Generating correct and efficient equality and hashing methods using JEqualityGen

Neville Grech[1,4,5] Julian Rathke[2,5] Bernd Fischer[3,6,5]

School of Electronics and Computer Science
University of Southampton
United Kingdom

Abstract

Determining equality of objects in Java requires the implementation of `equals` and `hashCode` methods. Such an implementation has to follow a specific object contract, making it a very tedious and error-prone process. Many equality and hashing methods implemented in Java applications violate this contract and are faulty, due to complexity arising from field shadowing, comparisons between different types, object cycles, etc. Equality and hashing implementations are conceptually simple, and can be derived automatically from information obtained statically from the program. However, existing systems that generate equality implementations use reflection and are consequently inefficient. Here we describe JEqualityGen, a source code generator that seamlessly integrates with the build process of typical Java projects: the programmer only needs to indicate for which classes implementations should be generated. JEqualityGen produces correct and efficient code which on a typical large-scale Java application exhibits a typical performance improvement of 162× in the equality operations generated by existing reflective systems. This was made possible through the use of optimised code generation and runtime profiling of these methods.

Keywords: meta-programming, equality, equals, hashCode, code generation, Java, AOP.

1 Introduction

Equality of objects in an object-oriented language is semantically defined as an equivalence relation, but often it does not have a straightforward implementation. The default equality operation in these languages merely compares references, however, it is common to have multiple objects in memory that are equivalent. In theory, two objects are equivalent if they are *indiscernible*. In practice, equivalence relations are specified through a supported interface in the object contract (such as the `equals` and `hashCode` methods in Java). Even though an informal equivalence

[1] Email: n.grech@ecs.soton.ac.uk
[2] Email: jr2@ecs.soton.ac.uk
[3] Email: b.fischer@ecs.soton.ac.uk
[4] Supported by a STEPS (Strategic Educational Pathways Scholarships) scholarship (ESF 1.25)
[5] This work was supported by EPSRC, grant EP/F052669/1. We thank Y. Smaragdakis for feedback and D. Rayside for making his system available for testing.
[6] This paper was written while visiting the University of Auckland

contract is specified [1], the implementation and verification of this contract is left to the user.

A number of studies [2] [3], together with our research, suggest that equality methods in many applications are buggy and violate the object contract. This should be quite worrying since equality bugs such as symmetry and transitivity violations tend to cause errors that are hard to track down. However, `equals` and `hashCode` methods are *conceptually* simple operations, and their implementations can be derived automatically. Rayside et al. [3] describe and evaluate a generic, reflective implementation of these methods. We developed a similar system, but one that actually generates code, instead of using reflection. This makes it possible to use static analysis and verification tools on the generated operations. It is far easier to verify, and to prove properties about generated code, than generic (and hence complex) code that makes use of reflection. Apart from facilitating further work on verifying correctness, this yields significant performance improvements: typically 162× for equality and 31× for hashing.

Throughout this paper we illustrate and tackle a number of challenges with equality implementations such as cyclic object structures, which our code generator overcomes. It is also designed with performance in mind, and profiles the equality methods in order to generate optimised equality operations. We demonstrate the effectiveness of code generation and our optimisations by running benchmarks on JEqualityGen (see http://sourceforge.net/projects/jequalitygen/) and related systems that make use of reflection. These benchmarks, together with other test cases, are based on a popular Java charting library, JFreeChart [15] .

2 Equality Definitions

2.1 Object identity

It is important to distinguish between object equality and object identity. The latter is tackled extensively in [4]. Two references to an object are said to be identical if they point to the same object. In Java, this is handled by the `==` operator.

A taxonomy of different object identity implementation strategies is also defined in [4]. 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. Identity using surrogates is the most powerful type of implementation [4]. Surrogates are also system-generated identifiers, however these are guaranteed to be unique to the whole system. Adding a surrogate to each object allows the identity implementation to use these identifiers rather than memory locations to compare identity, which is therefore unaffected if the object is moved in memory.

2.2 Types of equality

Khoshafian and Copeland [4] define different types of equality, namely shallow and deep equality while Grogono and Sakkinen [5] refine this concept. These types of equality naturally correspond to different types of copying such as deep and shallow

copying. *Reference equality* (or *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 copying, implies that for each field in a and b, reference 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 $k' < k$. If $k = \omega$, we also refer to this type of equality as *deep equality*. Objects a and b are thus deeply equal if all fields in a are deeply equal to the fields in b.

An elegant property of reference, shallow, depth-k and deep equality is that one 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.

2.3 The Java object contract

The Java object contract [1] explicitly states that the `equals` method implements an equivalence relation that has to be *reflexive*, *symmetric* and *transitive*. Equality has to be consistent, implying that it only takes into consideration the object's state and is side-effect free. Equality in [1] is only defined for non-null objects.

3 Equality implementation pitfalls

3.1 Comparing objects of different types

Vaziri et al. [2] claim that the Java object contract [1] cannot be adhered to in order to be able to compare two objects of different types. This is because one cannot have an implementation that is symmetric and transitive when two objects of different types might have a different interface as well as a different implementation.

It is, however, desirable to allow, for example, `TreeSet` and `HashSet` objects to be comparable since they can be interchanged while maintaining the same behaviour of the system. A pragmatic way around this limitation is to assign the same equality types to different classes. In practice, this presents a number of issues. For example, Hovenmeyer et al. [6] note that one common mistake is to have equality methods that return `true` even though the object types under consideration are incomparable. Another issue is that access to some `private` members is impeded.

3.1.1 Field shadowing when sub-classing

Java permits the overriding of fields throughout a class hierarchy. Unfortunately, this presents a number of challenges when implementing our equality methods. Such issues only happen when comparing objects of different types. For a simple example, consider a `Point` class containing two integer numbers and a `FancyPoint` class that extends `Point` and shadows its fields. We will initialise a `Point` object and a `FancyPoint` object and try to write an equality method that works for both. The naive implementation of equality in this situation is an `equals` method in the `Point` class that directly accesses both fields of the objects being compared.

```
public class Shadowing {
    public static void main(String[] args) {
```

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

Listing 1: Direct field access and inherited equality

However, Java dispatching mechanism does not use the dynamic type of an object when dispatching fields. In the `main` method, `p2.equals(p1)` is thus dispatched to `Point.equals` and instead of operating on the `FancyPoint` fields, we are operating on the `Point` fields, which are set to zero. `p2.equals(p1)` therefore evaluates to `false`, even though the two points are created with the same co-ordinates.

```
public class Shadowing {
    public static void main(String[] args) {
        Point p1=new Point();
        FancyPoint p2=new FancyPoint();
        p1.x=5; p1.y=5; p2.x=5; p2.y=5;
    }
}
class Point {
    public int x, y;
    public boolean equals(Object o) {
        Point that=(Point)o;
        return this.x==that.x && this.y==that.y;
    }
}
class FancyPoint extends Point {
    public int x, y; // shadows x and y in the Point class
    public boolean equals(Object o) {
        Point that=(Point)o;
        return this.x==that.x && this.y==that.y;
    }
}
```

Listing 2: Direct field access and overridden equality

In Listing 2, we override `equals` in `FancyPoint` in the hope of accessing the correct fields when making the comparisons. With this modification, even though `p2.equals(p1)` is `true`, `p1.equals(p2)` is `false` and therefore symmetry is violated. This happens because the `equals` method in `p2` is `FancyPoint.equals` and uses `FancyPoint.x` and `FancyPoint.y` while the `equals` method in `p1` is `Point.equals`, which only sees `Point.x` and `Point.y`.

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 accessor methods must be overridden together with all equality methods.

3.2 Symmetry, transitivity and reflexivity

It is common that `equals` implementations start with an instance check that short-circuits the entire operation. This tends to cause problems if we compare two objects whose respective types are subclasses of each other. Odersky et al. [7] note that

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

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

Listing 3: Correct implementation. Overriding equality methods and accessors

the instance check fails depending on whether `equals` is called on one object or the other, which violates symmetry. For example, a `FancyPoint` object is an instance of `Point` but a `Point` is not necessarily an instance of `FancyPoint`. A more "clever" `equals` design, as for example presented both in [7] and [8], dispatches on the type of its parameter and has different implementations for different types. Even though symmetry is not violated, transitivity is.

In order to ensure both symmetry and transitivity, Odersky et al. [7] 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 symmetrical if every class in the hierarchy defines this method. Its use can be seen in the generated code (cf. Listing 5).

A reference check at the beginning of an equality operation that can short-circuit the entire process not only enhances performance but also ensures reflexivity of the operation. This check can also be applied to individual fields.

3.3 Cyclic object graphs

A cyclic object graph can occur easily when objects are referencing each other. If the developer writing the `equals` or `hashCode` methods is not aware of this, an invocation to such methods would 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 this would make the equality method unfaithful to the abstract state of the original object [3].

It is however possible to write `equals` and `hashCode` methods that can deal with cycles. Rayside et al. [3] use an approach similar to the one in Eiffel [9] whereby two objects are assumed to be equal. Evidence is searched and checked to refute this assumption by traversing the cyclic object graph. If a cycle is encountered, no more evidence can be obtained by traversing the cycle multiple times. For `hashCode`, whenever a cycle is encountered, the object structure cycle's hash is substituted by a constant number.

3.4 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. A more serious consequence is that if an object is placed into a collection, the operations `add, remove` and `contains` will produce 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. This object cannot be retrieved since it resides in a different bucket that does not correspond to its new hash-code.

Countering this problem entails that equality 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.

3.5 Incorrect override

A number of authors [3] [2] [6] [8] [7] agree that a common mistake that can typically go undetected is that of specifying an `equals` method with an incorrect signature.

The `Object` class already defines a default `equals` method as one that computes object identity. This method could get called by a collection, and in case the method is not overridden, the default `Object.equals` method is used. This default implementation only performs a reference equality check. It is easy to imagine cases where calling a collection's `contains` method with an object returns `false` even though an identical object is in the collection.

3.6 The relationship between equals and hashCode

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 number of tools [6] [10] 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* [6].

Although not enforced by the compiler, the `Object` contract [1] specifies a clear relationship between the `equals` and `hashCode` methods. Mainly, if two objects are equal, they need to have the same hash-code. The converse need not apply. Similarly to `equals`, `hashCode` has to be side-effect free and consistent.

4 Overview of JEqualityGen

JEqualityGen is a code generator that automatically generates `equals` and `hashCode` methods. It follows the aspect oriented programming (AOP) paradigm, because object equality may be considered as a cross-cutting concern. In order to generate the equality implementations we needed a meta-programming language. Since we were working in Java, we chose Meta-AspectJ [11], a meta-programming extension for AspectJ [12]. It leverages the program transformation capabilities of AspectJ such as inter-type declarations. This enables the generated code to be statically woven into the existing Java bytecode.

JEqualityGen loads 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 weaved into the user's existing classes using the AspectJ compiler. Therefore all operations can be carried out at bytecode level. This makes it easier to integrate into the build process. JEqualityGen supports the following features:

(i) The developer annotates classes for which equality needs to be generated using JEqualityGen's annotations. A super-type may also be specified so that objects may be compared at that specific level. An example is shown in Listing 4.

(ii) The developer specifies which fields should *not* be considered as key fields.

(iii) Analysis is performed statically to see which keys might mutate throughout the execution of the application. Warnings and errors are issued accordingly. Various other checks are performed to ensure the correctness of the generated methods.

(iv) Implementations for `equals` and `hashCode` methods are generated automatically using the guidelines specified in Section 3. These are woven into the existing Java bytecode.

(v) In the case of cyclic object graphs, an approach similar to the one in Eiffel [9] is employed. To improve performance, the code concerned with cyclic graphs is selectively inserted. This is done only if there is the possibility of having such cycles.

(vi) Getters and setters are generated for each key field. Getters and setters are bypassed if equivalent behaviour is guaranteed. This optimisation is performed statically.

(vii) Optimisations based on dynamic run-time feedback are also performed. Fields that differ most often, thus determining non-equality most often, can be used to short circuit the equality operations, thus further increasing the efficiency of the generated methods.

(viii) Generating `equals` and `hashCode` for types that are not accessible to the weaver, as specified by the user.

Feature (viii) was added after we noticed that a number of classes in external libraries that are used in the user's code do not have a correct implementation of `equals`. In order to declare AspectJ inter-type declarations with updated equality and hashing implementations, it is necessary to have the library classes in the AspectJ `inpath`, something that is not always possible. For example, putting all JRE classes in the `inpath` would crash the weaver. Instead, we automatically sub-class these classes and add the actual equality logic in the sub-classes. All calls to the constructor of the original class are intercepted and an instance of the extended class is initialised instead. Any calls to the `equals` or `hashCode` methods are received by the automatically extended classes.

4.1 Generating equality and hashing logic

The `equals` methods follow a specific template. An example of a generated `equals` method is given in Listing 5. In order to compute equality, two methods are generated, `equals` and `canEqual`, which perform the following:

- Determine whether the object to which it is comparing itself to is an instance of the correct type.
- Coerce the object to the correct equality type.
- See whether the receiver object is comparable to the argument, and conversely see whether the argument is comparable to receiver.
- Profile the uniqueness of each key field by recording how much difference is observed in each field of different objects.
- Determine whether each key field in the receiver object is equal to each key field in the other object.

```
@Equality                              @Equality(class=Point)
class Point {                          class FancyPoint extends Point {
    public int x, y;                       public int x, y;
}                                      }
```

Listing 4: Annotated Point and FancyPoint classes with equality performed at the Point level.

```
private int test.Point.__get_x() { return this.x; }
private int test.FancyPoint.__get_x() { return this.x; }
private int test.Point.__get_y() { return this.y; }
private int test.FancyPoint.__get_y() { return this.y; }
public boolean test.Point.equals(Object other) {
    if (this == other) return true;
    if (other instanceof test.Point) {
        test.Point that = (test.Point)(other);
        return that.canEqual(this) && this.__get_x() == that.__get_x() &&
             this.__get_y() == that.__get_y();
    }
    return false;
}
public boolean test.Point.canEqual(Object other) {
    return other instanceof test.Point;
}
public boolean test.FancyPoint.equals(Object other) {
    if (this == other) return true;
    if (other instanceof test.Point) {
        test.Point that = (test.Point)(other);
        return that.canEqual(this) && this.__get_x() == that.__get_x() &&
         this.__get_y() == that.__get_y() && this.__get_x() == that.__get_x()
         && this.__get_y() == that.__get_y();
    }
    return false;
}
public boolean test.FancyPoint.canEqual(Object other) {
    return other instanceof test.Point;
}
public int test.Point.hashCode() {
    return this.__get_x() * 1 + this.__get_y() * 31;
}
 public int test.FancyPoint.hashCode() {
    return this.__get_x() * 1 + this.__get_y() * 31 +
           this.__get_x() * 961 + this.__get_y() * 29791;
}
```

Listing 5: Generated equality and hashing methods and accessors for the Point and FancyPoint classes in Listing 4

The actual logic which compares each field from the the receiver to each field in the argument is a conjoined equality expression. For this expression the code generation task is split into multiple parts. For each field, we see whether it is a primitive type or a simple reference type that requires only pointer equality (a class annotated with `@ReferenceEquality`). In such a case, 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 process depending 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. In the case of `hashCode` we use a similar approach; however, we generate a Kernighan and Ritchie's multiplicative hash expression [13] instead of an equality expression. The following listing shows part of a generated equality expression for various field types.

```
(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 expression for different types p2:Object, d1:double, f1:float, c1:char, s:1d array, s22:2d array

4.2 Handling Cycles

The code generated in Listing 5 does not take into consideration potential cycles in the object graph. 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. Given a particular container class and a containee (initially the same class), the algorithm 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 being tested is not a primitive type, the algorithm checks whether the containee may be contained in this field. This process is invoked recursively until all fields in the class hierarchy are tested.

Cycles are handled by generating advice that uses a point-cut descriptor as shown in Listing 7. The executing advice uses stacks containing the visited objects, one 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, the equality check whose path caused the cycle must be `true` since it 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 our target from the stack when the computation returns. We finally return the result of the equality computation.

```
...  execution(boolean equals(Object))
  && target(self)
  && cflowbelow(execution(boolean equals(Object))
  && target(org.jfree.chart.ChartRenderingInfo)) ...
```

Listing 7: Point-cut descriptor of the cycle handler advice

	JEqualityGen	JEqualityGen w/profiling optimisations	Rayside et al. [3]
`equals`	2297	1108	179856
`hashCode`	3602	2760	86683

Table 1
Time to run benchmark in ms

4.3 Other features

We implemented a performance optimisation by generating getters only if we have to. Fields that need getters are found by going up the class hierarchy to see which fields are declared at every level. If a field is found to be declared at more than one level, these fields are added to a result set. This data structure is later used to generate the appropriate getters. Whenever we refer to any field in a class that has the same name as the fields we have found, we do so using our custom getters.

In order to deter the mutation of key fields, we give warnings at compilation time indicating any locations where the mutation might occur after instantiation. This is possible thanks to AspectJ's static join-point model, which allows warnings to be issued if a point-cut descriptor can be matched statically.

Optimisations based on run-time feedback are done by generating code with data gathering statements. These simply compare each field in an object to the object being compared to. If two corresponding fields are different, then a "false" tally for that field is incremented, otherwise the "true" tally is incremented. The results are then used to sort the fields according to the tally ratio `false/true` and the generated equality expression is more likely to shortcut early on in the computation when invoked.

5 Experimental Evaluation

5.1 Performance analysis

Since we were making use of code generation over reflection, an expected increase in performance was expected. To assess our performance claims, we wrote a benchmark that exercises the `equals` and `hashCode` methods of a number of classes from the JFreeChart project [15]. The closest system we could find to JEqualityGen is the one presented by Rayside et al. [3]. Hence, benchmarks were run on both of these systems. It should be noted that [3] makes use of caching to enhance performance. The benchmark loop is run several times before starting the timer. This enabled both the JVM and the implementation of [3] 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 the solution in [3] 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.

5.2 Correctness analysis

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 would 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 would fail.

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.

6 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. Our prototypical implementation is expressive enough as a drop-in replacement in the context of large Java applications. It can 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 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 run-time systems would throw exceptions at runtime, which is much less convenient. A main motivation why we generate Java code is to verify the correctness of the generated code using formal methods. We plan to use these formal methods in our future work and any further effort on verifying correctness will be done using such methods. We therefore chose to devote less time on experimental testing and only tackle a subset of the JFreeChart test cases.

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 [14] fail to handle. This code generator is also naive in the sense that it does not concern itself with the object model, but rather with individual 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.

We did not try to 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 object's 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 [14]. Functionality responsible for serialising objects could also be automatically generated. Using code-generation, serialisation is known to run faster [16].

References

[1] Sun Microsystems Inc. *Java Platform Standard Ed. 6.* Available online at: http://java.sun.com/javase/6/docs/api/.

[2] M. Vaziri, F. Tip, S. Fink, and J. Dolby. Declarative object identity using relation types. *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] S. Khoshafian and G. P. Copeland. Object identity. In *OOPSLA*, pp. 406–416, 1986.

[5] P. Grogono and M. Sakkinen. Copying and comparing: Problems and solutions. In *ECOOP*, *LNCS 1850*, pp. 226–250. Springer, 2000.

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

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

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

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

[10] J. Jones and R. Smith. Automated auditing of design principle adherence. In S.-M. Yoo and L. H. Etzkorn, editors, *ACM Southeast Regional Conference*, pp. 158–159. ACM, 2004.

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

[12] 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.

[13] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, 1988.

[14] Eclipse IDE. http://www.eclipse.org/.

[15] JFreeChart. http://www.jfree.org/jfreechart/.

[16] 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.