# JEqualityGen: Generating Equality and Hashing Methods

Neville Grech
ECS, University of Southampton
n.grech@ecs.soton.ac.uk

Julian Rathke
ECS, University of Southampton
jr2@ecs.soton.ac.uk

Bernd Fischer
ECS, University of Southampton
b.fischer@ecs.soton.ac.uk

## Abstract

Manually implementing `equals` (for object comparisons) and `hashCode` (for object hashing) methods in large software projects is tedious and error-prone. This is due to many special cases, such as field shadowing, comparison between different types, or cyclic object graphs. Here, we present JEqualityGen, a source code generator that automatically derives implementations of these methods.

JEqualityGen proceeds in two states: it first uses source code reflection in MetaAspectJ to generate aspects that contain the method implementations, before it uses weaving on the bytecode level to insert these into the target application. JEqualityGen generates not only correct, but efficient source code that on a typical large-scale Java application exhibits a performance improvement of more than two orders of magnitude in the equality operations generated, compared to an existing system based on runtime reflection. JEqualityGen achieves this by generating runtime profiling code that collects data. This enables it to generate optimised method implementations in a second round.

***Categories and Subject Descriptors*** D.3.4 [*Processors*]: Code generation

***General Terms*** Algorithms, Performance, Theory

***Keywords*** meta-programming, equality, hashing, code generation, Java, Scala, AspectJ, AOP

## 1. Introduction

The notion of *equality* between objects is distinguished from object *identity* in that the latter is tied to the concept of individuating different objects regardless of their behaviour. Equality is a more general concept which is defined more loosely as 'some' equivalence relation between objects. The official Java documentation [1] published by Sun (Oracle) contains an object contract that specifies the details of this relation, and asks that `equals` methods behave as an equivalence relation. To support this, Java provides a default `equals` method in the `Object` class that follows this contract. It is intended that programmers override this method to define their own custom notion of equality between objects, obviously without breaking the contract. Note, however, that this contract is not at all enforced in the Java language.

The implementation of `equals` methods in many different classes can therefore be a tedious and error prone process. In fact, a number of studies [2, 3], together with our research, suggest that most of these equality methods are faulty and violate the object contract. This arises partly due to underspecification in the object contract itself, and partly due to subtleties in field shadowing, comparisons between different types, object cycles, etc. This should be quite worrying since program bugs due to equality such as symmetry and transitivity violations tend to cause errors that can be hard to track down. However, for most purposes the `equals` and `hashCode` methods are *conceptually* simple operations, and their implementations can be generated automatically. In order to do this though we must identify what notion of object equality we target.

One reasonable viewpoint is that two objects are equal if they are *semantically indiscernible*. This entails that all operations which may be performed on these objects produce semantically indiscernible results and cause semantically indiscernible changes in state. However, this relation is difficult to characterise formally and for general languages will be undecidable. In practice, we can approximate this extensional view with a finer notion of equality by simply comparing object state on a per-field basis. We adopt this view of object equality throughout the paper.

This approach is also taken by Rayside et al. [3] who describe and evaluate a generic, reflective implementation of `equals` and `hashCode` methods. Their solution relies entirely on runtime reflection to ascertain the full state of each object under comparison. Although this will adequately perform equality checks there are some drawbacks to doing this. Firstly, there is clearly a performance hit in using runtime reflection to traverse object graphs. In particular, detecting cycles in an object's state at runtime is unnecessarily costly. Secondly, the implementation of the equality method is generic and hence not available for analysis or specialisation. Instead, we adopt an approach where we statically generate equality implementations by using reflection in MetaAspectJ. The generated code is actually in the form of an aspect which is then statically woven in to the application's source code. This allows our development tool, JEqualityGen,[1] to be smoothly integrated in to the build process of large projects. Since we are effectively providing support at the bytecode level rather than at the Java source code level, JEqualityGen can also work for other programming languages that are compiled into the JVM, and we demonstrate this for Scala. Since JEqualityGen statically generates specialised implementations which require no runtime reflection, a typical performance improvement of almost two orders of magnitude can be observed over the runtime reflection solution by Rayside et al. [3]. A further speed-up can be achieved by adapting the order in which fields are compared to the application profile. JEqualityGen supports this by generating runtime profiling code that collects data which can then be used in a second round to automatically generate the optimised method implementations.

For the remainder of this paper we begin by identifying the challenges we are faced with in implementing equality methods. We



*GPCE'10,* October 10–13, 2010, Eindhoven, The Netherlands.


[1] `http://sourceforge.net/projects/jequalitygen/`

do this by considering the common causes of faulty equality implementations. This requires us to consider how the object contract impacts upon the comparison of objects at different but comparable types and also how the class hierarchy affects comparisons. Further to this, we need to be able to account for equality between objects with cyclic reference graphs. For example, consider two linked list objects in which one list is a single node cycle containing a value, and the other is a two node cycle in which both nodes contain the same value. These two objects should be considered equal and our generator recursively checks equality over such cyclic object structures.

An issue closely related to the generation of `equals` methods is the generation of the related `hashCode` method. This method is used for providing a hashed value of the receiver object's state and is supported in Java similarly to method `equals` by providing a default implementation in the class `Object`. What is worth noting is that the interplay between `hashCode` and `equals` is specified as part of the object contract. In particular, the hashed values must respect equality of objects. For this reason, we also support the generation of `hashCode` methods in JEqualityGen. Moreover, we provide support for compile-time warnings of mutations of key fields which may adversely affect hash code calculations.

In Section 4 we give an overview of JEqualityGen's architecture and provide a more detailed description of issues related to its implementation. This includes various code optimisations which have been made to both simplify the code of and improve performance of the generated methods. Given that we designed JEqualityGen with performance in mind, we also provide a mechanism for runtime profiling of equality operations. JEqualityGen can then use profiled information in order to regenerate code that optimises the order in which comparisons are made within equality operations. We demonstrate the effectiveness of code generation and our optimisations by running benchmarks comparing JEqualityGen and Rayside's [3] system. These benchmarks, together with other test cases, are based on a popular Java charting library, JFreeChart [4] .

## 2. Implementing Equality

In this section we consider the Java object contract and discuss the problems that arise with naive implementations of equality according to this model.

### 2.1 The Java object contract

The official Java documentation [1] describes the contract the `equals` method has to follow.

> The equals method implements an equivalence relation on non-null object references:
>
> - It is reflexive: for any non-null reference value `x`, `x.equals(x)` should return `true`.
> - It is symmetric: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
> - It is transitive: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
> - It is consistent: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the objects is modified.
> - For any non-null reference value `x`, `x.equals(null)` should return `false`.

We can see immediately from this contract that it does not specify exactly how equality should be implemented and it will accept object identity (also called *referential equality*) as an implementation. Indeed, object identity is the default implementation provided in class *Object*. Object identity is tackled extensively by Khoshafian and Copeland [5], who also define a taxonomy of different object identity implementation strategies. The simplest form of identity test is a comparison of the physical memory addresses of the objects. Other implementations compare virtual addresses, structural identifiers or user-specified identifier keys. In other systems such as PostgreSQL, objects contain a system-generated object identifier unique for every relation.

Java distinguishes between object identity, which is encoded in the `==` operator and remains fixed, and object equality, which is handled by the `equals` method and defaults to identity, but can be overridden.

There are different natural choices for implementing object equality. The common approach is to perform a state-based comparison between objects of the same, or at least similar types. Khoshafian and Copeland [5] define different levels of equality, namely shallow and deep equality while Grogono and Sakkinen [6] refine this concept.

Referential equality, also called *depth-0 equality*, holds for $a$ and $b$ iff $a$ and $b$ both point to the same objects. For primitive types such as integers, this is the only type of equality we are interested in. *Shallow equality* (or *depth-1 equality*), like shallow cloning, implies that for each field in $a$ and $b$, referential equality holds. *Depth-$k$ equality* holds on objects $a$ and $b$ if all fields in $a$ are depth-$k'$ equal to the corresponding fields in $b$, for all $k' < k$. We refer to this type of equality as *deep equality* in case $k = \omega$. Objects $a$ and $b$ are thus deeply equal if all fields in $a$ are deeply equal to the fields in $b$. Abiteboul and Van den Bussche [7] discuss three different logical characterisations of deep equality and show that they are equivalent.

An elegant property of referential, shallow, depth-$k$ and deep equality is that each type of equality implies the next. For example, reference equality implies all other aforementioned types of equality since if two objects are identical they are obviously equal in all possible aspects. In this paper we address the implementation of the coarsest of these relations, deep equality. A naive approach to this problem would be to implement equality in a given class by making a per-field comparison of equality on each field of the class. For fields of primitive type their values can be compared directly using depth-0 equality. For fields of reference type the equals method can be invoked recursively. There are two glaring problems with this approach—this solution does not interact well with subtyping at all, and it will cause divergent behaviour on cyclic object graphs. We now consider these problems in turn and consider how to address them.

### 2.2 Comparing objects with different types

Equality comparisons between two objects of different types are difficult in general. Vaziri et al. [2] claim that an `equals` implementation breaks the Java object contract [1] in this case. An implementation cannot be symmetric and transitive when it compares two objects of different types that might have a different interface as well as a different implementation.

It is, however, desirable to allow, for example, `TreeSet` and `HashSet` objects from the Java API to be comparable since they can be interchanged while maintaining the same behaviour of the system. A pragmatic solution to achieve this is to perform equality on different classes at a particular super-type level, although, in practice, this presents a number of issues. For example, Hovenmeyer et al. [8] note that one common mistake is to have equality methods that return `true` even though the object types under con-

```
public class Point {
  public int x, y;
    ..
}
public class CPoint extends Point {
  public int col; // extra field
  public CPoint(x,y,col) {
    this.x=x; this.y=y; this.col=col;
  }
  public boolean equals(Object other) {
    if (!(other instanceof Point))
        return false;
    Point that=(Point)other;
    return this.x==that.x && this.y==that.y
        && !(other instanceof CPoint &&
             ((CPoint)other).col != this.col);
  }
}
..
CPoint p1 = new CPoint(1,2,3);
Point  p2 = new Point(1,2);
CPoint p3 = new CPoint(1,2,4);
assert p1.equals(p2);
assert p2.equals(p1); // symmetry: ok
assert p3.equals(p2);
assert p3.equals(p1); // transitivity: error
..
```

**Listing 1.** Transitivity violations occur since this equality is stronger on CPoint than on Point.

sideration are incomparable. Another issue is that access to some `private` members is impeded.

It is common that `equals` implementations that work across different types start with an instance check that short-circuits the entire operation in the case of incompatible types. However, this tends to cause problems if we compare two objects whose respective types are subclasses of each other. Odersky et al. [9] note that the instance check fails depending on whether `equals` is called on one object or the other, which violates symmetry. For example, an `FPoint` object is an instance of `Point` (whose class definitions are both shown in Listing 2) but a `Point` is not necessarily an instance of `FPoint`. A better `equals` design, as for example presented both in [9] and [10], takes the type of its parameter into account, and has different implementations for different types. This can be seen in Listing 1. This approach will recover symmetry of equality but will still violate transitivity.

In order to ensure both symmetry and transitivity, Odersky et al. [9] suggest that each class should implement another method, `canEqual(Object o)`, which indicates whether the object on the right hand side of the comparison can compare itself with the object on the left hand side. The result from this method is conjoined to the equality expression. This guarantees that instance checks are always symmetric, as long as every class in the hierarchy defines this method. An example of its use can be seen in our generated code (cf. Listing 8).

Reflexivity of equality is the easiest property in the object contract to guarantee as a successful reference equality check at the beginning of an equality operation can be used to short circuit the entire process; this also enhances performance.

**Incorrect override.** A number of authors [2, 3, 8–10] argue that a common mistake that can easily remain undetected is that of specifying an `equals` method with an incorrect signature. In this case, equality checks default to the `equals` method defined in the `Object` class, which only performs a reference equality check. This can lead to errors that are very hard to track down, e.g., when

```
class Point {
  public int x, y;
  public Point() {}
  public Point(int x, int y) {
    this.x=x; this.y=y;
  }
  public boolean equals(Object o) {
    if (!(o instanceof Point)) return false;
    Point that=(Point)o;
    return this.x==that.x && this.y==that.y;
  }
}
class FPoint extends Point {
  public int x, y; // shadows x and y in the
    Point class
}
..
Point p1=new Point();
FPoint p2=new FPoint();
p1.x=5; p1.y=5; p2.x=5; p2.y=5;
assert p2.equals(p1);   // returns false
```

**Listing 2.** Direct field access and inherited equality

the equality is called out of some library method such as a collection's `contains` method. Automatically adding `@override` annotations to the manually implemented `equals` methods would allow the compiler to detect these incorrect override errors. However, if the methods themselves are automatically generated instead, then this problem is not only detected but resolved.

**Field shadowing when sub-classing.** Java permits the overriding of fields throughout a class hierarchy. Unfortunately, this presents a number of challenges when implementing equality methods that compare objects of different types. For a simple example, consider again the `Point` class in Listing 2 containing two integer numbers and a `FPoint` class that extends `Point` and shadows its fields. The naive implementation of equality in this situation is to implement a single `equals` method in the `Point` class that works for both classes, and directly accesses both fields of the objects being compared. However, Java does not use the dynamic type of an object when resolving field accesses, but its static type. Since `p2.equals(p1)` is dispatched to `Point.equals`, it does thus not operate on the `FPoint` fields, but instead on the shadowed `Point` fields. By default, these are set to zero and therefore `p2.equals(p1)` evaluates to `false`, even though the two points are created with the same coordinates.

In Listing 3, `equals` is overridden in `FPoint`, in the hope that equality would use the correct fields when making the comparisons. With this modification, even though `p2.equals(p1)` returns `true`, `p1.equals(p2)` returns `false` and therefore symmetry is violated. This happens because the `equals` method called on `p2` is `FPoint.equals` and uses the fields in `FPoint` while the `equals` method called on `p1` is `Point.equals`, which only sees `Point.x` and `Point.y`, as above.

Implementing getter methods and using them in the equality operations solves this problem. Care must be taken however, as the getter methods operate on the fields that are visible at that level in the class hierarchy. Therefore these methods must be overridden together with all equality methods.

Listing 4 shows a correct implementation with respect to field shadowing.

### 2.3 Cyclic object graphs

A cyclic object graph can easily occur when objects are referencing each other. If the developer writing the `equals` (or `hashCode`) methods is not aware of this, an invocation of such methods would

```
class Point {
  public int x, y;
  public boolean equals(Object o) {
    if (!(o instanceof Point)) return false;
    Point that=(Point)o;
    return this.x==that.x && this.y==that.y;
  }
}
class FPoint extends Point {
  public int x, y; // shadows x and y in the
    Point class
  public boolean equals(Object o) {
    if (!(o instanceof Point)) return false;
    Point that=(Point)o;
    return this.x==that.x && this.y==that.y;
  }
}
..
Point p1=new Point();
FPoint p2=new FPoint();
p1.x=5; p1.y=5; p2.x=5; p2.y=5;
assert p2.equals(p1);
assert p1.equals(p2); // false - error
```

**Listing 3.** Direct field access and overridden equality

```
class Point {
  public int x, y;
  public int getX() { return x; }
  public int getY() { return y; }
  public boolean equals(Object o) {
    if (!(o instanceof Point)) return false;
    Point that=(Point)o;
    return getX()==that.getX() &&
      getY()==that.getY();
  }
}
class FPoint extends Point {
  public int x, y;
  public int getX() { return x; }
  public int getY() { return y; }
  public boolean equals(Object o)
    if (!(o instanceof Point)) return false;
    Point that=(Point)o;
    return getX()==that.getX() &&
      getY()==that.getY();
  }
}
```

**Listing 4.** Correct implementation. Overriding equality methods and accessors

never return and would consequently overflow the call stack. Ignoring fields that may be involved in a cycle would make the method terminate without overflowing the stack, but it would also make the equality method unfaithful to the abstract state of the original object [3]. In fact, it would either identify all cyclic object graphs, or make them all distinct.

It is, however, possible to write `equals` (and `hashCode`) methods that can deal with cycles. One approach that was already used in Eiffel [11] is to assume that two objects are, prima facie, equal. Their object graphs are then traversed in parallel, and their corresponding fields are compared, in search of evidence to refute this assumption. Since no more evidence can be obtained by traversing a cycle multiple times, we can assume that the objects in a cycle are equal. Figure 1 shows an example of this. Note that cyclic objects can be equal even if their object graphs

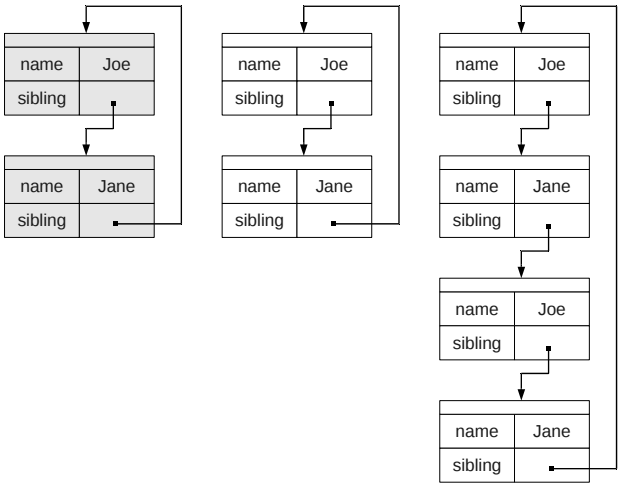


**Figure 1.** Comparing the "grey" Joe object to either of the "white" Joe objects using a naive equality implementation would never terminate.

are not isomorphic. We do not support an equality check based on graph isomorphism, but the modular implementation of JEquality-Gen should facilitate such a change. For `hashCode`, whenever a cycle is encountered, the object structure cycle's hash is substituted by a constant number. We follow this approach to handle cyclic object structures, which is similar to Rayside et al. [3].

## 3. Implementing Hashing

### 3.1 The relationship between equals and hashCode

Although not enforced by the compiler, the `Object` contract [1] also specifies a clear relationship between the `equals` and `hashCode` methods:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to `Object.equals`, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- If two objects are *not* equal according to the method `Object.equals`, then calling the `hashCode` method on each of the two objects may or may not produce the same integer result.

Listing 5 demonstrates what happens if `hashCode` is not in line with the `equals` method. Since the `HashSet` implementation in Java uses the hash code of an object to search for the actual object in the collection, objects with different hash codes are considered not equal. This will occur in our example since the default implementation of `hashCode` returns a value based on the location of the object. The `ArrayList` structure, on the other hand, does not make use of hash codes.

Rayside et al. [3] analyse three different Java projects and conclude that simple errors are all too common. One of the simplest errors is when `equals` is implemented but `hashCode` is not. A

```
Point p1 = new Point(1,2,3,4);
Point p2 = new Point(1,2,3,4);
assert p1.equals(p2) && p2.equals(p1);
HashSet<Point> pSet    = new HashSet<Point>();
ArrayList<Point> pList = new ArrayList<Point>();
pList.add(p1);
pSet.add(p1);
assert pList.contains(p2); // true
assert pSet.contains(p2);  // false -- error
```

**Listing 5.** Consequence of equals not being in line with hashCode

number of tools [8, 12] can easily spot this trivial mistake and enforce implementation of both methods at once. A human inspector however can easily miss this mistake because "the mistake lies in what is missing" [8]. Similarly, on larger projects, changes in the structure of the class require changes in `equals` and `hashCode`. Often, these changes are overlooked, especially considering that there is no enforcement mechanism in the Java language. If these methods are automatically generated then these problems are easily solved.

### 3.2 Consistency of key fields

Vaziri et al. [2] note that the object contract does not require that key fields be immutable. There are, however, undesirable consequences in allowing key fields that make up the abstract state of an object to mutate during runtime. A minor consequence is that equality and hash results cannot be cached (memoisation). A more serious consequence is that if an object is placed into a collection, the operations `add`, `remove` and `contains` will exhibit an unexpected behaviour. For example, in the case of a `HashSet`, if an object is added, it is stored in a hash bucket determined by the value of its hash code. Mutating one of the key fields in this object effectively changes the object's hash code, and it can no longer be retrieved since it resides in a different bucket that does no longer correspond to its new hash code.

Countering this problem entails that equality and hashing should be based on fields that are immutable. The Java specification, however, does not enforce this constraint. Ideally the Java runtime system would check whether an object's fields are mutated after the invocation of the first `equals` or `hashCode` and issue a runtime exception or warning.

## 4. JEqualityGen: Architecture and Implementation

JEqualityGen is a code generator that automatically generates `equals` and `hashCode` methods from annotated class archive files. In building JEqualityGen, we make use of aspect oriented programming (AOP) techniques, because we believe that object equality is a cross-cutting concern. In particular, we use Meta-AspectJ [13], a meta-programming extension for AspectJ [14]. Meta-AspectJ leverages the program transformation capabilities of AspectJ such as inter-type declarations. This enables us to statically weave the generated code into the existing Java bytecode. We also rely on AspectJ's runtime reflection, in particular its ability to inspect the call stack, to handle cyclic object graphs.

### 4.1 Overview

Figure 2 gives an overview of the structure of JEqualityGen. It works by loading the user's classes and, using reflection, statically analyses each class and generates AspectJ aspects with the appropriate equality and hashing implementations. These aspects are woven into the user's existing classes using the AspectJ compiler. All operations are therefore carried out on compiled Java classes. This makes JEqualityGen easy to integrate into the build process.

The code generation is modularised; each module is responsible for certain elements of the generated aspects. A set of introspectors act as facades [15] to the annotated Java classes. They analyse the actual classes and provide the basic information used by the remaining parts of the generator, including:

- which classes may be involved in cycles;
- which fields are being shadowed; and
- the order in which to best structure the equality expression based on separately collected runtime profiling information.

This approach allows us to separate our code generator into smaller generators. For example, there is a generator responsible for the "naive" equality implementation and another generator for the cyclic handlers. These generators are independent from each other, and since they do not impinge on each other's generated code, they can be switched on and off. We then rely on AspectJ's weaving facilities to "recombine" the individual fragments into a single implementation, instead of generating a monolithic implementation of equality we could make use of AspectJ's weaving facilities.

A *field accessor facade* acts as a facade [15] to each of the individual fields in a class. These facades are implemented in MAJ and can be viewed as a collection of generators that generate per field advice for the final generated aspect. Every facade generates:

- accessor expressions for the particular field;
- getters for the field; and
- mutation warnings and errors advice to ensure that key fields are not mutated in certain instances

The output of JEqualityGen is a single aspect that contains all the equality implementations, getters, warning declarations and cyclic-object graph advice. The implementations of the `equals` and `hashCode` methods constructed follow the guidelines described in Sections 2 and 3.

### 4.2 Annotations

In order to make use of JEqualityGen, the classes for which `equals` and `hashCode` should be generated must be annotated with a few simple annotations:

**Equality** JEqualityGen will generate appropriate equality implementations for classes that are annotated with this annotation. A super-type may also be specified so that objects may be compared at that specific level.

**ReferenceEquality** JEqualityGen will use Java's default reference equality (==) when classes annotated with this annotation are encountered.

**NonKey** Fields annotated as non-keys will *not* be considered as key fields and not be used for equality comparisons and hash code computations. All fields are considered as key fields by default.

In cases where the code that requires an equality implementation is inaccessible to the weaver (see Section 4.6), any information about the classes required in JEqualityGen can be given through command line arguments.

Annotating classes that already implement either `equals` or `hashCode` methods leads to a compile-time error, in particular, a duplicate method declaration, as we use inter-type declarations to insert the generated methods into the target classes.

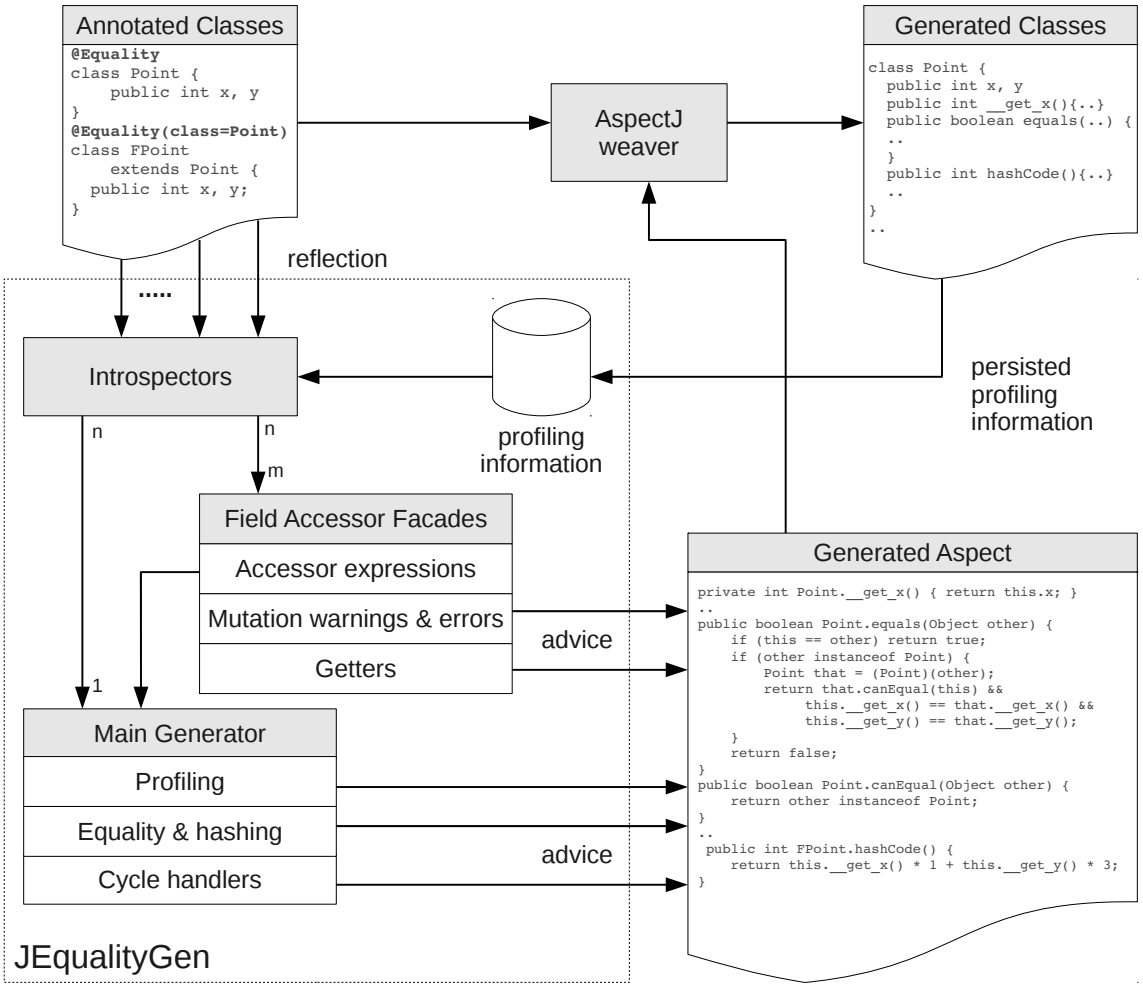


**Figure 2.** Structure of JEqualityGen

### 4.3 Generating equals methods

In order to implement equality, we generate two methods, `equals`, which is the main operation, and `canEqual`, which determines whether its argument is an instance of the correct type. An example of the generated methods is given in Listing 8. It is surprising to see that it takes so much code to implement proper equality operations for such conceptually simple classes.

The `equals` methods follow a specific template, and performs the following steps:

- coerce the argument object to the correct equality type (i.e., the type on which the equality is evaluated);
- check whether the receiver object is comparable to the argument, and conversely check whether the argument is comparable to the receiver;
- profile how often fields differ between different objects, to determine in which order they are compared; and
- (recursively) check whether each key field in the receiver object is equal to each key field in the other object.

An example of a generated equality expression, as discussed in the last item, is shown in Listing 6. The exact form of the equality expression for each field depends on the type of the field, and the code generation task was split into multiple parts, depending on the type of the field.

If a field is of a primitive type or a simple reference type which requires only reference equality (a class annotated with `@ReferenceEquality`), the Java == operator is used for equality. An expression that evaluates to the value of the field is generated (for example `this.__get_x()` for field `x`). The selection depends on whether the field can be accessed without the need of an accessor, whether it needs a custom accessor, or whether it requires the use of a standard Java accessor.

A problem which arises when comparing floats or doubles is that nothing is equal to `Float.NaN` or `Double.NaN`. JEqualityGen uses the function `floatToIntBits` in the case of floats and `DoubleToLongBits` in the case of doubles. In the case of an array, if the enclosing type of such array is yet another array, the `Arrays.deepEquals` function is used. In the simple case of having an array of a non-array `Arrays.equals` is used.

If the field is not a primitive type, the `equals` method needs to be used. However, JEqualityGen also needs to make sure that the receiver object is not null. Nullity checks are therefore added to the generated expression.

```
..
(this.p2 == that.p2 || (this.p2 != null && this.
  p2.equals(that.p2))) && Double.doubleToLongBits
  (this.d1) == Double.doubleToLongBits(that.d1)
  && Float.floatToIntBits(this.f1) == Float.
  floatToIntBits(that.f1) && this.c1 == that.c1
  && Arrays.equals(this.s, that.s) && Arrays.
  deepEquals(this.s22, that.s22)
..
```

**Listing 6.** Generated equality expressions for different types p2:Object, d1:double, f1:float, c1:char, s:1d array, s22: 2d array

```
..
(this.p2 == null ? 0 : this.p2.hashCode()) *
  28629151 + ((int)(Double.doubleToLongBits(this.
  d1) ^ Double.doubleToLongBits(this.d1) >>> 32))
   * 887503681 + Float.floatToIntBits(this.f1) *
  1742810335 + (int)(this.c1) * -1807454463 +
  Arrays.hashCode(this.s) * -196513505 + Arrays.
  deepHashCode(this.s22) * -1796951359
..
```

**Listing 7.** Generated hashCode expression for types p2:Object, d1:double, f1:float, c1:char, s:1d array, s22: 2d array

**Finding fields that require getters.** In Section 2, we have seen that field shadowing leads to unexpected equality results. It is therefore necessary to add getters whenever the fields being declared are shadowed.

Fields that need getters are identified by traversing the class structure, noting which fields are declared at every level. If throughout the search a field is found to be declared at more than one level, both of these fields are added to the results set. This data is later used to write the appropriate getters for the fields, and JEqualityGen always accesses these fields using the generated custom getters.

### 4.4 Generating hashCode methods

For the `hashCode` method, JEqualityGen generates an integer expression, rather than a boolean expression, but the underlying logic is similar to that of the equality generators. As in the case of the equality expression, dispatching is done according to the type of the field. In the case of a primitive field, dispatch is done over these various primitive types as follows:

**Boolean** The expression generated evaluates to a particular constant value in the case of `true` and a different constant in the case of `false`.

**Character** The character is cast to an integer.

**Float** The `Float.floatToIntBits()` function is used to get an integer value from the `Float`.

**Long** The 32 MSBs are `xor`-ed with the 32 LSBs.

**Double** The `Double.doubleToLongBits` function is used and the resulting 32 MSBs are `xor`-ed with the resulting 32 LSBs.

The individual expressions are then conjoined to form a Kernighan and Ritchie multiplicative hash expression [16]. Listing 7 shows part of a generated hashing expression for various field types. We have chosen this hash function because it is fast. The modular design of JEqualityGen makes it easy to change the particular hashing method. This could even be done based on runtime feedback.

```
private int Point.__get_x() { return this.x; }
private int FPoint.__get_x() { return this.x; }
private int Point.__get_y() { return this.y; }
private int FPoint.__get_y() { return this.y; }
public boolean Point.equals(Object other) {
  if (this == other) return true;
  if (other instanceof Point) {
    Point that = (Point)(other);
    return that.canEqual(this) &&
      this.__get_y() == that.__get_y() &&
      this.__get_x() == that.__get_x();
  }
  return false;
}
public boolean Point.canEqual(Object other) {
  return other instanceof Point;
}
public boolean FPoint.equals(Object other) {
  if (this == other) return true;
  if (other instanceof Point) {
    Point that = (Point)(other);
    return that.canEqual(this) &&
      this.__get_y() == that.__get_y() &&
      this.__get_x() == that.__get_x();
  }
  return false;
}
public boolean FPoint.canEqual(Object other) {
  return other instanceof Point;
}
public int Point.hashCode() {
  return this.__get_y() * 1 +
    this.__get_x() * 31;
}
public int FPoint.hashCode() {
  return this.__get_y() * 1 +
    this.__get_x() * 31;
}
```

**Listing 8.** Generated equality and hashing methods and accessors for the Point and FPoint classes in Listing 14 and the code snippet in Figure 2

```
declare warning: set(* Point.x1)  &&
  !withincode(Point+.new(..)):
    "Point.x1 is declared as key but is being
      mutated outside the constructor.";
```

**Listing 9.** Warnings are given whenever a key field is mutated outside the constructor

**Disallowing mutation of key fields.** As discussed in Section 3.2, mutating key fields leads to problems if the object is stored in a collection which uses the hash-code, because it becomes impossible to retrieve the object if its hash-code changes. For this reason, JEqualityGen generates warning declarations (Listing 9) that issue mutation warnings at compilation time. These are useful since they indicate to the programmer any locations where the mutation can occur.

In addition, JEqualityGen generates advice, as shown in Listing 10, that ensures that once `hashCode` has been called, no key fields may be mutated.

### 4.5 Handling Cycles

The code generated in Listing 8 does not take into consideration potential cycles in the object graph. Of course, this is a "good thing": given the class declarations, it is clear that there cannot be a

```
private transient boolean Point.hCalled = false;

before(Point self) :
  execution(* hashCode()) && target(self) {
  self.hCalled = true;
}
before(Point self, int val):
  if(self.hCalled && !(self.x1 == val)) &&
    set(* Point.x1) && target(self) && args(val){
      throw new KeyMutationException("Point.x1
        has been mutated");
}
```

**Listing 10.** An exception is raised whenever a key field is mutated if hashCode has been called already on that object

cycle, so there is no need to check for it. Hence, in order to reduce the size and increase the speed of the generated code, JEqualityGen makes use of a simple algorithm that statically detects whether cycles are possible at all.

**Determining whether a class might be involved in cycles.** In general, a class might be involved in a cycle if it contains a field which may be assigned from itself. By extension, this would happen also if any of a class's fields might in turn contain a field assignable from the original class. This property extends itself recursively.

Although not much code is required to determine such an eventuality, such an algorithm lends itself to bugs. Given a particular container class and a containee (initially the same class), JEqualityGen goes through each of the container class's fields (one parent class at a time) and sees whether any of the fields is assignable from the containee. If the field is not a primitive type, JEqualityGen then in turn sees whether the containee may be contained in this field. This process is invoked recursively until all fields in the class hierarchy are tested. Classes which have already been traversed do not need to be traversed a second time.

**Advice for cycle handling.** Cycles are handled by generating advice that uses a point-cut descriptor as shown in Listing 11. The executing advice uses one stack containing the visited objects for every different class that may be involved in cycles. If the current target of the point-cut is present in the stack, then the execution has reached a cycle. In such a case, object graphs involved in the cycle must be equal since the equality expression would have short-circuited otherwise. This would have terminated our equality computation, returning `false`. On the other hand, if the target object is not found on the stack, it is pushed on the stack. The execution then `proceed`s with the original equality computation and removes the target from the stack when the computation returns. The result is finally returned.

### 4.6 Generating code for inaccessible classes

JEqualityGen was initially designed to only operate on classes which the user has control over. This assumes that classes in external libraries that are used in the user's code have a correct implementation of `equals`. This is typically not the case.

The simple approach to extending an inaccessible class is to use an AspectJ inter-type declaration and let the AspectJ weaver weave in the equality method. However, it is impractical to have all libraries in the AspectJ `inpath`. For example, putting `rt.jar` in the `inpath` would crash the weaver.

The approach used in JEqualityGen instead is that of extending (i.e., sub-classing) the class and adding the actual equality logic in the extended class. All calls to the constructor of the original class are intercepted and replaced by calls to the extended class. This not

```
private transient Stack mrh = new Stack();

int around(MockCycle1 self):target(self)
    && execution(int hashCode())
    && cflowbelow(target(MockCycle1)
    && execution(int hashCode())) {
  for (int i = 0; i < mrh.size(); i++)
    if (self == mrh.get(i))
      return 127; // breaks the cycle
  mrh.push(self);
  int res;
  try {
    res = proceed(self);
  } finally {
    mrh.pop();
  }
  return res;
}
```

**Listing 11.** Cyclic advice for a hashCode method

only solves any problems in the auto-generated `equals` methods but also in any other invocations of `equals` on the original classes in the user's code.

Privileged access of classes require AspectJ to weave special accessors into the accessed classes. Unfortunately, this entails that the original classes are advised. Since these classes form part of the Java library and the state would be exposed through the Java API in a standard and consistent manner, JEqualityGen only accesses an object's state through its getters in the case of external libraries.

An experimental feature in AspectJ has to be used to make the generated aspect serialisable. All fields in the aspect are however marked as `transient` as these are not of any importance for the serialisation. The result is that any relevant aspects are now serialised with the classes they advise.

## 5. Runtime statistics-based optimisations

JEqualityGen can instrument the equality operations it generates in such a way that they gather statistics about the fields in the class; specifically, the methods keep a tally of how often each field in each class has failed the equality operation. This can then be used to generate optimised equality operations, simply by placing the fields with the highest probability of failing the equality operation at the beginning of the expression. This increases the chances of these fields failing the equality test early, and thus short-circuiting the equality operation.

### 5.1 Gathering and persisting statistics

When JEqualityGen is instructed to gather statistics about the generated code, the following steps take place:

- the code to declare the necessary data structures that record field statistics at runtime is generated;
- the necessary code fragments that perform the profiling are generated; and
- the code that is invoked before the program terminates is generated (see Listing 12). This code persists the profiling information to disk.

Listing 13 shows some of the code that gathers the statistics. The per-field equality expressions generated by the field accessor facades are also used as arguments to the `tally` method in our statistics gathering classes. In order to properly gather the statistics, the individual field equality expressions must, however, not short-circuit, and the effects of all fields must go into the tally. However,

```
after() : execution(static void *..main(..)) {
  try {
    fieldStatistics.persist("./profileinfo.dat");
    ..
  } catch (IOException ioe) {
    ..
  }
}
```

**Listing 12.** Advice which is executed before the application exits

```
..
fieldStatistics.get("Line.p1").tally(
  (this.p1 == that.p1 || (this.p1 != null && this
    .p1.equals(that.p1))));
fieldStatistics.get("Line.p2").tally(
  (this.p2 == that.p2 || (this.p2 != null && this
    .p2.equals(that.p2))));
fieldStatistics.get("Line.d1").tally(
  Double.doubleToLongBits(this.d1) == Double.
    doubleToLongBits(that.d1));
..
```

**Listing 13.** Profiling code which is inter-posed in the equality function

the functionality of the `equals` operation is unaffected by the profiling code.

### 5.2 Re-generating the equality methods

The re-generation of the equality methods follows the same process as the normal generation of the equality implementation, with one important difference. When JEqualityGen is launched in this mode, it reads the profiling information which was generated by a previous run of the program. This information is used by the introspectors on a per-class basis. The fields in the introspector are supplied to the main generator sorted in an optimised way.

The sorting happens according to the statistics gathered by the profiling code. For each field, we calculate the ratio of the number of times its equality operation fails divided by the number of times its equality operation succeeds. For every class, we sort the fields according to their corresponding ratio, in descending order. Whenever an equality operation is generated, the fields which make up the expression are ordered in this manner.

The performance effect in the best case is exponential to the depth of the object graph that is being compared. This can easily happen since this optimisation is applied for every class. In turn, the fields of each class also get this optimisation if their equality is also generated by JEqualityGen. In practice the effect of turning on this optimisation can be seen in Table 1.

## 6. Experimental Evaluation

### 6.1 Performance analysis

Since JEqualityGen uses code generation rather than reflection, we certainly expected an improvement in performance. In order to evaluate to which degree the performance improvements materialised, we wrote a benchmark that exercises the `equals` and `hashCode` methods of a number of classes from the JFreeChart project [4]. JFreeChart is a charting library that is part of the DACapo Benchmark Suite [17]. It contains 1158 classes, with 8960 methods. We generated equality and hashing methods for 478 of these classes, of which 101 had cyclic handling advice generated for. This means that 101 of these classes could *potentially* be involved in cycles. JEqualityGen generated 25000 lines of code for this project. The benchmark exercised the `equals` and `hashCode` methods of the the top level container class `org.jfree.chart.JFreeChart`. This class contained objects of most of the classes in the project.

| | JEG | JEG w/profiling optimisations | Rayside et al. [3] |
|---|---|---|---|
| `equals` | 2297 | 1108 | 179856 |
| `hashCode` | 3602 | n/a | 86683 |

**Table 1.** Time to run benchmark in ms

We compare JEqualityGen to the system presented by Rayside et al. [3], and benchmarks were run on both of these systems. Since the system by Rayside et al. uses caching to enhance performance, we ran the benchmark loop several times before starting the timer. This enabled both the JVM and the implementation of Rayside et al. to warm up.

Table 1 lists the results of running these benchmarks on a Lenovo T500 2.4GHz under 64-bit Debian running sun-java-6. JEqualityGen is able to produce `equals` methods that are about 162 times faster than [3] and `hashCode` methods that are about 31 times faster.

We note that given the sheer size of JFreeChart and the complexity of its class structure, invoking reflection on an entire object graph is much slower than a direct field access. Another reason why Rayside et al.'s solution is slower is that a lot of dispatching and analysis is carried out at runtime, while in our case this is carried out at *code generation time*. A case in point is the cycle detection optimisation that is done at code generation time. Runtime feedback and re-ordering the equality expression also helps to boost the performance of JEqualityGen's generated code, by a factor of two. Note that this is not applicable the same way for computing the hash codes, because the hash code must be computed from all key fields.

### 6.2 Correctness analysis

We initially planned to analyse the correctness of JEqualityGen by statically verifying the generated bytecode. This would have allowed us to prove that our implementation yields the correct notion of equality and to compare it to a reflective solution. Unfortunately, there are no mature Java bytecode verifiers available and therefore we had to resort to normal testing.

In order to assess the correctness of JEqualityGen, we modified the JFreeChart project to utilise our code generator for the equality and hashing implementations rather than using the manual implementations. Given the size of the project, this served as a good test case for JEqualityGen and it also influenced some of our design decisions. There were some problems we encountered throughout our testing, namely:

**Hard-coded hash-codes** Since our auto-generated hash functions are different (but still correct), test cases expecting a specific hash value for some objects obviously fail.

**Incorrect equality implementations** Some equality test cases are not faithful to the state of the object. For example, serialising and de-serialising the object would change the object. Other implementations were buggy for other reasons. Some test cases were written in such a way that a correct implementation fails.

**Key mutation** Whenever a key field is mutated in an object after the `hashCode` method is called, an exception is raised. Unfortunately, this runtime check caused some tests to fail. It was shown in Section 2 why key fields should not be allowed to mutate.

```
@Equality
class Point(x: int, y: int) {
}
@Equality{val eqClass = classOf[Point]}
class FPoint(x: int, y: int) extends Point (x,y){
}
```

**Listing 14.** Skeleton of an annotated Point and FPoint classes with equality performed at the Point level, in Scala.

Discounting for these cases, however, the modified JFreeChart implementation succeeded on all other test cases.

### 6.3 Compatibility with other JVM languages

One of the main advantages of working at the JVM bytecode level is that JEqualityGen is in principle independent of the actual source language on which it is being used, as long as this compiles to the JVM bytecode. Obviously, this is limited by AspectJ's ability to process this bytecode. Also note that the notion of equality will carry over from Java, and might not be appropriate for a third language. Existing code generators such as the *generate equals() and hashCode()* feature in Eclipse only work on a specific source language. In order to demonstrate the compatibility with other JVM languages, we applied JEqualityGen to Scala. Listing 14 shows a very simple Scala code snippet, which defines the same two classes we have used in some of our previous examples. The generated equality aspect is obviously exactly the same as the one generated for the classes implemented in Java, which can be seen in Listing 8.

## 7. Conclusion and Future Work

Implementing equality and hashing operations is both tedious and error-prone. JEqualityGen was developed specifically to address the pitfalls associated with these operations and to relieve the developer of the burden of implementing them. Code generation technology can be employed to address this problem, making the resulting implementations fast, efficient, and easier to verify in principle. Our prototypical implementation is expressive enough as a drop-in replacement in the context of large Java applications. It can also be integrated into the build systems of these applications with relative ease.

Apart from the substantial performance improvement we registered in our benchmarks, an advantage of code generation is that static analysis and formal verification tools can work with the generated code to infer some properties from the system. It is also possible for tools such as AspectJ to weave advice directly into the generated code. Another advantage of the static analysis of code is that we can issue warnings and errors at *code generation* time while other runtime systems would throw exceptions at runtime, which is much less convenient.

Apart from the usual object contract issues, we have addressed other practical issues such as field shadowing, which simple tools such as the *generate hashCode() and equals()* feature in Eclipse [18] fail to handle. This code generator is also naive in the sense that it does not concern itself with the interactions between different classes. As a result, inheritance and cyclic structures are not handled well. We are not aware of any other system that generates equality methods and takes field shadowing into consideration. Another big advantage of JEqualityGen is that even though it generates code, it can still be used with languages other than Java that run on the JVM such as Scala.

JEqualityGen also works for Java source code that uses Java generics. However, we did not tackle concurrency issues. If for example, an object is mutated while it is being compared, the behaviour of our equality methods would be undefined. A possible area of improvement would be to offer the user thread-safe versions of equality and hashing methods. In its current form, it is up to the user to take care of concurrency.

Lastly, other methods can be generated using the same techniques. These are, for example, the `clone` method and the `toString` method. The latter is catered for in Eclipse [18]. Functionality responsible for serialising objects could also be automatically generated. Using code generation, serialisation is known to run faster [19].

**Acknowledgements.** The research work disclosed in this publication is partially funded by a Strategic Educational Pathways Scholarship (Malta). The scholarship is part-financed by the European Union – European Social Fund (ESF). B. Fischer is supported by EPSRC grant no. EP/F052669/1. We thank D. Rayside for making his system available for testing and Y. Smaragdakis for his comments on an earlier version of this paper.

## References

[1] Sun Microsystems Inc. *Java Platform Standard Ed. 6*.

[2] M. Vaziri, F. Tip, S. Fink, and J. Dolby. Declarative Object Identity Using Relation Types. In *ECOOP*, *LNCS* 4609, pp. 54–78. Springer, 2007.

[3] D. Rayside, Z. Benjamin, R. Singh, J. P. Near, A. Milicevic, and D. Jackson. Equality and hashing for (almost) free: Generating implementations from abstraction functions. In *ICSE*, pp. 342–352. IEEE, 2009.

[4] *JFreeChart*. http://www.jfree.org/jfreechart/.

[5] S. Khoshafian and G. P. Copeland. Object Identity. In *OOPSLA*, *SIGPLAN Notices* 21(11), pp. 406–416, 1986.

[6] P. Grogono and M. Sakkinen. Copying and Comparing: Problems and Solutions. In *ECOOP*, *LNCS* 1850, pp. 226–250. Springer, 2000.

[7] S. Abiteboul and J. Van Den Bussche. Deep equality revisited. *Deductive and Object-Oriented Databases*, *LNCS* 1013, pp. 213–228. Springer, 1995.

[8] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[9] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima, 2008.

[10] J. Bloch. *Effective Java (2nd Edition)*. Prentice Hall, 2008.

[11] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[12] J. Jones and R. Smith. Automated auditing of design principle adherence. In *Proc. ACM Southeast Regional Conference*, pp. 158–159. ACM, 2004.

[13] D. Zook, S. S. Huang, and Y. Smaragdakis. Generating AspectJ Programs with Meta-AspectJ. In *GPCE*, *LNCS* 3286, pp. 1–18. Springer, 2004.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP*, *LNCS* 2072, pp. 327–353. Springer, 2001.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[16] B. W. Kernighan and D. M. Ritchie. *The C Programming Language (2nd edition)*. Prentice Hall, 1988.

[17] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pp. 169–190. ACM, 2006.

[18] *Eclipse IDE*. http://www.eclipse.org/.

[19] B. Aktemur, J. Jones, S. N. Kamin, and L. Clausen. Optimizing Marshalling by Run-Time Program Generation. In *GPCE*, *LNCS* 3676, pp. 221–236. Springer, 2005.