

Dynamic Measurement of Polymorphism

Kelvin H.T. Choi, Ewan Tempero

Department of Computer Science
University of Auckland
Auckland, New Zealand

kelvincht@hotmail.com; e.tempero@cs.auckland.ac.nz

Abstract

Measuring "reuse" and "reusability" is difficult because there are so many different facets to these concepts. Before we can effectively measure reuse and reusability, we must first be able to effectively measure these different facets. One such facet is the programming language constructs that are available. For example whether or not a language supports polymorphism is believed to affect how reusable a developer can make a code artifact. Effectively measuring polymorphism is a challenge because its behaviour is only observable at run-time. In this paper, we present a metric for polymorphism based on the dynamic behaviour of the code. We evaluate the usefulness of the metric through two case studies.

Keywords: software metrics, polymorphism, inheritance, dynamic profiling

1 Introduction

Measurement is considered important for many human endeavours. It is also often difficult to establish a form of measurement that allows management, especially prediction, to be done with confidence. Often we measure an attribute of interest by making a number of different measurements and combining them in a way that we believe (or hope) is representative of the original attribute. An example is "health". We have no way to measure health directly, and so we must make a number of measurements of other attributes, such as weight, blood pressure, cholesterol levels, and so on. So in order to measure health, we first have to be able to measure these attributes. So it is often the case that in order to measure what we want, we must identify all the relevant aspects and determine how to measure those first.

Managers of software engineers would like to be able to measure an engineer's productivity, but, as with health, there are many aspects that can affect productivity. Even measuring the individual aspects, were they clearly identified, is not easy. One such aspect is *software reuse*. Proponents of software reuse make many claims about how it affects productivity, but measuring that effect, and indeed even measuring how much reuse takes place, has proven difficult.

Copyright (c) 2007, Australian Computer Society, Inc. This paper appeared at the Thirtieth Australasian Computer Science Conference (ACSC2007), Ballarat, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 62. Gillian Dobbie, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

One confounding factor in measuring the benefits of reuse is that reusing different artifacts requires different amounts of effort. This means that we have the notion of *reusability*, the ease with which something can be reused, that must be taken into account when measuring reuse.

Polymorphism has been claimed to improve reusability, and so to fully measure productivity we may need some effective means to measure polymorphism. We also need to measure polymorphism in order to establish the veracity of the claims made about it with respect to reusability. It is the measurement of polymorphism that is the subject of this paper.

The problem with measuring polymorphism is that it is a concept whose behaviour is only seen at run-time. We can reason about what might be possible statically, but it is only dynamically that we can really see what is going on. In this paper, we present our attempt to measure polymorphism by means of software profiling. This is a technique that has been traditionally used for understanding other forms of dynamic behaviour, such as detecting "hot spots" for improving performance or determining the coverage of test cases. We apply similar techniques in an attempt to measure the polymorphism that takes place during the execution of an application.

The rest of the paper is organised as follows. The next section discusses the background and related work. Section 3 introduces the metric we propose for measuring polymorphism. Section 4 presents the methodology we used in our study of the metric and section 5 presents the case studies we carried out. We then discuss the results of our case studies and finally present our conclusions.

2 Background

2.1 Reuse, Reusability, and Polymorphism

The idea of developing software by re-using existing software has been around since the dawn of software engineering as a discipline (McClure 1997). Since that time, much research has been done to turn this idea into reality, of which we can only touch on here (see surveys such as (Krueger 1992; Mili et al. 1995) for more detail). For the most part this research has focused on planned or *systematic reuse*, that is, how organisations can, by using explicit processes and standards, get the most benefit from reuse. Early efforts in this regard examined the design, development, and organisational use of repositories of reusable assets (see (McClure 1997) for example). Later efforts considered domain engineering and domain analysis as promising avenues (see (Tracz et

al. 1993) for example), which led to software product lines (Clements and Northrop 2001).

Many authors have made a link between the reusability of code and the use of object-oriented concepts, for example (Meyer 1997). The common example is that of *inheritance*, and how it saves effort by supporting the easy use of an existing artifact without modification, that is, it aids **reuse**. Polymorphism is often lumped with inheritance, and so is accorded similar benefits with respect to reuse by implication. But in fact polymorphism has a more significant role for reuse than that. Consider the Java example below:

```
public interface XMLFormatter {
    public String asXML();
}
public void format(List<XMLFormatter> list) {
    for (XMLFormatter formatter: list) {
        System.out.println(formatter.asXML());
    }
}
```

The polymorphism that takes place here is in the call to the `asXML()` method on the `formatter` variable, since the contents of `list` could be any class that implements the `XMLFormatter` interface. Because polymorphism is being supported, the `format` method not only will work for any existing class that implements the `XMLFormatter` interface, *it will also work for any future such class*. It is instructive to consider what is being “reused” here. If it is the `XMLFormatter` interface, then not much (1 line) effort is being saved. Nor is it implementations of the interface, since they may have nothing in common. In fact, what is being reused here is the `format` method, since anyone needing this functionality can just use this method. If we didn’t have polymorphism, this would not be possible, so what polymorphism makes it easier for us to write code that can be used in multiple situations, that is, it supports **reusability**.

2.2 Measuring Reuse and Reusability

There have been a number of efforts in measuring reuse. Most of the research into measurement relating to reuse has focussed on the economics of reuse. Software reuse is only beneficial if the return on investment (ROI) (Kain 1994) is positive. There are costs in managing the reuse process. This includes the cost of maintaining the reuse libraries, the cost to modify and repackage reusable artifacts from the existing asset, and the cost of searching, identifying, evaluating, selecting and integrating the potential artifact.

Various economics metrics have been proposed to measure the cost-benefit ratio of reuse. Gaffney and Durek’s reuse economic metric (Gaffney and Durek 1989) is useful to measure the cost of reuse and its break-even point. Reuse is economically break-even when the cost of the reuse equals the reuse benefit. Balda and Gustafson’s COCOMO-based reuse model (Balda and Gustafson 1990) can estimate the total time it would take to build software with reuse. Barnes and Bollinger’s cost-benefit analysis (Barnes and Bollinger 1991), on the other hand is focused on the quality benefits.

Henderson-Sellers’s cost-benefit analysis (Henderson-Sellers 1993) is focused on calculating the ROI from reuse activities. Malan’s cost-benefit analysis with NPV (Malan and Wentzel 1993) can be used to determine the financial cost saving from reuse and their fixed costs. Poulin and Caruso’s reuse metrics and ROI (Poulin and Caruso 1993) can be used to calculate and differentiate Project-level ROI and Corporate-level ROI. Finally the U.S. Defense information system agency (DISA)’s reuse metrics and ROI (DISA/JIEO/CIM 1993) is focused on measuring cost avoidance.

The Gaffney and Durek’s reuse economic metric is representative of many of these. It is

$$C = (b + E/n - 1)R + 1$$

Where

C = relative cost of software development (generally C will be less than 1, less is better)

R = the proportion of reused code in the project ($0 \leq R \leq 1$)

b = the cost, relative to new code, of reusing existing code in this project. (For example searching, adaptation and integration cost)

E = the cost, relative to new code, of developing a component for reuse. (For example the cost to make code reusable)

n = the number of expected reuses

This metric requires the cost of creating the reused artifacts (E) as input, whereas our interest is ultimately in being able to measure this cost.

There are different interpretations about what to measure as reuse. Poulin and Kain (Kain 1994; Poulin 1997) only count external reuse (reuse across project boundaries) as reuse. For example, they do not count copy and paste, or programming languages features such as sub-classing and polymorphism, as reuse. Also they do not count modified component and Commercial Off-the-Shelf Software (COTS) as reuse. They do not give additional credit for how many times a component is reused. On the other hand, both Henderson-Sellers (Henderson-Sellers 1993) and Frakes (Frakes and Carol 1994) have another perspective about reuse. They view sub-classing via inheritance as reuse, count internal reuse as reuse, and give credit for how many times components are reused. Poulin and Kain emphasise ‘industrial benefits’ of reuse, whereas Henderson-Sellers and Frakes emphasise the academic definition of reuse.

Two commonly cited reuse metrics are those by Banker et al (Banker et al. 1994) and Frakes (Frakes and Carol 1994).

Reuse Percentage

Banker et al. developed a metric in a repository-based CASE environment named High Productivity Systems (HPS). Their metric measures the level of reuse using reuse percentage:

$$\text{Reuse Percentage} = (1 - \text{New objects built} / \text{Total objects used}) \cdot 100\%$$

For example suppose a project uses 400 objects, however the developers in this project only build 100 objects. In this case the reuse percentage would be 75%. This metric counts multiple invocations of the same object as multiple instances of reuse. For example 200 uses may come from using 50 objects 4 times each. Also this metric measures both internal and external reuse within the project. For example from the 400 uses, 200 uses may come from internal reuse and the other 200 may come from external reuse.

This metric does not precisely define object granularity. This does not obey the fourth property (Devanbu et al. 1996) discussed by Devanbu et al., which states that reuse benefit measures should be sensitive to the cost of the object being reused. In the above metric, reusing a small object may yield the same reuse benefit as a large object.

Reuse Level Metric

Frakes (Frakes and Carol 1994) introduced the Reuse Level Metric. This metric uses threshold levels to filter out the items that are not reused often enough. For example if the threshold level is 3, an item would have to be called at least 4 times in order to be counted as reuse. Also this metric differentiates between internal reuse level and external reuse level.

Internal reuse level = IU / T

External reuse level = EU / T

Total reuse level = Internal reuse level + External reuse level

Where

IU = number of external items reused, that is more than the external threshold level.

EU = number of internal items reused, that is more than the internal threshold level.

T = the total numbers of items in the system, both external and internal

Each reused item (such as IU and EU) only has a value 0 or 1. If the item is reused more than the threshold level, it will always be 1, otherwise it will be 0. This means that this metric does not take into account how many times an item is used.

Note that the Reuse Level Metric uses items instead of lines of code (LOC), since each item would be different and some items would be very large and some items would be very small. Another version of this metric assigns a weight to each item depending on the item's size. It also allows different threshold levels for internal and external reuse. This means that in order to compare results between projects, their threshold levels must be same. Otherwise, projects with smaller threshold values will appear better.

3 Measuring Polymorphism

The problem with measuring polymorphism is that exactly what happens can vary from execution to execution of an application. In the earlier example, the implementation of the `format` method that is called can

potentially change every time through the loop. In order to better understand what benefit we are getting through the use of polymorphism, we would like to characterise what actually happens.

3.1 Polymorphic behaviour index

Consider the following example, which provides a simpler view of polymorphism in action than our earlier example:

```
List list; //general type declaration
If (external condition)
    list = new ArrayList();
else
    list = new LinkedList();
list.add(...);
```

In the example, whether the call to the `add` method on the `list` variable causes the `ArrayList` implementation or the `LinkedList` implementation to be called depends on the external condition. For any given condition, we can not always tell whether or not that condition is ever satisfied, so while it looks like polymorphism will take place in our example, if the condition is never true (or false) then in fact no polymorphism will happen. In this particular example, because the declared type of `list` is an interface type, any method invoked on it will involve a polymorphic call, but that will not always be the case.

We can use dynamic analysis to get better information. At runtime, when the `add` method is called, the actual class of the `list` variable is known, and so the actual implementation being used can be determined. Whether or not polymorphism has actually occurred depends on whether the type of the object on which the method call is dispatched agrees with the declared type of the variable.

This gives the basis for our metric. We can determine whether or not a method call is polymorphic as described above. This gives us a simple metric that relates to the amount of polymorphism that takes place:

Polymorphic behaviour index = $P / \text{Total dispatches}$

Where

Total dispatches = $(P + NP)$

P = Unique polymorphic dispatches executed

NP = Unique non-polymorphic dispatches executed

The **Declared Interface** is the interface of the variable that is declared in the source code. In the example: `List list = new ArrayList();` the `list` variable will be declared using the `List` interface.

The **Dispatched Class** is the class that the method is invoked on. In the example, if the external condition is true, then the dispatched class will be `ArrayList`.

Conforms and *implements* are relations between a Class and an Interface, and they are different. Any class that implements an interface must also conform to that interface. However a class that conforms to an interface might not directly implement it. The implementation of an interface method may come from its parent class. The

method called from an interface is dispatched to the implementation of that interface.

When an object's method is called, the actual method that is executed is the deepest inherited method that has an implementation. Consider this example:

```
Class Parent{
    void method3 (){};
}
class Child extends Parent{}
Child c = new Child();
c.method3();
```

The `c.method3()` will be dispatched to `parent.method3()` because it is the deepest inherited method that has an implementation.

Polymorphic dispatch is detected when the Declared Interface and the Dispatched Class is different.

Non-polymorphic dispatch is detected when the Declared Interface and the Dispatched Class is the same.

For example:

```
interface I{
    m();
}
class Parent implements I{
    m(){ //a implementation of method m()
        ...; //do something
    }
    m2(){...; //do something }
}
class Child extends Parent{
    m2(){
        super();
        ...; //do something more specific
    }
}
```

When:

```
Parent obj1 = new Parent();
obj1.m();
```

The `obj1` variable points to a `Parent` Object. The `Parent` class has a method `m()` implementation so `Parent.m()` is dispatched. **The Dispatched Class of method `m()` in this case is `Parent`.** This dispatch is **non-polymorphic** because the Declared Interface is the same as the Dispatched Class; `Parent` in this case.

```
Child obj2 = new Child();
obj2.m();
```

Although `obj2` points to an Object of type `Child`, the method that is dispatched is `Parent.m()`. Since `Child` does not have a method `m()` implementation, the *deepest* inherited method that has such implementation is `Parent.m()`. **The Dispatched Class in this case is `Parent`.** This dispatch is **polymorphic** because the Declared Interface is `Child`, whereas the Dispatched Class is `Parent`.

```
I obj3 = new Parent();
obj3.m();
```

In this case, although `Parent.m()` will be dispatched. However the Declared Interface is `I` and the Dispatched

Class is `Parent`. This dispatch is **polymorphic** because Declared Interface = `I`, Dispatched Class = `Parent`.

Inherited Method Call using inherited class

```
Parent obj4 = new Child();
obj4.m();
```

In this case, the dispatched method is `Parent.m()` and the Declared Interface of `obj4` is `Parent`. So the dispatch is **Non-polymorphic** because Declared Interface = `Parent`, Dispatched Class = `Parent`.

```
Parent obj5 = new Child();
obj5.m2();
```

In this case `obj5` uses the inherited method `m2()` from `Child`. `Child.m2()` overrides the `Parent's m2()` and this inheritance reuses the effort from the parent's `m2()` method. The dispatched method is `Child.m2()` because it is the deepest method that has an implementation. So the dispatch is **polymorphic** because Declared Interface = `Parent`, Dispatched Class = `Child`.

Consider the following example with loop:

```
static void m1(ArrayList arrayList){
    for (int i=0;i<10;i++){
        arrayList.add();//this is non-poly..
        //Declared Interface = ArrayList,
        //Dispatched Class = ArrayList
    }
}
static void m2(List list){
    for (int i=0;i<10;i++){
        list.add();//this is polymorphic
        //Declared Interface = List,
        //Dispatched Class= ArrayList
    }
}
```

In the above example, `m2()` supports polymorphism and it is more reusable than `m1()`, because the input parameter of `m2()` uses an interface `List` instead of a strict `ArrayList`. This means that other people using `m2()` can use all kinds of `List` such as `LinkedList`, `Vector` and `ArrayList`. However people using `m1()` can only use `ArrayList`.

Now consider this case

```
//it is forced to use ArrayList
m1(new ArrayList());
```

When `m1()` is called, the code inside `m1()` will be executed. Within the `m1()` loop, `arrayList.add()` will be called 10 times.

The Declared Interface of `arrayList` is `ArrayList`. Since `arrayList` points to an object of `ArrayList`, the dispatched method is `ArrayList.add()`. In this case the declared interface and dispatched class are same so this will be in non-polymorphic. Also although there are 10 calls, they all occur at one call site due to a single call to `m1()`, so they will be counted as one *unique* dispatch.

Since there are no polymorphic dispatches but there is one non-polymorphic dispatch, in this case:

Polymorphic behaviour index = $0 / (0+1) = 0\%$

Consider another case:

```
m1(new ArrayList()); //non-polymorphic
m2(new LinkedList()); //polymorphic
m2(new Vector()); //polymorphic
m2(new ArrayList()); //polymorphic
```

In this example, `m2()` is called three times and `m1()` is called once. `m2()` is polymorphic and it takes all types of lists. The `ArrayList.add()` method within the `m1()` loop will be executed 10 times and there will be 1 unique non-polymorphic dispatch. Also `list.add()` method within the `m2()` loop will be executed. The `list.add()` method within `m2()` has a Declared Interface of `List`.

- The `list.add()` will be executed with `LinkedList` 10 times. Dispatched Class = `LinkedList.add()`. The method calls will be count as one unique polymorphic dispatch.
- The `list.add()` will be executed with `Vector` 10 times. Dispatched Class = `Vector.add()`. The method calls will be count as one unique polymorphic dispatch.
- The `list.add()` will be executed with `ArrayList` 10 times. Dispatched Class = `ArrayList.add()`. The method calls will be count as one unique polymorphic dispatch.

The total number of unique polymorphic dispatches is 3 and the total number of unique non-polymorphic dispatches is 1.

Since there are 3 polymorphic dispatches and one non-polymorphic dispatch, in this case:

Polymorphic behaviour index = $3 / (3+1) = 75\%$

3.2 External and internal reuse

Software reuse can be classified as internal reuse or external reuse. External reuse occurs when calls are made to code supplied from a source external to the project, what is often referred to as an “API”, whereas internal reuse is when calls are made to code that has been already written for the application. Some people believe external reuse is most beneficial and so internal reuse should not be counted. However internal reuse is also important. For example developers reusing their own methods are more productive than creating another new method that does the same thing. For this reason, we measure both internal and external reuse.

Internal methods are defined as methods that are created by the application developers. They also include methods implementing the API interface or custom code extending the API.

External methods are defined as methods that are supplied from external source and which developers cannot change. These include API and external libraries.

An *external method call* happens when an *internal method* calls an *external method*. This excludes an *external method* calling another *external method*. The

reason for this is because the benefit of reusing the external method directly called by an internal method is equivalent to reusing all of the methods that that external method reuses. Therefore it is not required to re-count the inter-external method reuse.

An *internal method call* happens when any *method* calls another *internal method*. In most cases it will be internal methods calling internal methods. However sometimes internal methods are used as handlers and are passed into the API and the API will call them. In that case they are still counted as internal method calls because such external to internal method calls is a consequence of the developer’s decision just as with other internal to internal method calls.

Consider the following example:

```
class Example extends java.lang.Thread{
void methodA(){
    methodB();
}
void methodB(String s){
    java.lang.System.out.println("abc");
}
/*this method implements the run() method required in
thread. When Thread.start() is called, this method
will be called by the JVM scheduler.*/
Public void run(){
    methodA();
}
public static void main(...){
    this.start(); //this class extends Thread.
    // This will call Thread.start()
}
}
```

When this `Example` class is executed, the `main()` method will be called by the *Java Launcher*. Since `Example` extends `Thread`, calling the `start()` method will put this `Thread` into the *Java Scheduler*. Eventually the *Java Scheduler* will call the `run()` method and the `run()` method will call `methodA()`, which will, in turn, call `methodB()`. `methodB()` will then call `System.out.println()`.

There are many types of method calls in this example:

The internal method calls consist of:

- `methodA()` calling `methodB()`, both are *internal methods*
- *Java Scheduler* calling `run()`, notice the *Java Scheduler* is an API and it is not in the code. It is *external to internal* method call and it is counted as internal method calls.
- `run()` calling `methodA()`, `run()` conforms to the `Thread.run()` interface, but it is an *internal method* because it is created by the application developer. `methodA()` is an *internal method*. It is *internal to internal* method call.
- *Java Launcher* calling `main()`, notice the *Java Launcher* is an API and it is not in the code. It is *external to internal* method call.

The external method calls consist of:

- `methodB()` calling `System.out.println()`, internal to external method calls
- `main()` calling `start()`, `main()` is an internal

method implemented by the developer. The `start()` method from `Thread`. Although `Example.start()` method is called and `Example` is an *internal* class, `Example` does not have an implementation of `start()`. It inherits the `Thread.start()` implementation. So the `Example.start()` method is an *external* method. `main()` is *internal* and `start()` is *external* so it is *internal to external* method call.

There are many *external to external* method calls, for example when `System.out.println()` is called, many inter-API calls will be made such as the `StringBuffer` calling `PrintWriter`, etc. However we will not classify them as either internal or external method calls.

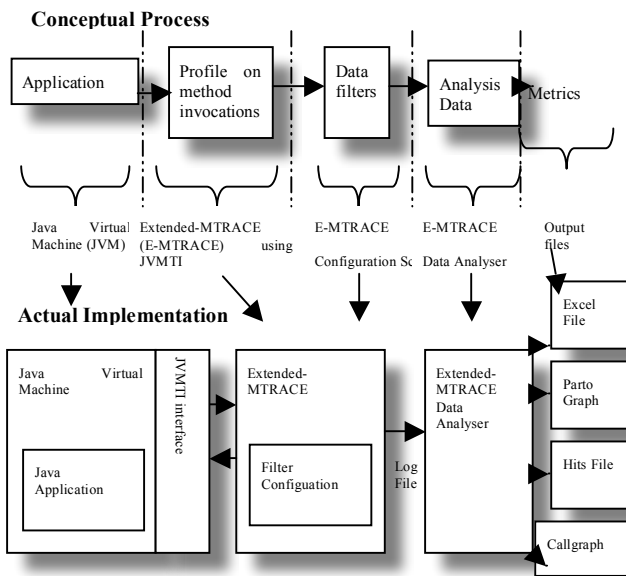


Figure 1 the Process for Dynamic Metrics.

4 Methodology

Our methodology for evaluating our metric is to compare two software applications from the same domain both in terms of our metric, and by manually examining the source code to determine their relative reuse and reusability performance.

Computing the metric requires instrumenting the compiled version of each application. Each application is then executed via a script that provides the same workload to each to produce a profile of their execution.

This will result in runtime data such as: method calls, call sites, Dispatched Class and so on. A filter is needed to remove the unnecessary results such as system method calls made by the JVM. Then the filtered data is analysed to identify polymorphic and non-polymorphic dispatches, from which the metric value is computed.

The tool we have developed to do all this is called E-MTRACE (Choi 2006). E-MTRACE uses the JVM Tool Interface (JVMTI), which provides a means to inspect and control the execution of programs running in the Java Virtual Machine. E-MTRACE extends the JVMTI demo MTRACE. E-MTRACE first inserts bytecodes into methods by using a File Hook. A File Hook interrupts the JVM when the JVM wants to load the java .class file into the Heap. The Hook interprets the original .class file and inserts profiling instrumentation code before any method that is called. This allows instrumentation code to be inserted during the runtime. When a method is called, the instrumentation code is executed. The instrumentation code uses the program stack to identify the Dispatched Class. It then uses the Local Variable Table to trace the Declared Interface. The details are discussed in the Choi's Thesis (Choi 2006). Once the Declared Interfaces and Dispatched Classes are identified, they will be processed using the E-MTRACE Analyser tool, which produces the list of polymorphic dispatches and non-polymorphic dispatches. An example is shown in figure 2.

5 Case Study

We performed a case study to evaluate the polymorphic behaviour index metric. The case study consisted of comparing two Search Engines – Egothor (Galambos 2006) and Lucene (Bialecki et al. 2005) and two FTP servers – Dano (Paregos 2002) and Jupiter (Filion 2002) using the polymorphic behaviour index.

5.1 Search Engines

The two search engines are Egothor (Galambos 2006) and Lucene (Bialecki et al. 2005). Egothor is a fully featured search package and it can be configured as a full-text search engine, meta searcher and crawler robot. It uses a pipe-line architecture. It also uses an extendable filter architecture. The search algorithm can automatically find the cheapest search-plan (the order of search-operations executed) using weighting. The plan can be automatically compiled and executed. Search operations are dispatched into separate thread allowing it to take advantage of a multiple CPU machine.

	A	B	C	D	E	F	G	H
449	DeclaredType	DispatchedClass	ToMethod	FromMethod	Variable			
450	java/io/DataInput	java/io/DataInputStream	java/io/DataInputStream.readU	org/egothor/store/util/	org/egothor/store/Util/Compress.readFac			
451	java/io/DataInput	java/io/DataInputStream	java/io/DataInputStream.readU	org/egothor/store/util/	org/egothor/store/Util/Compress.readFac			
452	java/io/DataInput	java/io/RandomAccessFile	java/io/RandomAccessFile.read	org/egothor/store/util/	org/egothor/store/Util/Compress.readFac			
453	java/io/DataInput	java/io/RandomAccessFile	java/io/RandomAccessFile.read	org/egothor/store/util/	org/egothor/store/Util/Compress.readFac			
454	java/io/DataInputStream	java/io/FilterInputStream	java/io/FilterInputStream.close	org/egothor/db/disc/D	org/egothor/db/disc/DiscIndexData.getM			
455	java/io/InputStream	java/io/BufferedInputStream	java/io/BufferedInputStream.cl	org/egothor/store/disc/	org/egothor/store/disc/ThickBarrel.setLoc			
456	java/io/InputStream	java/io/FileInputStream	java/io/FileInputStream.close	org/egothor/dir/Tanke	org/egothor/dir/TankerImpl.loadStateO.i			
457	java/lang/Object	java/lang/String	java/lang/String.toString	org/egothor/db/disc/D	org/egothor/db/disc/DiscKeyIndexData.i			
458	java/lang/Object	java/lang/String	java/lang/String.charAt	org/egothor/db/disc/D	org/egothor/db/disc/DiscKeyIndexData.v			
459	java/lang/Object	java/lang/String	java/lang/String.length	org/egothor/db/disc/D	org/egothor/db/disc/DiscKeyIndexData.v			
460	java/lang/Object	java/lang/String	java/lang/String.toString	org/egothor/db/disc/D	org/egothor/db/disc/DiscKeyIndexData.v			
461	java/util/Enumeration	java/util/Vector.\$l	java/util/Vector.\$l.hasMoreEle	org/egothor/query/QC	org/egothor/query/QGroup.addTermsO.e			
462	java/util/Enumeration	java/util/Vector.\$l	java/util/Vector.\$l.nextElement	org/egothor/query/QC	org/egothor/query/QGroup.addTermsO.e			
463	java/util/Enumeration	java/util/Vector.\$l	java/util/Vector.\$l.hasMoreEle	org/egothor/query/QC	org/egothor/query/QGroup.applyCWI0.e			
464	java/util/Enumeration	java/util/Vector.\$l	java/util/Vector.\$l.nextElement	org/egothor/query/QC	org/egothor/query/QGroup.applyCWI0.e			
465	java/util/Enumeration	java/util/Vector.\$l	java/util/Vector.\$l.hasMoreEle	org/egothor/query/QC	org/egothor/query/QGroup.attachO.e			
466	java/util/Enumeration	java/util/Vector.\$l	java/util/Vector.\$l.nextElement	org/egothor/query/QC	org/egothor/query/QGroup.attachO.e			
467	java/util/Enumeration	java/util/Vector.\$l	java/util/Vector.\$l.hasMoreEle	org/egothor/query/QC	org/egothor/query/QGroup.explainO.en			
468	java/util/Enumeration	java/util/Vector.\$l	java/util/Vector.\$l.nextElement	org/egothor/query/QC	org/egothor/query/QGroup.explainO.en			
469	java/util/Enumeration	java/util/Vector.\$l	java/util/Vector.\$l.hasMoreEle	org/egothor/query/QC	org/egothor/query/QGroup.setModelO.e			
470	java/util/Enumeration	java/util/Vector.\$l	java/util/Vector.\$l.hasMoreEle	org/egothor/query/QC	org/egothor/query/QGroup.toStringO.e			
471	java/util/Enumeration	java/util/Vector.\$l	java/util/Vector.\$l.nextElement	org/egothor/query/QC	org/egothor/query/QGroup.toStringO.e			
472	java/util/Iterator	java/util/HashMap.\$HashIterator	java/util/HashMap.\$HashIterate	org/egothor/dir/Group	org/egothor/dir/Group.\$RetrieveCWIActi			
473	java/util/Iterator	java/util/HashMap.\$HashIterator	java/util/HashMap.\$HashIterate	org/egothor/query/Ex	org/egothor/query/Executor.queryO.i			
474	java/util/Iterator	java/util/HashMap.\$KeyIterator	java/util/HashMap.\$KeyIterator	org/egothor/dir/Group	org/egothor/dir/Group.\$RetrieveCWIActi			

Figure 2 Polymorphic dispatches output data

Lucene provides search engine functionality through a simple API. It is intended to be a scalable, fast, cross platform system providing full-text search. The design uses recursion method calls and many handlers.

Egothor provides more features than Lucene, but both provide full-text searching, so that functionality is profiled and compared. Egothor and Lucene use different design architectures to implement their core engine and so it is a useful comparison. In order to compare the two search engines, their mutual inclusive functionalities are used and these include:

- Index search
- Create index
- Search syntax operators such as AND OR NOT
- Dynamically update index

Searcher	Reuse type	Polymorphic behaviour index (more the better)	Polymorphic dispatches	Non polymorphic dispatches
Egothor	External	64%	45	25
Egothor	Internal	40%	46	70
Lucene	External	20%	5	19
Lucene	Internal	29%	52	127

Table 1 Searchers Polymorphic Dispatches

The results are shown in Table 1. Egothor (both externally and internally) has a higher polymorphic behaviour index than Lucene. Also Lucene has significantly more internal dispatches than external dispatches.

When inspecting the Lucene code and design, the internal design of Lucene confirms the internal reuse and also significant use of inheritance. Therefore it has a lot of polymorphic dispatches – 52. For example there are many polymorphic types of keywords (Terms) and Keyword-Indexes.

However the inheritances trees are disjointed and the trees are shallow. For example there are many types of Index, there are many types of Terms but Term and Index do not inherit from each other. That is why it has many polymorphic dispatches – 52 but there is also a lot of non-polymorphic dispatches -127. It only has 29% of internal polymorphic behaviour index.

Externally, Lucene does not have many dispatches (either polymorphic or non-polymorphic) at all. Egothor on other hand is very good on dispatching external API and it even inherit on the external API (64% external polymorphic). For example its Cache extends HashMap, and its in-house developed QueryHit extends the standard Java Iterator.

The Egothor internal polymorphic behaviour usually comes from different Query Types extending the Query Interface. For example QueryTerm and QueryGroup extends Query. Also Egothor’s internal design is a procedural, linear pipeline and uses filters. For example it is structured in steps:

- Step 1: Find cheapest search-plan
- Step 2: compile search-plan
- Step 3: execute search-plan.

Each step makes many calls to the external API and reuses the internal plan elements such as Nodes and the Edges (filters). Also filters are polymorphic and they can be extended. For example BuildHTML extends Edge. BuildHTML takes the input of data tokens from previous Node and convert it into HTML documents.

However there are places where inheritance could have been used, but was not. For example there are three BuildHTML classes. They are BuildHTML1, BuildHTML2 and BuildHTML3. All of them are similar because they build different format of the HTML document. Unfortunately all of them only extend edge. It is not enough and it can be done better, for example:

Egothor way

```
BuildHTML1, BuildHTML2, BuildHTML3 extends Edge
```

Better way

```
AbstractBuildHTML extends Edge
BuildHTML1, BuildHTML2, BuildHTML3 extends
AbstractBuildHTML
```

Assuming the commonality between the BuildHTML classes exist, the common functions of the BuildHTML classes should be extracted and put into the AbstractBuildHTML, so when a new BuildHTML4 is created less effort is needed because it can reuse the code in the AbstractBuildHTML. If such a design were used, then we would probably see a higher polymorphic behaviour index. This could be one reason why the internal polymorphic behaviour of the Egothor (40%) is less than the external polymorphic behaviour (64%).

Overall, Egothor has a higher percentage of polymorphic behaviour than Lucene.

5.2 Ftp Servers

The two FTP servers are Dano (Paregos 2002) and Jupiter (Filion 2002). The implementation is quite different between the two. Jupiter is based on a simple design that uses as few classes as possible. Dano is based on a highly extendable design that has many little classes and each of them is responsible for different functions. In order to compare two FTP servers, their mutual inclusive functionalities are used and these include:

- Browsing a folder
- Upload a file
- Download a file
- Move/Copy/Remove file

FTP Server	Reuse type	Polymorphic behaviour index (more the better)	Polymorphic dispatches	Non polymorphic dispatches
Dano	External	24%	7	22
Dano	Internal	63%	59	34
Jupiter	External	12%	7	52
Jupiter	Internal	0%	0	22

Table 2 FTP Servers polymorphic dispatches

The results are shown in table 2. It shows that Dano has significantly higher external and internal (24%, 63%) polymorphic behaviour index than Jupiter (12%, 0%). It is interesting that Jupiter internally has zero polymorphic dispatches.

Dano has higher internal polymorphic behaviour index (63%) than externally (24%), whereas Jupiter has lower internal polymorphic behaviour index (0%) than external polymorphic index (12%).

Comparing external polymorphic behaviour

Dano FTP extends its FTPConnection on a Java Thread object. Jupiter on the other-hand does the same thing, but with three classes: ConnectionThread, FileTransferThread and PasvListeningThread. It appears Jupiter is more multi-threaded than Dano.

Dano indirectly made some external polymorphic dispatches by using the polymorphic Factory API, such as:

```
InetAddress ia = InetAddress.getLocalHost();
...//Call ia.anyMethod()
```

The variable ia is declared as InetAddress, but using the InetAddress.getHost() method returns an Inet4Address object. Therefore variable ia is declared as InetAddress but the actual class is Inet4Address. It means using a polymorphic factory API will improve the polymorphic behaviour index.

Both of the FTP server applications scored the same number of internal polymorphic dispatches (7), however Jupiter has more non-polymorphic external dispatches (52) than Dano (22). Therefore Jupiter has proportionally fewer polymorphic dispatches (24%) than Dano (12%).

Comparing internal polymorphic behaviour

Internally it is a different story. Dano designs for polymorphism while Jupiter does nothing at all. Dano has 63% internal polymorphic behaviour index but Jupiter has 0%. This can be explained by the architecture:

Jupiter only has a small number of Java files. However each of them is very big and does many things. For example, Jupiter puts all the FTP-commands handling logic into one class called ProtocolInterpreter. It is a huge class and it is very difficult to understand. Also many things are hard-coded.

If the FTP protocol is changed, for example command "SYST" is renamed "SYSE", the whole of ProtocolInterpreter needs to be inspected and updated.

On the other-hand, Dano uses inheritance a lot internally. For example each different FTP command has its own class and all of them extend FTPCommand.

```
public interface FTPCommand {
    //Sets the FTPUserContext before the //command
    executes.
    Public void setUserContext( FTPUserContext uc );
    //Sets the FTPCommand argument string.
    public void setArgument( String arg );
    //Executes the command.
    public void runCommand( ) throws
        FTPException;
    //Aborts the command.
    public void abortCommand( );
}
```

Inside Dano, there are 28 Command classes that extend the FTPCommand interface. For example, within SYST:

```
public void runCommand( ) {
    pi.sendResponse( pi.SYSTEM_NAME,
        System.getProperty("os.name") );
}
```

As illustrated, the Dano's command code is very clean compared to Jupiter. There are many advantages of this design. For example, if there is a new SYST2 command based on SYST, the new command can inherit from it:

```
Class SYST2 extends SYST{
    Public void runCommand(){
        Super();
        //New code here..
    }
}
```

Another advantage of this design is the commands are separated into different classes. It makes the parts easier to be reused by other applications. For example if a 'stripped-down' version of a FTP server is needed, it only needs to delete the un-needed classes. Comparing this to Jupiter, implementing a stripped-down version of FTP server in Jupiter is not trivial because all of commands are 'tangled' inside the huge ProtocolInterpreter class.

Apart from commands, Dano also has many internal polymorphic Factories such as LoggerFactory and FTPFileFactory. The factories can return different implementations of the same thing based on a standard interface. For example:

```
FTPFile ftpFile = (new
    FTPFileFactory).getFTPFile();
```

It returns a FTPFileImpl that conforms to the FTPFile interface. That makes the program highly extendable because if a new kind of FTPFile type is needed, it only needs to provide a new FTPFile implementation and leave the old implementation unchanged. Also this architecture encourages inheritance because a NewFTPFile can inherit on the OldFTPFile. This encourages potential future reuse and makes the parts of the FTP server more reusable.

6 Evaluation

The polymorphic behaviour index appears to represent what is intended to measure. When inspecting the Lucene internal code, we see Lucene has many types, particularly searching terms and queries, that can be reused in other contexts. This is reflected by a high internal polymorphic behaviour index.

Egothor both extends the API, and internally extends the Query interface providing both external and internal polymorphic behaviour, again reflected by a high polymorphic behaviour index.

For the FTP-servers, our conclusion is that Dano's design features more classes that can be easily reused, both internally and externally.

However there is a problem. Consider the case where a declaration is made such as:

```
Parent obj = new Child();
obj.m();
```

but where the child does not have a method m() implementation:

```
class Child1 extends Parent{ ... }
```

The dispatch on obj.m() appears to be polymorphic, but is actually non-polymorphic because the Declared Interface is Parent but the dispatched method is Parent.m(). This case does not occur much in the case studies. However this is a potential problem.

Another issue is the workload used when computing the metric. Different workloads may yield different results. So far we have dealt with this issue by using the metric to just compare between applications in the same domain. Since the same workload is being provided to each application, the comparison does provide useful information. There is still the question, however, as to whether some workloads may give one application an advantage over another. For this we must be careful in choosing representative behaviour in the workloads.

A related issue is the fact that our workloads do not provide complete coverage of the applications' code. This is due to the fact that the applications considered do not have identical functionality, and so a workload that causes 100% coverage of one application will not give complete coverage of the other. As observed above, the code not covered may give that application a disadvantage with respect to our metric.

The polymorphic behaviour metric treats all polymorphic dispatches equally, which means that information about individual dispatches is lost. For example, it may be that a one call site, all dispatches are polymorphic, but at another call site, only some are. It would be interesting to consider a more fine-grained view of polymorphism to examine such behaviour.

7 Conclusions

There are many factors that affect whether or not the benefits claimed of reuse are achieved. In order to gather empirical evidence to support such claims we need to determine how to measure many attributes of software. In this paper we have considered one attribute, namely polymorphism.

We have presented a metric for polymorphic behaviour. We have developed a tool, E-MTRACE, that can measure Java applications based on this metric. We have carried out two case studies that give us confidence that this metric has value, although much more work is needed to validate it.

The polymorphic behaviour metric can provide the basis for other metrics that may be of interest. For example, it can be used to identify the interface that has the most polymorphic dispatches. Such interfaces may form the basis for new APIs or frameworks. Similarly, how many different dispatches a given interface has, or whether the interface has both polymorphic and non-polymorphic

dispatches, may be useful for API development. Finally, there are different ways in which inheritance is used to provide polymorphism. The metric may be extended to distinguish between dispatches due to an internal class inheriting from an API class, versus inheriting from another internal class.

Acknowledgements

We would like to thank the anonymous referees for their comments.

References

- Balda, D. M. and Gustafson, D. A. (1990). "Cost Estimation Models for the Reuse and Prototype Software development lifecycles." ACM SIGSOFT Software Engineering Notes vol. 15(3): 42-50.
- Banker, R., Kauffman, J., Wright, C. and Zweig, D. (1994). "Automating output size and reuse metrics in a repository based computer aided software engineering environment." IEEE Transactions on Software Engineering vol. SE-20(3): 169-187.
- Barnes, B. H. and Bollinger, T. B. (1991). "Making Reuse Cost Effective." IEEE Software vol. 8(1): 13-24.
- Bialecki, A., Cutting, D., Ganyo, S., Goller, C., Gospodnetic, O., Harwood, M., Hatcher, E. and Naber, D. (2005). Lucene project. Apache Project, retrieved from <http://lucene.apache.org/>.
- Choi, K. H. T. (2006). Dynamic Reuse Metrics. Masters Thesis, Electrical and Computer Engineering. Auckland, New Zealand, University of Auckland.
- Clements, P. and Northrop, L. M. (2001). Software Product Lines: Practices and Patterns, Addison Wesley.
- Devanbu, P., Karstu, S., Melo, W. and Thomas, W. (1996). "Analytical and Empirical Evaluation of Software Reuse Metrics." Proceedings of the 18th International Conference on Software Engineering, May.
- DISA/JIEO/CIM (1993). Software Reuse Metrics Plan. Defense Information System Agency, Joint Interoperability Engineering Organization, Center for Information Management, Version 4.1.4.
- Filion, P. (2002). Jupiter-FTP Server. Source forge project, retrieved at <http://jupiter-ftp.sourceforge.net>.
- Frakes, W. and Carol, T. (1994). "Reuse level metrics." Third International Conference on Software Reuse (ICSR '93), Rio de Janeiro, Brazil: 139-148.
- Gaffney, J. E., Jr. and Durek, T. A. (1989). "Software Reuse – Key to enhanced productivity: Some Quantitative models." Information and Software Technology vol. 31(5): 258-267.
- Galambos, L. (2006). Egothor Project. Open Source Project, retrieved from <http://www.egothor.org/>.
- Henderson-Sellers, B. (1993). "The Economics of reusing library Classes." Journal of Object-Oriented Programming vol.6(4): 43-50.
- Kain, J. B. (1994). "Measuring the ROI of reuse." Object Magazine vol. 4(3): 48-54.
- Krueger, C. W. (1992). "Software reuse." ACM Computing Surveys 24(2).
- Malan, R. and Wentzel, K. (1993). Economics of Software Reuse Revisited. Hewlett-Packard Technical Report HPL 93-31.
- McClure, C. (1997). Software Reuse Techniques, Prentice Hall.
- Meyer, B. (1997). Object-oriented software construction (2nd ed.), Prentice-Hall, Inc., Upper Saddle River, NJ, 1997.
- Mili, H., Mili, F. and Mili, A. (1995). "Reusing Software: Issues and Research Directions." IEEE Transactions on Software Engineering 21(6): 528-561.
- Paregos, M. (2002). DanofTP Server. Sourceforge Project, retrieved at <http://sourceforge.net/projects/danoftp/>.
- Poulin, J. S. (1997). Measuring Software Reuse, Addison Wesley Longman, Inc, Massachusetts.
- Poulin, J. S. and Caruso, J. M. (1993). "Determining the Value of a Corporate Reuse Program." Proceedings of the IEEE Computer Society International Software Metrics Symposium, Baltimore, MD, 21-22 16-27.
- Tracz, W., Coglianese, L. and Young, P. (1993). "A domain-specific software architecture engineering process outline." SIGSOFT Software Engineering Notes 18(2): 40-49.