attribute, but none of the above	0,09	0,00	0,00
pUnexplainedII	II edges not explained by above metrics	0,17	0,00	0,00

Appendix G. Code listings

This is an extraction of some of the important bits of source code used to analyse the data presented in the results. All source code is available at the GitHub repository, see https://github.com/basbrekelmans/inheritance-msc.

Three samples are included – the code that visits ASTs for C# code, the main rascal visiting code and a view that calculates metrics.

C# - Ast visiting

This is the class *CallVisitor* in the C# analysis tool. It uses context information (types, methods) to extract facts from source code files.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using CSharpInheritanceAnalyzer.Model.Relationships;
using CSharpInheritanceAnalyzer.Model.Types;
using ICSharpCode.NRefactory.CSharp;
using ICSharpCode.NRefactory.CSharp.Resolver;
using ICSharpCode.NRefactory.Semantics;
using ICSharpCode.NRefactory.TypeSystem;

namespace CSharpInheritanceAnalyzer.ViewModel
{
    public class CallVisitor : VisitorBase
    {
        public CallVisitor(CSharpAstResolver resolver, IDictionary<string, CSharpType> types,
            List<IInheritanceRelationship> edges, HashSet<string> ownCodeAssemblyNames)
            : base(resolver, types, edges, ownCodeAssemblyNames)
        {
        }

        public override void VisitConditionalExpression(ConditionalExpression conditionalExpression)
        {
            base.VisitConditionalExpression(conditionalExpression);
            Expression left = conditionalExpression.TrueExpression;
            Expression right = conditionalExpression.FalseExpression;

            ResolveResult leftResolve = Resolver.Resolve(left);
            ResolveResult rightResolve = Resolver.Resolve(right);

            if (leftResolve.IsError || rightResolve.IsError) return;
            CreateSubtypeRelation(conditionalExpression, rightResolve.Type, leftResolve.Type,
SubtypeKind.Assignment,
                right is ThisReferenceExpression);
            CreateSubtypeRelation(conditionalExpression, leftResolve.Type, rightResolve.Type,
SubtypeKind.Assignment,
                left is ThisReferenceExpression);
        }

        public override void VisitInvocationExpression(InvocationExpression invocationExpression)
        {
            base.VisitInvocationExpression(invocationExpression);
            var result = Resolver.Resolve(invocationExpression) as InvocationResolveResult;
            if (result == null)
            {
                Trace.WriteLine(String.Format("Unknown invocation resolution at {0}",
invocationExpression));
                return;
            }
            CSharpType methodDeclaringType = GetTypeOrCreateExternal(result.Member.DeclaringType);
            CheckCallForSubtype(invocationExpression.Arguments, result.Member);

            CSharpType targetDeclaringType = GetTypeOrCreateExternal(result.TargetResult.Type);
            ResolveResult currentDeclaringTypeResolve =
                Resolver.Resolve(invocationExpression.GetParent<TypeDeclaration>());
```

```
            if (currentDeclaringTypeResolve.IsError) return;
            var currentMethod = invocationExpression.GetParent<MethodDeclaration>();
            string fromReference = currentMethod == null ? "(field initializer)" : currentMethod.Name;
            var currentDeclaringType = (Class) GetTypeOrCreateExternal(currentDeclaringTypeResolve.Type);
            if (currentDeclaringType.IsChildOf(methodDeclaringType))
            {
                IEnumerable<IInheritanceRelationship> items = currentDeclaringType.GetPathTo(methodDeclaringType);
                bool direct = currentDeclaringType.IsDirectChildOf(methodDeclaringType);
                foreach (IInheritanceRelationship item in items)
                {
                    item.InternalReuse.Add(new Reuse(direct, ReuseType.MethodCall, result.Member.Name,
                        currentDeclaringType, fromReference));
                }
            }
            else if (targetDeclaringType.IsChildOf(methodDeclaringType))
            {
                IEnumerable<IInheritanceRelationship> items = targetDeclaringType.GetPathTo(methodDeclaringType);
                bool direct = targetDeclaringType.IsDirectChildOf(methodDeclaringType);
                foreach (IInheritanceRelationship item in items)
                {
                    item.InternalReuse.Add(new Reuse(direct, ReuseType.MethodCall, result.Member.Name,
                        currentDeclaringType, fromReference));
                }
            }

            if (result.IsVirtualCall &&
                (currentDeclaringType == methodDeclaringType || currentDeclaringType.IsChildOf(methodDeclaringType)))
            {
                Method method = CreateMethod(result.Member);
                //maybe a downcall somewhere
                foreach (
                    CSharpType downcallCandidate in
                        methodDeclaringType.AllDerivedTypes().Where(t => t.DeclaredMethods.Contains(method)))
                {
                    IInheritanceRelationship relation = downcallCandidate.GetImmediateParent(methodDeclaringType);
                    relation.Downcalls.Add(
                        new Downcall(relation.BaseType, relation.DerivedType, method, fromReference));
                }
            }
        }

        private void CheckCallForSubtype(IEnumerable<Expression> args, IParameterizedMember member)
        {
            IEnumerator<IParameter> paramsEnumerator = EnumerateParameters(member).GetEnumerator();
            IEnumerator<Expression> argumentsEnumerator = args.GetEnumerator();

            while (argumentsEnumerator.MoveNext() & paramsEnumerator.MoveNext())
            {
                Expression argument = argumentsEnumerator.Current;
                IParameter parameter = paramsEnumerator.Current;
                ResolveResult argumentResolve = Resolver.Resolve(argument);
                CreateSubtypeRelation(argument, argumentResolve.Type, parameter.Type, SubtypeKind.Parameter,
                    argument is ThisReferenceExpression);
            }
        }

        private IEnumerable<IParameter> EnumerateParameters(IParameterizedMember member)
        {
            bool isParams = false;
            int i = 0;
            while (i < member.Parameters.Count || isParams)
            {
                yield return member.Parameters[i];
                isParams |= member.Parameters[i].IsParams;
                if (!isParams)
                    ++i;
```

```
            }
        }

        public override void VisitObjectCreateExpression(ObjectCreateExpression objectCreateExpression)
        {
            base.VisitObjectCreateExpression(objectCreateExpression);
            //constructor call, can never be internal/external reuse
            ResolveResult resolve = Resolver.Resolve(objectCreateExpression);

            if (resolve.IsError)
            {
                Trace.WriteLine("Could not resolve constructor: " + objectCreateExpression);
                return;
            }

            if (resolve is ConversionResolveResult)
            {
                //found an occurrance of "new Action(MyMethod)" pattern
                //don't care about those
                return;
            }

            if (resolve is DynamicInvocationResolveResult)
            {
                //cannot do something with dynamic invocation
                return;
            }

            CheckCallForSubtype(objectCreateExpression.Arguments, ((InvocationResolveResult) resolve).Member);

            IType test = resolve.Type;
        }

        public override void VisitVariableInitializer(VariableInitializer variableInitializer)
        {
            base.VisitVariableInitializer(variableInitializer);
            IType leftType = null;
            ResolveResult result = Resolver.Resolve(variableInitializer);
            if (result.IsError)
            {
                Trace.WriteLine("Error resolving: " + variableInitializer);
                return;
            }
            var memberResult = result as MemberResolveResult;
            if (memberResult != null)
            {
                leftType = memberResult.Member.ReturnType;
            }
            else
            {
                var localResult = result as LocalResolveResult;
                if (localResult != null)
                {
                    leftType = localResult.Variable.Type;
                }
                else
                {
                    Debugger.Break();
                }
            }
            ResolveResult initializerResolve = Resolver.Resolve(variableInitializer.Initializer);

            CreateSubtypeRelation(variableInitializer, initializerResolve.Type, leftType,
                SubtypeKind.VariableInitializer, variableInitializer.Initializer is ThisReferenceExpression);
        }

        public override void VisitCastExpression(CastExpression castExpression)
        {
            base.VisitCastExpression(castExpression);

            ResolveResult targetTypeResolve = Resolver.Resolve(castExpression);
            ResolveResult fromTypeResolve = Resolver.Resolve(castExpression.Expression);
            CSharpType leftType = GetTypeOrCreateExternal(targetTypeResolve.Type);
            CSharpType rightType = GetTypeOrCreateExternal(fromTypeResolve.Type);
```

```
            if (rightType.IsObject)
            {
                leftType.HasBeenCastFromObject = true;
            }
            CreateSubtypeRelation(castExpression, targetTypeResolve.Type, fromTypeResolve.Type, SubtypeKind.Cast,
                castExpression.Expression is ThisReferenceExpression);
            CreateSubtypeRelation(castExpression, fromTypeResolve.Type, targetTypeResolve.Type, SubtypeKind.Cast,
                castExpression.Expression is ThisReferenceExpression);
        }

        public override void VisitReturnStatement(ReturnStatement returnStatement)
        {
            base.VisitReturnStatement(returnStatement);
            Expression expr = returnStatement.Expression;
            ResolveResult exprResolve = Resolver.Resolve(expr);
            CSharpType exprType = GetTypeOrCreateExternal(exprResolve.Type);
            IType returnType = TryGetReturnType(returnStatement);
            if (returnType != null)
            {
                CreateSubtypeRelation(returnStatement, exprResolve.Type, returnType, SubtypeKind.Return,
                    returnStatement.Expression is ThisReferenceExpression);
            }
        }

        private IType TryGetReturnType(AstNode node)
        {
            IType resolvedType = TryGetEntityDeclarationReturnType(node);
            if (resolvedType == null)
            {
                var anonymousMethodExpression = node.GetParent<AnonymousMethodExpression>();
                if (anonymousMethodExpression == null) return null;
                AstNode parent = anonymousMethodExpression.Parent;
                if (parent is AssignmentExpression)
                {
                    resolvedType = Resolver.Resolve(((AssignmentExpression) parent).Left).Type;
                }
                else if (parent is VariableInitializer)
                {
                    resolvedType = Resolver.Resolve(((VariableDeclarationStatement) parent.Parent).Type).Type;
                }
            }
            return resolvedType;
        }

        private IType TryGetEntityDeclarationReturnType(AstNode node)
        {
            var method = node.GetParent<EntityDeclaration>();
            if (method != null)
            {
                ResolveResult resolve = Resolver.Resolve(method);
                if (!resolve.IsError)
                    return resolve.Type;
            }
            return null;
        }

        public override void VisitAsExpression(AsExpression asExpression)
        {
            base.VisitAsExpression(asExpression);

            ResolveResult targetTypeResolve = Resolver.Resolve(asExpression);
            ResolveResult fromTypeResolve = Resolver.Resolve(asExpression.Expression);
            CSharpType rightType = GetTypeOrCreateExternal(fromTypeResolve.Type);
            CSharpType leftType = GetTypeOrCreateExternal(targetTypeResolve.Type);
            if (rightType.IsObject)
            {
                leftType.HasBeenCastFromObject = true;
            }
            CreateSubtypeRelation(asExpression, fromTypeResolve.Type, targetTypeResolve.Type, SubtypeKind.Cast,
                asExpression.Expression is ThisReferenceExpression);
            CreateSubtypeRelation(asExpression, targetTypeResolve.Type, fromTypeResolve.Type, SubtypeKind.Cast,
```

```
            asExpression.Expression is ThisReferenceExpression);
        }

        public override void VisitAssignmentExpression(AssignmentExpression assignmentExpression)
        {
            base.VisitAssignmentExpression(assignmentExpression);
            //subtype occurs if left type is a base class of right type
            ResolveResult resolveLeft = Resolver.Resolve(assignmentExpression.Left);
            ResolveResult resolveRight = Resolver.Resolve(assignmentExpression.Right);
            CreateSubtypeRelation(assignmentExpression, resolveRight.Type, resolveLeft.Type, SubtypeKind.Assignment,
                assignmentExpression.Right is ThisReferenceExpression);
        }

        private void CreateSubtypeRelation(AstNode node,
            IType right, IType left, SubtypeKind kind, bool isRightTypeThis)
        {
            CSharpType leftType = GetTypeOrCreateExternal(left);
            CSharpType rightType = GetTypeOrCreateExternal(right);

            EntityDeclaration currentMethod = node.GetParent<MethodDeclaration>() ??
                                              node.GetParent<ConstructorDeclaration>() as EntityDeclaration ??
                                              node.GetParent<PropertyDeclaration>();
            string fromReference = currentMethod == null ? "(field initializer)" : currentMethod.Name;
            ResolveResult currentDeclaringTypeResolve = Resolver.Resolve(node.GetParent<TypeDeclaration>());
            fromReference += " in " + currentDeclaringTypeResolve.Type.FullName;

            //left is the parent, right is the child
            for (int i = 0; i < left.TypeArguments.Count && i < right.TypeArguments.Count; i++)
            {
                IType leftArg = left.TypeArguments[i];
                IType rightArg = right.TypeArguments[i];
                CreateSubtypeRelation(node, leftArg, rightArg, SubtypeKind.CovariantTypeArgument, false);
                CreateSubtypeRelation(node, rightArg, leftArg, SubtypeKind.ContravariantTypeArgument, false);
            }

            if (rightType.IsChildOf(leftType))
            {
                rightType.HasSubtypeToObject |= leftType.IsObject;
                IEnumerable<IInheritanceRelationship> relations = rightType.GetPathTo(leftType);
                foreach (IInheritanceRelationship item in relations)
                {
                    item.Subtypes.Add(new Subtype(item.BaseType == leftType && item.DerivedType == rightType, kind,
                        fromReference));
                }
            }
            if (isRightTypeThis && kind == SubtypeKind.Parameter)
            {
                foreach (CSharpType derivedType in rightType.AllDerivedTypes())
                {
                    IInheritanceRelationship relation = derivedType.GetImmediateParent(rightType);
                    relation.Subtypes.Add(new Subtype(derivedType.IsDirectChildOf(rightType),
                        SubtypeKind.ThisChangingType, fromReference));
                }
            }
        }

        public override void VisitIdentifierExpression(IdentifierExpression identifier)
        {
            base.VisitIdentifierExpression(identifier);
            //prevent duplicate entries from member reference
            if (identifier.GetParent<MemberReferenceExpression>() != null) return;
            ResolveResult resolveResult = Resolver.Resolve(identifier);
            var memberResolve = resolveResult as MemberResolveResult;
            //variable access without this qualifier
            if (memberResolve != null && memberResolve.Member.DeclaringType.Kind != TypeKind.Enum)
            {
                CSharpType targetType = GetTypeOrCreateExternal(memberResolve.Member.DeclaringType);
                ResolveResult currentDeclaringTypeResolve = Resolver.Resolve(identifier.GetParent<TypeDeclaration>());
```

```
                //it is possible that we are inside an enumeration, inside a nested type, referencing an identifier
                //defined in the outer type. In that case, we want to use the outer type as the source
                //22-9: fixed refernce to boolean constant defined in outer type
                if (currentDeclaringTypeResolve.Type.Kind == TypeKind.Enum
                    || currentDeclaringTypeResolve.Type.Kind == TypeKind.Interface
                    || currentDeclaringTypeResolve.Type.Kind == TypeKind.Delegate)
                {
                    currentDeclaringTypeResolve =
Resolver.Resolve(identifier.GetParent<TypeDeclaration>().GetParent<TypeDeclaration>());
                }

                string currentReferenceName = identifier.GetParent<MethodDeclaration>() == null
                    ? "(field initializer)"
                    : identifier.GetParent<MethodDeclaration>().Name;
                var currentDeclaringType = (Class) GetTypeOrCreateExternal(currentDeclaringTypeResolve.Type);
                bool possibleUpCall = currentDeclaringType.IsChildOf(targetType);
                if (possibleUpCall)
                {
                    bool direct = currentDeclaringType.IsDirectChildOf(targetType);
                    foreach (IInheritanceRelationship item in currentDeclaringType.GetPathTo(targetType))
                    {
                        item.InternalReuse.Add(new Reuse(direct, ReuseType.FieldAccess, memberResolve.Member.Name,
                            currentDeclaringType, currentReferenceName));
                    }
                }
            }
        }

        public override void VisitMemberReferenceExpression(MemberReferenceExpression memberReferenceExpression)
        {
            OnVisitMemberReference(memberReferenceExpression);
            foreach (AstNode astNode in memberReferenceExpression.Children)
            {
                astNode.AcceptVisitor(this);
            }
        }

        public override void VisitForeachStatement(ForeachStatement foreachStatement)
        {
            base.VisitForeachStatement(foreachStatement);
            AstType variableType = foreachStatement.VariableType;
            IType variableTypeResolve = Resolver.Resolve(variableType).Type;
            ResolveResult enumerableResolution = Resolver.Resolve(foreachStatement.InExpression);
            if (enumerableResolution.IsError) return;

            ParameterizedType enumerableInterfaceBase =
                enumerableResolution.Type.GetAllBaseTypes().OfType<ParameterizedType>()
                    .FirstOrDefault(
                        t => t.Kind == TypeKind.Interface && t.FullName ==
"System.Collections.Generic.IEnumerable");
            IType elementType;
            if (enumerableInterfaceBase != null)
            {
                elementType = enumerableInterfaceBase.TypeArguments[0];
            }
            else if (enumerableResolution.Type.Kind == TypeKind.Array)
            {
                elementType = ((ArrayType) enumerableResolution.Type).ElementType;
            }
            else if (enumerableResolution.Type.Kind == TypeKind.Dynamic)
            {
                DynamicUsage++;
                return;
            }
            else if (enumerableResolution.Type.Kind == TypeKind.Unknown)
            {
                //unbound generic or unknown element type;
                return;
            }
            else
```

```
            {
                IType nonGenericEnumerableBase = enumerableResolution.Type.GetAllBaseTypes()
                    .FirstOrDefault(b => b.Kind == TypeKind.Interface && b.FullName == "System.Collections.IEnumerable");
                if (nonGenericEnumerableBase != null)
                {
                    elementType =
                        nonGenericEnumerableBase.GetAllBaseTypes().FirstOrDefault(t => t.FullName == "System.Object");
                }
                else
                {
                    //corner case: Only implements GetEnumerator()
                    IMethod method =
                        enumerableResolution.Type.GetMethods()
                            .FirstOrDefault(m => m.Name == "GetEnumerator" && m.Parameters.Count == 0);
                    IProperty property;
                    if (method != null &&
                        (property =
                            method.ReturnType.GetProperties()
                                .FirstOrDefault(p => p.Name == "Current" && p.CanGet)) != null)
                    {
                        elementType = property.ReturnType;
                    }
                    else
                    {
                        Trace.WriteLine("Unresolved foreach statement at " + foreachStatement);
                        return;
                    }
                }
            }
            CreateSubtypeRelation(foreachStatement, elementType, variableTypeResolve, SubtypeKind.Foreach,
                foreachStatement.InExpression is ThisReferenceExpression);
            CreateSubtypeRelation(foreachStatement, variableTypeResolve, elementType, SubtypeKind.Foreach,
                foreachStatement.InExpression is ThisReferenceExpression);
        }

        private void OnVisitMemberReference(MemberReferenceExpression memberReferenceExpression)
        {
            ResolveResult resolveResult = Resolver.Resolve(memberReferenceExpression);
            var methodGroupResolve = resolveResult as MethodGroupResolveResult;
            var memberResolve = resolveResult as MemberResolveResult;
            if (methodGroupResolve != null)
            {
                //handled by invocation
            }
            else if (memberResolve != null)
            {
                CSharpType memberDeclaringType = GetTypeOrCreateExternal(memberResolve.Member.DeclaringType);
                ResolveResult target = memberResolve.TargetResult;
                ResolveResult currentTypeResolve =
                    Resolver.Resolve(memberReferenceExpression.GetParent<TypeDeclaration>());
                if (currentTypeResolve.IsError) return;
                string currentReferenceName = memberReferenceExpression.GetParent<MethodDeclaration>() == null
                    ? "(field initializer)"
                    : memberReferenceExpression.GetParent<MethodDeclaration>().Name;

                CSharpType currentType = GetTypeOrCreateExternal(currentTypeResolve.Type);
                CSharpType targetType = GetTypeOrCreateExternal(target.Type);
                bool possibleDownCall = currentType.IsParentOf(memberDeclaringType);
                bool possibleUpCall = currentType.IsChildOf(memberDeclaringType);
                bool externalReuse = !possibleUpCall && targetType.IsChildOf(memberDeclaringType);
                bool isDirectRelation = false;

                IEnumerable<IInheritanceRelationship> upcallRelations = null;
                IEnumerable<IInheritanceRelationship> externalReuseRelations = null;
                if (possibleUpCall)
                {
                    upcallRelations = currentType.GetPathTo(memberDeclaringType);
                    isDirectRelation = currentType.IsDirectChildOf(memberDeclaringType);
                }
                if (externalReuse)
```

```
                {
                    externalReuseRelations = targetType.GetPathTo(memberDeclaringType);
                    isDirectRelation = targetType.IsDirectChildOf(memberDeclaringType);
                }

                ReuseType reuseType;
                switch (memberResolve.Member.SymbolKind)
                {
                    case SymbolKind.Field:
                        reuseType = ReuseType.FieldAccess;
                        //downcall not possible
                        break;
                    case SymbolKind.Property:
                    case SymbolKind.Indexer:
                    case SymbolKind.Event:
                    case SymbolKind.Operator:
                    case SymbolKind.Constructor:
                        //upcall for "Super"
                    case SymbolKind.Destructor:
                        reuseType = ReuseType.MethodCall;
                        break;
                    default:
                        throw new ArgumentOutOfRangeException();
                }
                if (possibleUpCall)
                {
                    foreach (IInheritanceRelationship item in upcallRelations)
                    {
                        item.InternalReuse.Add(new Reuse(isDirectRelation, reuseType,
                            memberResolve.Member.Name,
                            (Class) currentType, currentReferenceName));
                    }
                }
                if (externalReuse)
                {
                    foreach (IInheritanceRelationship item in externalReuseRelations)
                    {
                        item.ExternalReuse.Add(new Reuse(isDirectRelation, reuseType,
                            memberResolve.Member.Name,
                            (Class) currentType, currentReferenceName));
                    }
                }
            }
            //other cases: Type/Namespace access; not relevant for this case.
        }
    }
}
```

Rascal – Main visitor code

This code visits ASTs and delegates to various functions that determine if a relevant fact such as a subtype occurrence is present. File is named ‘Main.rsc’

```
module Main

import lang::java::jdt::m3::Core;    //code analysis
import lang::java::m3::AST;          //code analysis
import util::Resources;              //projects()
import IO;                           //print
import Relation;                     //invert
import List;                         //size
import Map;                          //size
import Set;                          //takeOneFrom
import String;                       //split
import ValueIO;                      //readBinaryValueFile

import FileInfo;                     //getBasePath;
import Types;                        //inheritance context
import TypeHelper;                   //method return type
import ModelCache;                   //loading models
```

```
import InheritanceType;              //inheritance types (CC/CI/II)
import Subtype;
import ExternalReuse;
import InternalReuse;
import Downcall;
import Super;
import Generic;



public void analyzePreloaded() {
    loc baseLoc = defaultStoragePath();
    set[loc] done = {};
    if (exists(baseLoc + "done.locset"))
        done = readBinaryValueFile(#set[loc], baseLoc + "done.locset");
    println("Loading <size(done)> projects");
    int i = 0;
    for (p <- done) {
        try {
            i = i + 1;
            print("<i>/<size(done)>: <p.authority>...");
            analyzeProject(p, false);
        }
        catch error: {
            println("Error!!!");
            println(error);
            }
    }
    println("Completed");
}

public void analyzeProject(loc projectLoc) {
    analyzeProject(projectLoc, false);
}

public void analyzeProject(loc projectLoc, bool forceCacheRefresh) {
    M3 model = getM3(projectLoc, forceCacheRefresh);
    asts = getAsts(projectLoc, forceCacheRefresh);

    //if (forceCacheRefresh) {
    //print("Counting LOC....");
    //writeLinesOfCode(projectLoc);
    //println("done");
    //}
    print("Creating additional models....");
    rel[loc, loc] directInheritance = model@extends + model@implements;
    rel[loc, loc] allInheritance = directInheritance+;
    map[loc, loc] declaringTypes = (f:t | <t,f> <- model@containment, t.scheme == "java+enum" ||
t.scheme == "java+class" || t.scheme == "java+interface" || t.scheme == "java+anonymousClass");
    map[loc, TypeSymbol] typeMap = (f:t | <f,t> <- model@types);
    InheritanceContext ctx = ctx();
    ctx@m3 = model;
    ctx@asts = asts;
    ctx@directInheritance = directInheritance;
    ctx@allInheritance = allInheritance;
    ctx@super = [];
    ctx@generic = [];
    ctx@typesWithObjectSubtype = {};
    ctx@declaringTypes = declaringTypes;
    ctx@invertedOverrides = invert(model@methodOverrides);
    ctx@typeMap = typeMap;
    println("done");
    getInheritanceTypes(projectLoc, ctx);
    ctx = visitCore(ctx);
    print("Saving output....");
    saveTypes(projectLoc, ctx);
    saveInternalReuse(projectLoc, ctx);
    saveExternalReuse(projectLoc, ctx);
    saveSubtype(projectLoc, ctx);
    saveDowncall(projectLoc, ctx);
    saveSuper(projectLoc, ctx);
    saveGeneric(projectLoc, ctx);
    println("done");
```

```
}

private InheritanceContext visitCore(InheritanceContext ctx) {
    //visit all methods, field initializers, constructors and type initializers
    list[Reuse] internalReuse = [];
    list[Reuse] externalReuse = [];
    list[Subtype] subtypes = [];
    list[Generic] generics = [];
    list[Super] supers = [];
    set[loc] typesWithObjectSubtype = {};
    list[Downcall] downcallCandidates = [];
    print("Analyzing project..");
    total = size(ctx@asts);
    n = 0;
    for(k <- ctx@asts) {
        if (n % 300 == 0) {
            print("<n * 100 / total>%..");
        }
        n = n + 1;
        Declaration ast = ctx@asts[k];
        loc returnType = tryGetReturnType(ast);
        loc methodDeclaringType = |unresolved:///|;
        if (hasDeclAnnotation(ast) && (ast@decl in ctx@declaringTypes)) {
            methodDeclaringType = ctx@declaringTypes[ast@decl];
        }
        else
        {
            //find field initializer, first occurrence of field
            //use its declaration to find the containing type

            top-down-break visit (ast) {
                  case Expression variable: \variable(str name, int extraDimensions): {
                        methodDeclaringType = ctx@declaringTypes[variable@decl];
                  }
                  case Expression variable: \variable(str name, int extraDimensions, Expression \initializer): {
                        methodDeclaringType = ctx@declaringTypes[variable@decl];
                  }
            }

        }
        visit (ast) {
            //case \arrayAccess(Expression array, Expression index):
            //internal reuse through array access is handled by the \simplename case;
            //external reuse through the qualifiedName case;

            //case \newArray(Type \type, list[Expression] dimensions, Expression init):
            //handled by other cases

            //case \newArray(Type \type, list[Expression] dimensions):
            //handled by other cases

            //case \arrayInitializer(list[Expression] elements):
            //handled by other cases
            case Statement foreach: \foreach(Declaration parameter, Expression collection, Statement body): {
                <stResult, objectSubtypes> = checkForeachForSubtype(ctx, parameter, collection);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
            }
            case Expression assignment: \assignment(Expression lhs, str operator, Expression rhs):  {
                <stResult, objectSubtypes> = checkAssignmentForSubtype(ctx, assignment, lhs, rhs);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
            }

            case Expression castExpression: \cast(Type \type, Expression expression): {
                //TODO: Generic attribute
                generics += checkCastForGeneric(ctx, \type, expression);
                //SUBTYPE: cast a child to a parent type
                <stResult, objectSubtypes> = checkDirectCastForSubtype(ctx, castExpression, \type, expression);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
```

```
             }

            //case \characterLiteral(str charValue):
            //not applicable

            case Expression ctor: \newObject(Expression expr, Type \type, list[Expression] args, Declaration class): {
                <stResult, objectSubtypes> = checkCallForSubtype(ctx, ctor@decl, args);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
            }
            case Expression ctor: \newObject(Expression expr, Type \type, list[Expression] args): {
                        <stResult, objectSubtypes> = checkCallForSubtype(ctx, ctor@decl, args);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
            }
            case Expression ctor: \newObject(Type \type, list[Expression] args, Declaration class): {
                    <stResult, objectSubtypes> = checkCallForSubtype(ctx, ctor@decl, args);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
            }
            case Expression ctor: \newObject(Type \type, list[Expression] args): {
                    <stResult, objectSubtypes> = checkCallForSubtype(ctx, ctor@decl, args);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
            }
            case \qualifiedName(Expression qualifier, Expression expression): {
                //Requires: accessed item's type, declaring type on accessed item
                //Declaring type on parent
                externalReuse += checkQualifiedNameForExternalReuse(ctx, qualifier, expression);

            }
            case Expression conditional: \conditional(Expression expression, Expression thenBranch, Expression elseBranch): {
                <stResult, objectSubtypes> = checkConditionalForSubtype(ctx, methodDeclaringType, conditional, thenBranch, elseBranch);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
            }
            case Expression fieldAccess: \fieldAccess(bool isSuper, Expression expr, str name): {
                //REMARK: isSuper only true when the Super keyword was used; so not relevant
                //INTERNAL REUSE: handles cases this.x and super.x
                //EXTERNAL REUSE: handles cases x.y;
                externalReuse += checkFieldAccessForExternalReuse(ctx, fieldAccess, expr);

                internalReuse += checkFieldAccessForInternalReuse(ctx, methodDeclaringType, fieldAccess, expr);
            }
            case Expression fieldAccess: \fieldAccess(bool isSuper, str name): {
                //REMARK: isSuper only true when the Super keyword was used; so not relevant for us
                //INTERNAL REUSE: handles cases this.x and super.x
                //EXTERNAL REUSE: not applicable
                internalReuse += checkFieldAccessForInternalReuse(ctx, methodDeclaringType, fieldAccess);
            }
            //case \instanceof(Expression leftSide, Type rightSide):
            case Expression methodCall: \methodCall(bool isSuper, str name, list[Expression] arguments): {
                //internal reuse
                //receiver is not present here; external reuse is not possible. E.g. super.X() or X();
                internalReuse += checkCallForInternalReuse(ctx, methodCall, methodDeclaringType);


                <stResult, objectSubtypes> = checkCallForSubtype(ctx, methodCall@decl, arguments);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
                //downcall candidate possible
                if (!isSuper) {
                loc decl = hasDeclAnnotation(ast) ? ast@decl : |type://unresolved/|;
                    downcallCandidates += checkCallForDowncall(ctx, methodCall, methodDeclaringType, decl);
                }
```

```

            }
            case Expression methodCall: \methodCall(bool isSuper, Expression receiver, str name, list[Expression] arguments): {
                internalReuse += checkCallForInternalReuse(ctx, methodCall, methodDeclaringType, receiver);
                externalReuse += checkCallForExternalReuse(ctx, methodDeclaringType, receiver, methodCall);
                <stResult, objectSubtypes> = checkCallForSubtype(ctx, methodCall@decl, arguments, receiver);
                typesWithObjectSubtype += objectSubtypes;
                subtypes += stResult;
                if (!isSuper) {
                    //if we are in a field initializer, we cannot provide the current method declaration. However; the field initializer
                    //cannot be overridden, so we don't care about the method declaration
                    //therefore we provide an unresolved location
                    loc astDeclaration = |unresolved:///|;
                    if (hasDeclAnnotation(ast)) {
                        astDeclaration = ast@decl;
                    }
                    downcallCandidates += checkCallForDowncall(ctx, methodCall, methodDeclaringType, astDeclaration, receiver);
                }
            }
            //case \null():
            //case \number(str numberValue):
            //case \booleanLiteral(bool boolValue):
            //case \stringLiteral(str stringValue):
            //case \type(Type \type):
            //case \variable(str name, int extraDimensions):
            case Expression variable: \variable(str name, int extraDimensions, Expression \initializer): {
                <stResult, objectSubtypes> = checkVariableInitializerForSubtype(ctx, variable, \initializer);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
            }
            //case \bracket(Expression expression):
            //case \this():
            //case \this(Expression thisExpression):
            //case \super():
            //case \declarationExpression(Declaration decl):
            //case \infix(Expression lhs, str operator, Expression rhs, list[Expression] extendedOperands):
            //case \postfix(Expression operand, str operator):
            //case \prefix(str operator, Expression operand):
            case Expression simpleName: \simpleName(str name): {
                //parent is a var access expr:
                //handles direct field access through a field name without this or super qualifier
                internalReuse += checkSimpleNameForInternalReuse(ctx, methodDeclaringType, simpleName);
            }
            //case \markerAnnotation(str typeName):
            //case \normalAnnotation(str typeName, list[Expression] memberValuePairs):
            //case \memberValuePair(str name, Expression \value):
            //case \singleMemberAnnotation(str typeName, Expression \value):
            // STATEMENTS
             case \return(Expression expression): {
                 //subtype might occur here
                <stResult, objectSubtypes> = checkReturnStatementForSubtype(ctx, returnType, expression);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
            }

            case Statement ctorCall: \constructorCall(bool isSuper, Expression expr, list[Expression] arguments): {
                if (isSuper) {
                    s = {t | t <- ctx@directInheritance[methodDeclaringType],t.scheme == "java+class" };
                    if (size(s) > 0) //nonsystem type
                     supers += super(methodDeclaringType, getOneFrom(s), ctorCall@src);
                }
```

```
                <stResult, objectSubtypes> = checkCallForSubtype(ctx, ctorCall@decl, arguments);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
            }
            case Statement ctorCall: \constructorCall(bool isSuper, list[Expression] arguments):{
                if (isSuper) {
                    s = {t | t <- ctx@directInheritance[methodDeclaringType],t.scheme == "java+class"};
                    if (size(s) > 0) //nonsystem type
                     supers += super(methodDeclaringType, getOneFrom(s), ctorCall@src);
                }

                <stResult, objectSubtypes> = checkCallForSubtype(ctx, ctorCall@decl, arguments);
                subtypes += stResult;
                typesWithObjectSubtype += objectSubtypes;
            }
        }
    }
    ctx@internalReuse = internalReuse;
    ctx@externalReuse = externalReuse;
    ctx@downcallCandidates = downcallCandidates;
    ctx@subtypes = subtypes;
    ctx@super = supers;
    ctx@generic = generics;
    ctx@typesWithObjectSubtype = typesWithObjectSubtype;
    println("100%..done");
    return ctx;
}
```

T-SQL – computing metrics from relationship attributes

This is a small sample of the SQL code used to analyse extracted facts.

```
CREATE view [dbo].[BaseMetrics] as
select a.ProjectId,
    a.FromType,
    a.ToType,
--nExplicitCC    Number of explicit userdefined cc edges
--- {(UserDefined) and (Explicit) and (CC)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CC'
    then 1.0 else 0.0 end as nExplicitCC,
--nCCUsed    Explicit class edges for which some subtype use or reuse use was seen
--- {(UserDefined) and (Explicit) and (CC) and (DirectExReuseField or IndirectExReuseField or DirectExReuseMethod
---or IndirectExReuseMethod or DirectSubtype or IndirectSubtype or UpcallField or UpcallMethod)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CC'
    and (ExternalReuse = 1 or Subtype = 1 or Upcall = 1)
    then 1.0 else 0.0 end as nCCUsed,
--nCCDC    Number of explicit CC edges that have Downcall use
--- {(UserDefined) and (Explicit) and (CC) and (Downcall)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CC' and Downcall = 1
    then 1.0 else 0.0 end as nCCDC,
--nCCSubtype    Used system CC edges for which subtype use was seen
--- {(UserDefined) and (Explicit) and (CC) and (DirectSubtype or IndirectSubtype)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CC' and Subtype = 1
    then 1.0 else 0.0 end as nCCSubtype,
--nCCSubtype    Used system CC edges for which subtype use was seen
--- {(UserDefined) and (Explicit) and (CC) and (DirectSubtype or IndirectSubtype)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CC' and (Subtype = 1 or Generic = 1 or Framework = 1)
    then 1.0 else 0.0 end as nCCSuspectedSubtype,
--nCCExreuseNoSubtype    Used system CC edges for which no subtype use was seen, but external reuse use was seen
--- {(UserDefined) and (Explicit) and (CC) and (DirectExReuseField or IndirectExReuseField or DirectExReuseMethod or IndirectExReuseMethod)
--- and (not DirectSubtype) and (not IndirectSubtype)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CC' and ExternalReuse = 1 and Subtype = 0
    then 1.0 else 0.0 end as nCCExreuseNoSubtype,
--nCCUsedOnlyInRe    Used system CC edges for which only internal reuse was seen
```

```
--- {(UserDefined) and (Explicit) and (CC)
--and (not DirectExReuseField) and (notIndirectExReuseField) and (not DirectExReuseMethod) and (not IndirectExReuseMethod) a
--nd (not DirectSubtype) and (not IndirectSubtype) and (UpcallField or UpcallMethod)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CC' and ExternalReuse = 0 and Subtype = 0 and Upcall = 1
    then 1.0 else 0.0 end as nCCUsedOnlyInRe,
--nCCUnexplSuper    Explict system edges that have no use or explanation but super constructor calls
--- {(UserDefined) and (Explicit) and (CC) and (not DirectExReuseField) and (not IndirectExReuseField) and (not DirectExReuseMethod)
--and (not IndirectExReuseMethod) and (not DirectSubtype) and (not IndirectSubtype) and (not UpcallField) and (not UpcallMethod)
--and (not Downcall) and (not ConstantsClass) and (not ConstantsInterface) and (not Marker) and (notFramework) and (not GenericUse)
--and (UpcallConstructorSuper)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CC' and ExternalReuse = 0 and Subtype = 0 and Upcall = 0
    and Downcall = 0 and Constants = 0 and Marker = 0 and Framework = 0 and Generic = 0 and UpcallConstructor = 1
    then 1.0 else 0.0 end as nCCUnexplSuper,
--nCCUnexplCategory    Explicit system edges that have no use or explanation (incl. super constructor calls) but has category use
--- {(UserDefined) and (Explicit) and (CC) and (not DirectExReuseField) and (not IndirectExReuseField) and (not DirectExReuseMethod)
--and (not IndirectExReuseMethod) and (not DirectSubtype) and (not IndirectSubtype) and (not UpcallField) and (not UpcallMethod)
--and (not Downcall) and (not ConstantsClass) and (not ConstantsInterface) and (notMarker) and (not Framework) and (not GenericUse)
--and (not UpcallConstructorSuper) and (Category)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CC' and ExternalReuse = 0 and Subtype = 0 and Upcall = 0
    and Downcall = 0 and Constants = 0 and Marker = 0 and Framework = 0 and Generic = 0 and UpcallConstructor = 0 and Category = 1
    then 1.0 else 0.0 end as nCCUnexplCategory,
--nCCUnknown    Explicit system class edges that no use or explanation is known (nCCUnused = nCCExplained+nCCUnknown)
--- {(UserDefined) and (Explicit) and (CC) and (not DirectExReuseField) and (not IndirectExReuseField) and (not DirectExReuseMethod)
---and (not IndirectExReuseMethod) and (not DirectSubtype) and (not IndirectSubtype) and (not UpcallField)
---and (not UpcallMethod) and (not Downcall) and (not ConstantsClass) and (not ConstantsInterface) and (not Marker)
---and (not Framework) and (not GenericUse) and (not UpcallConstructorSuper) and (not Category)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CC' and ExternalReuse = 0 and Subtype = 0 and Upcall = 0
    and Downcall = 0 and Constants = 0 and Marker = 0 and Framework = 0 and Generic = 0 and UpcallConstructor = 0 and Category = 0
    then 1.0 else 0.0 end as nCCUnknown,
--nExplicitCI    Explicit edges between user-defined classes and user-defined interfaces
--- {(UserDefined) and (Explicit) and (CI)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CI'
    then 1.0 else 0.0 end as nExplicitCI,
--nOnlyCISubtype    Edges between classes and interfaces for which subtype use was seen (the only use possible for such edges)
--- {(UserDefined) and (Explicit) and (CI) and (DirectSubtype or IndirectSubtype)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CI' and Subtype = 1
    then 1.0 else 0.0 end as nOnlyCISubtype,
--nExplainedCI    Edges from class to interface with no subtype use seen, but with one of Framework, Generic, etc (not Category)
--- {(UserDefined) and (Explicit) and (CI) and (not DirectSubtype) and (not IndirectSubtype)
---and (Framework or GenericUse or Marker or ConstantsInterface or ConstantsClass)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CI' and Subtype = 0
    and (Framework = 1 or Generic = 1 or Marker = 1 or Constants = 1)
    then 1.0 else 0.0 end as nExplainedCI,
--nCategoryExplCI    Edges for which no subtype use or other explanation was seen, but which have Category
--- {(UserDefined) and (Explicit) and (CI) and (notDirectSubtype) and (not IndirectSubtype) and (not Framework)
--- and (not GenericUse) and (not Marker) and (not ConstantsInterface) and (notConstantsClass) and (Category)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CI' and Subtype = 0
    and Framework = 0 and Generic = 0 and Marker = 0 and Constants = 0 and Category = 1
    then 1.0 else 0.0 end as nCategoryExplCI,
--nUnexplainedCI    Edges from class to interface with no subtype use seen or explained (including Category)
--- {(UserDefined) and (Explicit) and (CI) and (notDirectSubtype) and (not IndirectSubtype) and (not Framework)
```

```sql
--- and (not GenericUse) and (not Marker) and (not ConstantsInterface) and (notConstantsClass) and (not Category)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'CI' and Subtype = 0
    and Framework = 0 and Generic = 0 and Marker = 0 and Constants = 0 and Category = 0
    then 1.0 else 0.0 end as nUnexplainedCI,
--nExplicitII    Explicit edges between user-defined interfaces
--- {(UserDefined) and (Explicit) and (II)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'II'
    then 1.0 else 0.0 end as nExplicitII,
--nIISubtype    Edges between interfaces with at least subtype use
--- {(UserDefined) and (Explicit) and (II) and (DirectSubtype or IndirectSubtype)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'II' and Subtype = 1
    then 1.0 else 0.0 end as nIISubtype,
--nOnlyIIReuse    Edges between interfaces for which reuse was seen but not subtype
--- {(UserDefined) and (Explicit) and (II)
--- and (DirectExReuseField orIndirectExReuseField or DirectExReuseMethod or IndirectExReuseMethod)
--- and (not DirectSubtype) and (not IndirectSubtype)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'II' and Subtype = 0 and ExternalReuse = 1
    then 1.0 else 0.0 end as nOnlyIIReuse,
--nExplainedII    Unused edges between interface with some explanation (not category)
--- {(UserDefined) and (Explicit) and (II) and (not DirectExReuseField) and (notIndirectExReuseField)
--- and (not DirectExReuseMethod) and (not IndirectExReuseMethod) and (not DirectSubtype) and (not IndirectSubtype)
--  and (Framework or GenericUse or Marker or ConstantsInterface or ConstantsClass)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'II' and Subtype = 0 and ExternalReuse = 0
    and (Framework = 1 or Generic = 1 or Marker = 1 or Constants = 1)
    then 1.0 else 0.0 end as nExplainedII,
--nCategoryExplII    Edges for which no use or other explanation has been seen, but which have Category
--- {(UserDefined) and (Explicit) and (II) and (notDirectExReuseField) and (not IndirectExReuseField)
--- and (not DirectExReuseMethod) and (not IndirectExReuseMethod) and (not DirectSubtype) and (notIndirectSubtype)
--- and (not Framework) and (not GenericUse) and (not Marker) and (not ConstantsInterface) and (not ConstantsClass) and (Category)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'II' and Subtype = 0 and ExternalReuse = 0
    and Framework = 0 and Generic = 0 and Marker = 0 and Constants = 0 and Category = 1
    then 1.0 else 0.0 end as nCategoryExplII,
--nUnexplainedII    Edges between interfaces with no explanation (including Category)
--- {(UserDefined) and (Explicit) and (II) and (not DirectExReuseField) and (notIndirectExReuseField)
--- and (not DirectExReuseMethod) and (not IndirectExReuseMethod) and (not DirectSubtype) and (not IndirectSubtype)
--- and (notFramework) and (not GenericUse) and (not Marker) and (not ConstantsInterface) and (not ConstantsClass) and (not Category)}
    case when UserDefined = 1 and Explicit = 1 and RelationType = 'II' and ExternalReuse = 0 and Subtype = 0
    and Framework = 0 and Generic = 0 and Marker = 0 and Constants = 0 and Category = 0
    then 1.0 else 0.0 end as nUnexplainedII
from dbo.RelationAttributes a
```