An Empirical Study of the Design and Implementation of Object Equality in Java

Chandan R. Rupakheti and Daqing Hou

Clarkson University, Potsdam, NY 13699
{dhou,rupakhcr}@clarkson.edu

Abstract

Applications built on top of an existing design and implementation are in general expected to collaborate well with that design and respect all of its intent. Failure in achieving this may result in buggy, fragile, and less maintainable code in the applications. When the dependence on an existing design becomes more wide-spread, this requirement on proper extension obviously becomes even more critical. As an instance of this general problem, the design for object equality in Java as well as its extensions is examined in detail and empirically. By examining how object equality is extended in a large amount of Java code, a set of typical problems are detected and their root causes analyzed. A set of design guidelines for object equality is proposed, which, if followed, will help programers systematically design and evolve rather than hack on a solution. Examples are drawn from a case study of multiple industrial and open source projects to illustrate the identified problems and how the proposed guidelines can help solve these problems.

1 Introduction

It is common for object-oriented programming languages like Java and C# to provide a useful set of collection data types like set, map, vector, and hash table. [1] [2] In order to work as elements inside such a container data structure, application objects need to support an equality predicate with which a pair of objects can be compared. The design adopted by both Java and C# is to specify a contract for equality in the Object class, which is expected to be supported by all other application classes. In this way, the collection data types can be implemented with the assumption that the element objects will honor this equality contract.

In particular, in Java, the equals(Object) method in the java.lang.Object class is designed to support the work of collection types. As its behavioral specification, the equals method is required to implement an *equivalence relation* on any two *non-null* object references x and y, supporting the following properties:

1. *Reflexivity*: x.equals(x) always returns true.
2. *Symmetry*: x.equals(y) returns true if and only if y.equals(x) returns true.
3. *Transitivity*: for any non-null reference z, if x.equals(y) and y.equals(z) return true, then x.equals(z) should return true.
4. *Consistency*: multiple invocations of x.equals(y) consistently return true or false, provided that no information used in comparing the objects is modified.
5. *non-nullity* [3]: x.equals(null) must return false.

Copyright © 2008 Daqing Hou and Chandan R. Rupakheti. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

[1]JDK 1.5. http://java.sun.com/j2se/1.5.0/docs. Verified April 10, 2008.

[2]C# Language Specification. http://msdn.microsoft.com/en-us/library/aa645596(VS.71).aspx.

[3]The term non-nullity is due to Bloch [1].

```
public class Point3D extends Point2D {
   private int z; // the z coordinate
   public boolean equals(Object o) {
       if ((o instanceof Point3D)) {
         return super.equals(o) &&
            (Point3D)o.z==this.z;
       }
       else return super.equals(o);
   }
}
```

Figure 1: An implementation of Point3D and equals() that violates transitivity (pp. 182 of [2], with modification of class names).

All objects that become an element of a collection from the *Java Collection Framework* [4] must obey this contract in order to function properly inside the collection.

This issue of object equality is a general design problem present in both Java and C#. It is also a long-standing problem that can be traced back to the Lisp community [3]. It has a wide impact on many classes. For example, 624 classes in JDK 1.5 implement an equals() method, covering areas such as security, corba implementation, utility classes, collection types, GUI, and so on. However, implementing equals() to respect this contract needs several considerations that can be easily overlooked by a developer, resulting in buggy or fragile code. Such defects are notoriously difficult to find, even for experienced programmers. And to make matters worse, textbooks often present flawed versions of these critical operations (e.g., the example shown in Figure 1) or give unsound advices. The understanding of the problem is so vague that even popular Java IDEs like Eclipse 3.3 [5] and NetBeans 6.0 [6] have bugs in the equals() method generated through their wizard. It is also the center of much controversy. For example, type-incompatible equality [7] between a supertype and a subtype would violate the well-known Liskov Substitution Principle [2, 4]. Should type-incompatible equality be therefore prohibited? Furthermore, should mutable types be allowed to define equals()?

In this paper, we develop a set of design guidelines for implementing equals(). First, an analysis of the equality design is performed in Section 2. Three kinds of equality are distinguished and their corresponding implementations described. Findings from a case study of 4 real-life projects are presented in Section 3. The case study not only helps validate the design guidelines developed in Section 2, but also provides additional input to our guidelines with regards to implementation choices. The whole set of guidelines is summarized in Section 4. [8] Finally, Section 5 presents related work and Section 6 concludes the paper.

2 Design Intent and Implementation Patterns for Equality

In this section, the equality relation is analyzed and three kinds of equality (type-compatible, type-incompatible, and hybrid equality) are introduced. Their relation with a type hierarchy and how they can be designed and implemented properly in a type hierarchy is described. In particular, the relation between type-incompatible equality and subtyping is discussed.

2.1 Type, state, and equality relation

A type is defined by a value space (a.k.a state) of named properties as well as operations on the value space. For example, an object of type Point2D may have two properties, the x and y cartesian coordinates, and setters and getters for these properties. Two types may be related by subtyping, which is governed by the Liskov Substitution Principle (LSP hereafter) such that an object of a subtype can be substituted for an object of a supertype in all contexts where a supertype object is expected [2, 4]. For example, if it is intended and legitimate for a Point3D object to appear in all of the contexts where a Point2D is used, then Point3D can be made a subtype of Point2D. Note that the appropriateness of subtyping is often determined by domain semantics and contexts of use. Finally, each object has a reference (address) that uniquely identifies the object, and each object has one or more types.

Given a set of objects, multiple equivalence relations can be defined. In theory, given a set of objects that share n properties, 2^n equivalence relations can be defined in terms of these properties.

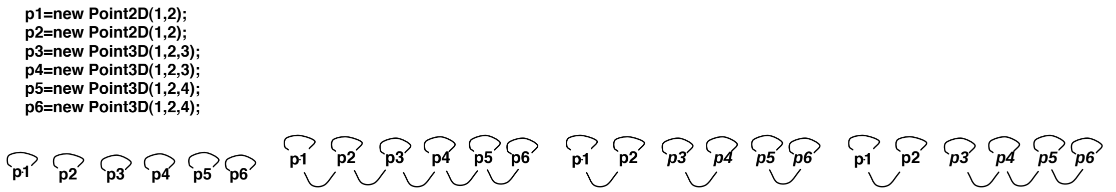
[4]http://java.sun.com/docs/books/tutorial/collections/index.html. Verified April 10, 2008.

[5]http://www.eclipse.org/eclipse

[6]http://www.netbeans.org

[7]Type-incompatible equality is defined in Section 2.

[8]Note that the complete set of guidelines is derived from both the *analysis* in Section 2 and the *empirical findings* presented in Section 3.


```
p1=new Point2D(1,2);
p2=new Point2D(1,2);
p3=new Point3D(1,2,3);
p4=new Point3D(1,2,3);
p5=new Point3D(1,2,4);
p6=new Point3D(1,2,4);
```

(a) Reference equality (b) Type-compatible equality defined in terms of x, y, or both (c) Type-incompatible (d) Type-incompatible equality

Figure 2: Examples of equality relations for Point2D and Point3D. For Point3D, the equality in 2c is defined in terms of x, y, and z, and the one in 2d is defined in terms of x and y. The links between two nodes represent equality. Links that can be inferred via transitivity are omitted for clarity.

Figure 2 shows several examples of equality defined for objects of Point2D and Point3D.

There are two special cases of equality. One is the *reference equality* [9] where an object is uniquely identified by its reference, and thus can be equal only to itself. Therefore, reference equality is the most discriminating equality relation. The default implementation in Object.equals() is a reference equality. When reference equality is too discriminating to be useful, the default implementation can be overridden by another notion of equality based on domain semantics. The other special case of equality, which we call *full equality*, takes into account the full value space of objects. *Full equality* is a special case of domain-semantics-based equality and is suitable for testing whether two objects are fully behaviorally equivalent rather than just partially similar.

2.2 Type-compatible and type-incompatible equality

As the first step in designing an equality relation, it is always helpful to distinguish between the following two intents with respect to a notion of compatibility between object types. *Type-compatible equality* is determined by comparing a subset of common properties between two types S and T, which may or may not be in the same type hierarchy. *Type-incompatible equality* requires that only objects from the same type can be possibly considered equal by comparing a subset of the properties between the objects. Under this notion of equality, two objects of two different types will never be equal, and thus the two types involved are considered *incompatible*.

```
1  public class Point2D {
2  ... defining representation & operations
3  public boolean equals(Object o) {
4          if (!(o instanceof Point2D))
               return false;
5          Point2D that = (Point2D)o;
6          return this.getX() == that.getX() &&
7                 this.getY() == that.getY();
8  }
9 }
```

Figure 3: Implementing type-compatible equality with instanceof.

```
1  public class Point2D {
2  ... defining representation & operations
3  public boolean equals(Object o) {
4     if (o==null) return false;
5     try { Point2D that = (Point2D)o;
6         return this.getX()==that.getX()&&
7                this.getY() == that.getY();
8     } catch (ClassCastException cce){
9             return false;
10    }
11   }
12 }
```

Figure 4: Implementing type-compatible equality with exception handling rather than instanceof.

If two types involved in a type-compatible equality form a type hierarchy, the type hierarchy can help reduce the number of equals needed to be implemented due to inheritance and dynamic dispatching [2]. Figure 3 illustrates how an equality between Point2D (superclass) and Point3D (subclass) can be implemented by the equals() in class Point2D and inherited by Point3D. The type testing at line 4 checks whether o can be type-cast to Point2D. o instanceof Point2D will return true if o's type is either Point2D or one of its subtypes, and it will return false when o is null. Line 5 does the type cast, and lines 6 and 7 compare the state.

The instanceof type testing can also be done with the exception handling mechanism. Figure 4 shows an implementation that is functionally equivalent to

[9]Bloch uses identity equality for what we call reference equality. [1]

```
1  public class Point2D {
2  ... defining representation & operations
3  public boolean equals(Object o) {
4     if (o == null) return false;
5     if (!o.getClass().equals(this.getClass())
          return false;
6     Point2D that = (Point2D)o;
7     return this.getX() == that.getX() &&
8            this.getY() == that.getY();
9   }
10 }
11  public class Point3D extends Point2D{
12  ... defining representation & operations
13  public boolean equals(Object o) {
14     if (o == null) return false;
15     if (!o.getClass().equals(this.getClass())
           return false;
16     Point3D that = (Point3D)o;
17     return this.getX() == that.getX() &&
18            this.getY() == that.getY() &&
19            this.getZ()==that.getZ();
20  }
    }
```

Figure 5: Implementing type-incompatible equality with getClass().

the one in Figure 3. While using exception handling may result in a small benefit in performance, it is not a common way of implementing equals(). It also increases the number of implementation patterns a developer has to master and understand.

Figure 5 shows an implementation for a type-incompatible equality between Point2D and Poin3D. This implementation uses the Java reflection API getClass(), which returns the runtime type of the receiver object. Therefore, the tests on lines 5 and 15 will permit only objects of the same type to pass. The null tests at lines 4 and 14 address the *non-nullity* property and ensure the subsequent calls to getClass() will not throw null pointer exceptions. When the equality is defined in terms of the same set of properties shared by two types, a subclass may inherit the equals() from the superclass rather than implement its own. This is impossible for the example in Figure 5 because the equals() in Point3D adds the z dimension.

When two types involved in a type-incompatible equality are intended to form a subtyping relation but the two equals() are implemented in terms of their respective type and state, the pair of equals() will be incompatible under LSP and will cause the two types to not form a proper subtyping relation. (That is, the specification of Point2D.equals() states that Point2D can only be equal to another Point2D that has identical x and y, and Point3D.equals() states that Point3D can only be equal to another Point3D with identical x, y, and z.) In this case, the equals() should be excluded from the type specifications, and the rest of the operations in the two types can still conform to LSP. It can then be required that such equals() must not be used in contexts where objects of subtype are substituted for those of supertype. But these two equals() do conform to LSP with Object.equals() because the specification of Object.equals() is weaker than the afore-mentioned ones with respect to how equality is exactly defined.

2.3 Hybrid equality

Sometimes, it can be useful or even necessary to define an equality that mixes type-compatible and type-incompatible equality in the same hierarchy (*hybrid equality*). When the main hierarchy is type-incompatible and a sub-hierarchy is type-compatible, they can be implemented respectively using the techniques introduced above. However, when a type-compatible hierarchy contains a type-incompatible sub-hierarchy, a new implementation pattern is needed. In what follows, it is first shown that it is impossible to use the techniques introduced so far to implement this kind of hybrid equality in a way that satisfies the equals contract. An implementation for hybrid equality based on the template method pattern [5] is then introduced.

```
public boolean equals(Object o) {
    if (!(o instanceof ColorPoint)) return false;
        ColorPoint that = (ColorPoint)o;
        return this.getX()==that.getX() &&
            this.getY()==that.getY() &&
            this.getColor() == that.getColor();
}
```

Figure 6: Color sensitive equals for ColorPoint.

Suppose a type-compatible equality has been implemented for Point2D and Point3D as shown in Figure 3. Now a new subclass ColorPoint needs to be added to Point2D, and suppose ColorPoint needs to implement a type-incompatible equality with regards to Point2D and Point3D. Figure 6 shows an implementation that attempts to provide type-incompatible equality at the ColorPoint side, but loses the symmetry property. The loss of symmetry can be demonstrated by the following code snippet:

```
Point2D p1 = new Point2D(1,2);
ColorPoint p2 = new ColorPoint(1,2,Color.RED);
p1.equals(p2); // returns true
p2.equals(p1); // returns false (broken symmetry)
```

An attempt to fix the symmetry violation but at the expense of the transitivity property is shown in Figure 7. The loss of transitivity can be illustrated by the following code snippet:

```
public boolean equals(Object o) {
    if (!(o instanceof Point2D)) return false;
    // For Point2D, ignore color in comparison
    if (!(o instanceof ColorPoint))
        return o.equals(this);
    // o is a ColorPoint; compare color as well
    ColorPoint that = (ColorPoint)o;
    return this.getX()==that.getX() &&
            this.getY()==that.getY() &&
            this.getColor() == that.getColor();
}
```

Figure 7: Color insensitive equals for ColorPoint.

```
ColorPoint p1 = new ColorPoint(1,2,Color.RED);
Point2D p2 = new Point(1,2);
ColorPoint p3 = new ColorPoint(1,2,Color.BLUE);
p1.equals(p2); // returns true
p2.equals(p3); // returns true
p1.equals(p3); // false (broken transitivity)
```

The root cause of this problem on transitivity is that the two equality comparisons that ColorPoint participates in make use of different properties. When compared with a Point2D, color is ignored. When compared with a ColorPoint, color is included.

The loss of symmetry in the implementation of Figure 6 is caused by the fact that the equals() of Point2D fails to include the color property of ColorPoint. When the argument is a ColorPoint, this implementation will compare only the x and y coordinates but not the color, resulting a violation of symmetry. This violation can be fixed by giving the argument an opportunity to add its own properties in addition to those of the superclass.

Figure 8 shows how Point2D and ColorPoint can be modified to achieve this. Notice that a new method equalsDelegate() is introduced and called by equals() in Point2D. ColorPoint overrides equalsDelegate() to add its type-specific comparison. In this case, ColorPoint enforces the requirement that the argument must also be a ColorPoint. It can be verified that the new implementation satisfies the equals contract. Also notice that the equals() method is declared final so that no subtypes of Point2D can override it. An incompatible subtype such as ColorPoint needs to override the equalsDelegate() method to perform type-specific comparison. Compatible subtypes will simply inherit both equals() and equalsDelegate().

By calling equalsDelegate() twice (that.equalsDelegate(this) && equalsDelegate(o)), the equals() of Point2D in Figure 8 essentially provides a bi-directional check for the two types involved in a comparison. As discussed above, the first check (that.equalsDelegate(this) is to give a subtype an opportunity to do a subtype-specific

```
public class Point2D {
  public final boolean equals(Object o) {
    if (!(o instanceof Point2D)) return false;
    Point2D that = (Point2D)o;
    return comparing x and y &&
    // For symmetry not provided by instanceof
    that.equalsDelegate(this)&&equalsDelegate(o);
    }
 // type-specific comparison. true by default.
 protected boolean equalsDelegate(Object o) {
    return true;  }
 // Remainder omitted
}

public class ColorPoint extends Point2D {
  protected boolean equalsDelegate(Object o) {
    if (!(o instanceof ColorPoint)) return false;
    ColorPoint that = (ColorPoint)o;
    // return the comparison of x, y, and color;
    }
 // Remainder omitted
}
```

Figure 8: Implementation of hybrid equality.

comparison. The purpose of the second check becomes evident when considering the evaluation of aColorPoint.equals(aPoint2D), where equalsDelegate(o) will cause equalsDelegate() of ColorPoint to be called, which will correctly return false.

It can be verified that all the implementations (Figures 3, 4, 5, and, 8) satisfy the equals contract introduced in Section 1.

2.4 Discussion

```
public class Point2D {
... defining representation & operations
public boolean equals(Object o) {
   if (o == null) return false;
   if (o.getClass().equals(Poin2D.class){
   Point2D that = (Point2D)o;
   return this.getX() == that.getX() &&
            this.getY() == that.getY();
    }
    if (o.getClass().equals(Poin3D.class){
    Point3D that = (Point3D)o;
    return this.getX() == that.getX() &&
             this.getY() == that.getY();
     }
  return false; }
}
public class Point3D {
... defining representation & operations
public boolean equals(Object o) {
   if (o == null) return false;
   if (o.getClass().equals(Poin2D.class){
     // return result of comparison
    }
   if (o.getClass().equals(Poin3D.class){
   Point3D that = (Point3D)o;
    // return result of comparison
   }
   return false;}
}
```

Figure 9: Implementing type-compatible equality between types not in the same hierarchy.

It is even possible to define an equivalence relation where two objects from two different type hierarchies are considered equal. Such equality may be useful in scenarios where both kinds of objects are used as a key to a hash table. Two equals() need to be implemented, one for each type. Figure 9 shows how this can be done under the assumption that Point2D and Point3D be independent classes. However, in the projects we studied, no instances of equality were found being done in this way. This is probably because when two types share properties, a type hierarchy can usually be designed to relate them. In this paper, we focus on the case where any two objects that are to be considered equivalent also belong to the same type hierarchy and on how to design and implement equals() in a type hierarchy such that the equals contract is respected.

Equality among objects from different types must be compared on the basis of the same set of properties. The implementation in Figure 9 satisfies this requirement, but that of Figure 1 does not.

Java's design for equality can be restrictive. When an object needs to participate in more than one hash table as a key under distinct notions of equality, distinct equality tests are needed to work with each hash table. But the object can define only one equality test (either via the equals() method in its class or by inheriting one). One way to work around this limitation is to design a wrapper class with a pair of equals() and hashCode() methods for each notion of equality and use as keys objects of the wrapper class rather than the original objects. Another way in which Java's design may become restrictive is when an equality different from the one designated for hash table is needed for comparing two objects directly. For example, a full equality may be needed to compare two objects of the same type, but a weaker, type-compatible notion of equality is provided for the objects to participate in the hash table, which is not suitable for the purpose of comparing objects. When this kind of clashes happen, a method with a different name that reflects the right intent, like similar() or identical(), must be added to the object's class instead of attempting the impossible task of overloading equals() for multiple purposes.

3 Case Study

To understand how equals() are implemented and what kinds of problem actually occur in real-life code, and to validate the (potential) usefulness of the guidelines proposed in Section 2 in dealing with these problems, we performed an empirical study of equals implementation in 4 Java projects of various sizes and domains. The study was conducted semi-automatically with a combination of both tool support and manual inspection. Several static checkers were developed using Eclipse's Java code analysis API [10] to search for code patterns that potentially violate the equals contract. The output from the checkers was examined by both authors to make a conclusion about their nature. While still in preliminary stage, the checkers were helpful in quickly processing large amounts of code and directing our attention to cases that are more likely to have problems. However, the design of the checkers is not the focus of this paper.

	JDK1.5	**Lucene**	**BCEL**	**SCL**
KLOC	2 552	88	39	22.6
#packages	571	14	8	29
#classes	12400	752	333	386
#interfaces	1743	26	35	11
#equals	624	40	20	15

Table 1: Project summary.

Table 1 provides some overall measures of the 4 projects, JDK 1.5, Apache Lucene [11], BCEL [12], and SCL [6]. Apache Lucene is a full text search engine, BCEL is a Byte Code Engineering Library, and SCL is a static analysis tool. JDK 1.5 is the largest project (2552 KLOC, 12400 classes) and SCL (22.6 KLOC, 386 classes) is the smallest in the group. These projects make use of the collection framework and rely on the correct implementation of equals() to function properly.

We find that JDK 1.5 is the most representative among the 4 projects in terms of the diverse equals() related issues that are exhibited. We have seen a total of 174 suspicious implementations of equals(), which cover a wide variety of areas such as collections, utility classes, security, object broker protocol, component model, network management, compiler, GUI and image processing, and naming services. In this section, we discuss the nature, possible cause, and possible solution to the problems using JDK1.5 as a primary source of examples. We provide both class names and package names so that interested readers can verify with the

[10]http://www.eclipse.org/jdt. Verified April 10, 2008.
[11]http://lucene.apache.org/java/docs
[12]http://jakarta.apache.org/bcel

Problem	Summary	Section
inheritance for impl. reuse	5 hierarchies	3.1
equals() for other purposes	9 classes	3.2
type-incomp. eq.	10 hierarchies	3.3
type-comp. eq.	several impl. of Map.Entry	3.4
hybrid eq.	Map and List hierarchies	3.5
evolution	Rectangle2D and Point2D	3.6
super.equals()	98	3.7
type casting	30	3.8
null checking	26	3.8

Table 2: Summary of inspected equals()-related problems in JDK 1.5.

JDK source code themselves. Table 2 provides a summary of the problems in JDK 1.5 that have been inspected by both authors as well as the sections where further details can be found.

3.1 Using inheritance for implementation reuse

When a class hierarchy is used for implementation reuse instead of subtyping [4], some problems with equals() may be detected. The benefit of implementation reuse is, of course, that one does not have to rewrite the similar code again. But using inheritance for implementation reuse overloads the same mechanism with two purposes. When there is not documentation of intent, it can be hard to tell which one, subtyping or reuse, is intended just from the code. It becomes even worse when one part in a hierarchy is used for subtyping and another for implementation reuse. The DefaultCaret and its superclass Rectangle provide such an example.

The Rectangle class (java.awt) and its superclasses, RectangularShape and Rectangle2D (java.awt.geom), form a type hierarchy, and implement a type-compatible equality. The relation among them is subtyping, and a Rectangle can behave like a Rectangle2D and a RectangularShape. However, when the DefaultCaret class in the javax.swing.text package is added as a subclass of Rectangle, the symmetry is lost in the equality implementation because DefaultCaret implements the reference equality as follows:

```
/**
 * Compares this object to the specified object.
 * The superclass behavior of comparing
 * rectangles is not desired, so this is changed
 * to the Object behavior.
 * ... (Remainder documentation omitted)
 */
public boolean equals(Object obj) {
      return (this == obj);
}
```

As can be seen in the Javadoc above, clearly, the designer of DefaultCaret is overriding the equals() with a clear intention. However, this equals() violates the symmetry property as follows:

```
Rectangle r = new Rectangle();
DefaultCaret c = new DefaultCaret();
r.equals(c); // returns true
c.equals(r); // returns false
```

The root cause of this problem is using inheritance as implementation reuse. DefaultCaret, as its name suggests, implements a caret in a text box. When a caret is moved to a new location, the area (a.k.a. bound box) where it was displayed last time needs to be tracked and repainted. A bounding box happens to be a rectangle and DefaultCaret was made a subclass of Rectangle so that it can inherit and use four instance variables x, y, width, and height. In addition, it appears that these variables are intended to be private to DefaultCaret. However, Rectangle contains methods that can change them (e.g., setRect(x,y,w,h)). Therefore, in this case, the benefit of reuse seems to be minor in comparison with the cost of documenting and ensuring that these inherited methods must not be applied to DefaultCaret. A better design would be for DefaultCaret to compose rather than inherit Rectangle.

There are several well-known examples of (improper) implementation reuse in JDK, including Vector/Stack and Date/Timestamp. Our checker did not report equals-related problem for Vector/Stack because they belong to the AbstractList hierarchy, which implements a type-compatible equality. The onus is on the developer to ensure that a vector and a stack are never compared for equality, and mutators unique to a vector must not be applied to a stack. Our checker does detect a symmetry problem between Date and Timestamp (which extends Date and adds a nanosecond field). The author of Timestamp documents the intention of reuse implementation and cautions that Timestamp should not be substituted for Date.

In addition to these well-known examples, our checker also detects some new instances of implementation reuse from JDK 1.5. A class MirrorImpl is made the root of a large inheritance hierarchy in Java Debugging API (com.sun.tools.jdi) just so that all the subclasses can have a field to represent the Java virtual machine they are interacting with. A subclass BuddhistCalendar is derived from GregorianCalendar, but it appears not suitable to substitute BuddhistCalendar for GregorianCalendar. However, this is not documented. In contrast, BakedArrayList (sun.swing) is intended for reusing the implementation of ArrayList and the author clearly documented this intention in the code.

Although it may be possible to infer only from code whether inheritance is used for implementation reuse or subtyping, the inference is not always straightforward. For example, by inspecting the code, we conclude that DefaultCaret is not used as a Rectangle. But the process of drawing this conclusion is costly, and every future maintainer would need to repeat the same reasoning to ensure that DefaultCaret is not used as a Rectangle. The reasoning is not local and needs to be enforced whenever code changes. Thus, making DefaultCaret a subclass of Rectangle is not a good idea. If implementation reuse is at all justified, at least consider explicitly documenting the intent. In general, it appears that this documentation practice is performed better in the public API of JDK than its private part.

```
public boolean equals(Object o) {
    if (o == this) { return true; }
    else if (o instanceof IdentityHashMap) {
        IdentityHashMap m=(IdentityHashMap) o;
        // for each pair of (key, value) in m
        // test this.containsMapping(key,value)
    } else if (o instanceof Map) {
        // use value-based comparison
    } else { return false; }
}

// returns true if a pair p exists in this map
// such that p.key==key && p.value==value
private boolean containsMapping(Object key,
        Object value) {
        ...
    }
}
```

Figure 10: equals() of IdentityHashMap, which compares key-value pairs with == rather than equals() (with modificatons).

Map is an important type hierarchy in the Java Collection Framework, with concrete implementations such as HashMap and TreeMap that differ in implementation strategy, e.g., a hash table based map versus a tree-based map. In the Map interface, it is specified that the equals() method returns true if the given object is also a map and the two maps represent the same mappings. More formally, two maps t1 and t2 represent the same mappings if *t1.entrySet().equals(t2.entrySet())*. This ensures that the equals() method works properly across different implementations of the Map interface. As a result, a type-compatible equality is implemented in the abstract class AbstractMap, which is inherited by other maps.

A new class IdentityHashMap added in Java 1.4, however, violates the symmetry property of the equals contract, as shown in the following snippet:

```
Map hMap = new HashMap();
Map ihMap = new IdentityHashMap();
ihMap.put("1", new Integer(1));
hMap.put("1", new Integer(1));
hMap.equals(ihMap); // returns true
ihMap.equals(hMap); // returns false
```

This loss of symmetry is caused by a change in the behavior of the equals() in IdentityHashMap, which compares keys and values with reference equality (==) rather than value equality. Thus, hMap.equals(ihMap) returns true because hMap is a HashMap and uses value equality, but ihMap.equals(hMap); returns false because IdentifyHashMap uses reference equality to compare pairs between itself and another map. Figure 10 depicts the details of equals() in IdentityHashMap.

At the time of developing IdentityHashMap, clearly this violation was noticed and was treated as an exception, which is evident by the following comment highlighted in bolded text in the IdentityHashMap specification:
This class is not a general-purpose Map implementation! While this class implements the Map interface, it intentionally violates Map's general contract, which mandates the use of the equals() method when comparing objects. This class is designed for use only in the rare cases wherein reference-equality semantics are required.

There can be two possible fixes for this problem, both of which are easy to implement. The first solution is to change the Map hierarchy from type-compatible equality to hybrid equality in the way illustrated in Figure 8. This would involve modifying the equals() in AbstractMap (not shown) and adding equalsDelegate to IdentifyHashMap, which is shown in Figure 11. The second solution would be to move IdentityHashMap out of Map to create an independent hierarchy. This can be done by copy-and-pasting Map to create a new interface IdentityMap, and AbstractMap to create a new class AbstractIdenityMap. IdentityHashMap should be changed to inherit from AbstractIdentityMap rather than AbstractMap. The entrySet() of

```
protected boolean equalsDelegate(Object o) {
    if (o.getClass().equals(getClass()){
        IdentityHashMap m=(IdentityHashMap) o;
        // for each pair of (key, value) in m
        // test this.containsMapping(key,value)
    }
    return false;
}

// returns true if a pair p exists in this map
// such that p.key==key && p.value==value
private boolean containsMapping(Object key,
     Object value) {
       ...
    }
}
```

Figure 11: equalsDelegate() for IdentityHashMap as part of implementing a hybrid equality for the Map hierarchy.

IdentityHashMap and the equals() of AbstractIdentityMap also need to be modified. We have implemented a new IdentityHashMap in this way in a few hours. However, a potential problem with the second solution is that it separates IdentityHashMap from the Map abstraction, and thus makes it impossible for an IdentityHashMap to participate in client code written in terms of Map. If this is proven undesirable, the first solution could still be used instead.

3.2 Overloading equals with multiple purposes

The equals() contract as defined in java.lang.Object can support only one specific notion of equality. Sometimes, a class may need to support additional equality or some kind of *similarity* that is not intended to be used by a client such as a collection data type and even does not have to be an equivalence relation. Intentionally or incidentally, developers tend to overload the equals() implementation to encode all of them in one place. For ease of understanding and maintenance, it is advisable to implement separate predicates for such relations.

For example, String and StringBuffer classes (java.lang) implement the CharSequence interface and represent a sequence of characters. The difference is that String is immutable and StringBuffer is mutable. The equals() in String will always return false when compared with a StringBuffer. Since objects from these classes contain character sequences, it makes sense to test if the content of a String object is the same as that of a StringBuffer. Instead of piggybacking onto equals(), String class provides a "contentEquals(StringBuffer)" method for this purpose, which is not symmetric with respect to StringBuffer.

The Arg class represents an argument on the stack (com.sun.org.apache.xpath.internal). Among other fields, an Arg has a qualified name QName (com.sun.org.apache.xml.internal.utils).

```
public class Arg {
 private QName m_qname;
 public boolean equals(Object obj) {
  if (obj instanceof Qname)
   {return m_qname.equals(obj);}
  else return super.equals(obj);
 }
 ... // Remainder omitted
}
```

The equals() in Arg directly checks for the type of obj (obj instanceof QName) and compares it with m_qname. Note that QName does not belong to the type hierarchy of Arg. Furthermore, if obj is not of type QName (including objects of Arg class), equals() will perform a reference equality check. Though the equals() method of Arg checks for QName, the equals() in QName does not check for Arg, which means a loss of symmetry between the two classes. Thus, the equals() definition is probably intended to serve as a similarity check, and a better solution would be to remove the equals() method from Arg (in which case it will inherit Object's equals()) and add a new method as follows:

```
public boolean hasName(QName qName) {
  return m_qname.equals(qName);
}
```

There are several ways to detect such cases of piggybacking on equals(). A particularly useful means is to detect the presence of multiple instanceof type testing. Our experience is that most equals() implementations contain only one instanceof. Therefore, when an equals() contains more than one instanceof testing, it is more likely that the equals() predicate is overloaded to carry other notion of 'similarity'. A checker was written to detect the presence of multiple instanceof and 15 such cases were found in JDK 1.5. We inspected all of them and concluded that 9 of them are true positives and 6 false positives.

The AlgorithmId class in sun.security.x509 represents algorithms such as cryptographic transformations. Its equals(), as shown below, clearly overloads itself to carry both an equality check and a similarity check with ObjectIdentifier in sun.security.util.

A similar case occurs between the Oid class in org.ietf.jgss and ObjectIdentifier.

```
public boolean equals(Object other) {
if (this == other) {
    return true; }
if (other instanceof AlgorithmId) {
    return equals((AlgorithmId) other);
} else if (other instanceof ObjectIdentifier) {
    return equals((ObjectIdentifier) other);
} else {
    return false;}
}
```

Two classes GroupImpl and NetmaskImpl (com.sun.jmx.snmp.IPAcl) in the java.security.Principal hierarchy mistakenly encode a partial order similarity between 2 subnet masks in the equals() predicates (255.255.255.0 is 'equal' to 255.255.0.0 but not vice versa).

The XObject hierarchy in com.sun.org.apache.xpath.internal.objects shows that the developer is not clear about how to implement a sophisticated equality, under which, for example, a string (XString) that represents a number would be considered equal to another object that represents a true number. The strategy appears to be for XString.equals() to invoke equals(Object) from 2 other classes. Because only XString overrides equals(Object), and all the other classes in the hierarchy implement equals(XObject) instead, we conclude that the overriding of equals() in XString is incidental and that it is not intended to conform to the contract of java.lang.Object.equals().

3.3 Suspicious implementations of type-incompatible equality

The most common pattern found in JDK is probably implementing a type-incompatible equality by using an instanceof test in both a supertype and a subtype. In the example shown in Figure 12, NTSid represents a Security Identifier for Windows NT OS, which has 4 concrete subclasses that model a user, a domain, a group, and a primary group, respectively. These are principals that can be attached to a subject such as a person to grant the subject a certain permission.

Because NTSid and its subclasses are logically distinct objects, a type-incompatible equality should be implemented for this type hierarchy. Unfortunately, the solution presented in Figure 12 is suspicious as it breaks the symmetry property between NTSid and its subclasses. Fortunately, this can be fixed with the type-incompatible implementation shown in Figure 5. Another possibility is that NTSid is not intended to be instantiated. If that is the case, then at least it could have been made

```
// implementation of NTSid
public boolean equals(Object o) {
   if (o == null) return false;
   if (this == o) return true;

   if (!(o instanceof NTSid))
           return false;
   NTSid that = (NTSid)o;

   if (sid.equals(that.sid)) {
        return true;
   }
   return false;
}

// implementation of NTSidUserPrincinpal
public boolean equals(Object o) {
     if (o == null)       return false;
     if (this == o)       return true;

     if (!(o instanceof NTSidUserPrincipal))
         return false;

     return super.equals(o);
}
```

Figure 12: A common mistake of using instanceof to implement type-incompatible equality in a supertype (NTSid) and a subtype (NTSidUserPrincinpal).

an abstract class. Furthermore, its equals() can be removed to make it clear that a type-incompatible equality is intended for this hierarchy.

package	root class
com.sun.java_cup.internal	lr_item_core
com.sun.jndi.ldap	ClientId
com.sun.security.auth	NTSid
java.beans	PropertyDescriptor
java.security	CodeSource
javax.management	MBeanFeatureInfo
javax.management	MBeanInfo
javax.imageio	ImageTypeSpecifier
com.sun.jmx.snmp.IPAcl	PermissionImpl
java.awt.image	ColorModel

Table 3: Class hierarchies in JDK 1.5 that implement type-incompatible equality with instanceof.

Table 3 shows other similar cases we found from JDK 1.5. Note that these are all type hierarchies and a much larger number of classes are involved.

3.4 Suspicious implementations of type-compatible equality

When a type hierarchy is implemented by multiple developers, it can be easy for some to forget about the programming disciplines.

The Map interface defines a map entry (key-value pair) to model the mapping from a key to a

```
// implementation in java.util.Hashtable
public boolean equals(Object o) {
    if (!(o instanceof Map.Entry))
       return false;
    Map.Entry e = (Map.Entry)o;

    return (key==null ? e.getKey()==null :
    key.equals(e.getKey())) &&
    (value==null ? e.getValue()==null :
    value.equals(e.getValue()));
}

// implementation in java.text.AttributeEntry
public boolean equals(Object o) {
     if (!(o instanceof AttributeEntry)) {
        return false;
     }
    AttributeEntry other = (AttributeEntry) o;
    return other.key.equals(key) &&
    (value == null ? other.value == null :
    other.value.equals(value));
}
```

Figure 13: A mistake of type testing the wrong type when implementing type-compatible equality.

value. Formally, two entries e1 and e2 represent the same mapping if the following holds:

```
     (e1.getKey()==null ?
      e2.getKey()==null:
        e1.getKey().equals(e2.getKey()))  &&
     (e1.getValue()==null ?
      e2.getValue()==null:
        e1.getValue().equals(e2.getValue()))
```

This ensures that equals() works properly across implementations of the Map.Entry interface.

Figure 13 shows both a good implementation of equals() in Hashtable's map entry, which accepts Map.Entry, and an inappropriate implementation in java.text.AttributeString, which accepts only AttributeEntry. Another inappropriate implementation of map entry can be found in ParserImplTableBase (com.sun.corba.se.spi.orb). Clearly, these developers either do not know or forget to follow the right pattern in Figure 13. Although it is not certain whether the inappropriate type casting would lead to a problem, it would be a good idea to conform to the standard implementation shown in Hashtable.

3.5 Hybrid equality

Initially, the Map type hierarchy is designed to have a type-compatible equality. An abstract class java.util.AbstractMap provides the skeleton implementation for most of the operations specified in the Map interface, including the equals() method. AbstractMap is then extended by other maps.

However, not all subclasses are type-compatible with AbstractMap. For example, the Java Debugging API (com.sun.tools.jdi) defines a LinkedHashMap whose equals() requires its parameter to be the same type by an instanceof testing. It is also noticed that this class is very similar to a same named class in java.util, which inherits the equals from AbstractMap. Thus it seems reasonable to conclude that the jdi LinkedHashMap is intended to be type-incompatible with AbstractMap. Two other similar cases are RenderingHints (java.awt) and TabularDataSupport (javax.management.openmbean), which implement Map and should be type-incompatible with other maps due to domain semantics. Again, these two classes also use instanceof to implement type-incompatible equality, which can be better done by following the implementation shown in Figure 8.

The List hierarchy has similar subclasses that are intended to be type-incompatible. For example, com.sun.corba.se.impl.ior defines a list called FreezableList that is intended to be type-incompatible with other lists. Another class BakedArrayList from sun.swing is also type-incompatible, which specializes ArrayList for better performance. Although its author documents that BakedArrayList is for local use only and thus its equals() implementation is good enough, changing the List hierarchy into hybrid equality would make this class exhibit the general equals() behavior, which will ease future maintenance since the general behavior can always be assumed and one does not have to worry whether objects of this class are compared with other general lists.

3.6 Evolution of type hierarchy

When a type hierarchy is evolved, it needs to be revisited to ensure that the right equality is implemented properly.

The state of Rectangle2D (java.awt.geom) can be specified by 4 methods (getX(), getY(), getWidth(), getHeight()), all return double. The Rectangle class (java.awt) is a subclass of Rectangle2D that uses int as its representation. (There are several other subclasses using other representations like float and double.) In theory, a type-compatible equality can be implemented for this type hierarchy by defining an equals() in Rectangle2D. However, as shown in Figure 14, Rectangle also defines an equals() in addition to that of its superclass Rectangle2D.

The implementation of equals() in Rectangle is redundant and can be removed. By inspecting the code for Rectangle in JDK 1.0, we found that Rectangle exists before Rectangle2D (added

```
public abstract class Rectangle2D
 extends RectangularShape {
  /** ... @since 1.2 */
  public boolean equals(Object obj) {
   if(obj == this) { return true; }
   if (obj instanceof Rectangle2D) {
    Rectangle2D r2d=(Rectangle2D)obj;
    return((getX() == r2d.getX()) &&
      (getY() == r2d.getY()) &&
      (getWidth() == r2d.getWidth()) &&
      (getHeight() == r2d.getHeight()));
   }
   return false;
  }
  ... // Remainder omitted
}

/** ... @since JDK1.0 */
public class Rectangle extends Rectangle2D
 implements Shape, Serializable {
  public int x;
  public double getX() { return x; }
  ... // Remainder getters omitted
  public boolean equals(Object obj) {
   if (obj instanceof Rectangle) {
    Rectangle r = (Rectangle)obj;
    return ((x == r.x) && (y == r.y) &&
      (width == r.width) && (height == r.height));
   }
   return super.equals(obj);
  }
  ... // Remainder omitted
}
```

Figure 14: equals() in Rectangle2D and Rectangle.

in JDK 1.2). After Rectangle2D class was introduced in JDK 1.2, Rectangle was retrofitted to extend Rectangle2D. As a result, methods like getX() was added to Rectangle. Furthermore, as a quick fix, most of the original equals() was kept and a super.equals() call was added.

A similar case happens between java.awt.geom.Point2D and java.awt.Point.

A notable pattern in the evolution of type hierarchy is to specialize a general class for local use. The local use may relax from the subclass some of the restrictions put on the general class. For example, it may be guaranteed that 'the subclass may never interact with other general classes in the same hierarchy'. However, because the truth of such properties depends on the context of use, it can be costly to enforce. If such local properties are desired, they should at least be documented, as did in java.sql.Timestamp and sun.swing.BakedArrayList.

3.7 Implementation variations

In this section, common ways of implementing equals() are summarized and evaluated, particularly, calling super.equals() and type testing operations, and advices are given for their use.

We suggest to avoid calling super.equals() in equals() method unless absolutely necessary. For each class in a hierarchy, define its state first, including those inherited from its superclasses, and implement the equals() in terms of the state independent of any superclass. In this way, it becomes easier to understand the equals as everything it depends on is presented in a single location. Furthermore, a subclass gains the maximum independence from superclass because the equals depends on state instead of representation. Of course, this would imply that more code needs to be typed, especially for deep class hierarchies. But our experience is that most equals() are short.

To understand how super.equals() is called in practice, a checker was developed and 98 classes in JDK 1.5 are detected whose equals() call super.equals(). 72 out of 98 are for implementation reuse, most of which could be changed to follow the above advice easily. 11 out of 98 end up calling Object.equals(), and thus are redundant. In the example that follows, the super call should be replaced with false.

```
class Point2D { // java.awt.geom
...
public boolean equals(Object obj) {
  if (obj instanceof Point2D) {
      Point2D p2d = (Point2D) obj;
      return (getX() == p2d.getX()) &&
      (getY() == p2d.getY());
  }
  return super.equals(obj);
}
```

However, there is one special case where a superclass makes all its instance variables private but does not define accessors. In such cases, super.equals() would be justified. Finally, 5 out of 98 super calls are used in a subclass whose equals() is semantically equivalent to that of superclass but differs in performance or the representation used. For example, EnumMap is a map whose keys are enum. It provides a specialized implementation for equals() using its own representation. When the incoming argument is not EnumMap, the superclass' equals() is called. This represents another condition where super.equals() is justified in a subclass.

Table 4 depicts the use of various type testing operations in the 4 projects. instanceof as a type checking mechanism is more popular than getClass(), Type.class, and try-catch. The use of instanceof operator seems to be a norm for equals() implementation for most projects. This would explain why there are so many violations of symmetry when instanceof is used to implement type-incompatible equality between a supertype and a subtype (Section 3.3), where getClass() should

	JDK	Lucene	BCEL	SCL
instanceof	496	32	13	9
getClass	26	5	0	5
Type.class	0	2	0	0
try-catch	37	1	0	0
none	65	0	7	1
multiple testing	15	0	0	0

Table 4: Use of type testing in equals().

have been used instead.

3.8 Other design considerations

Two implementation details of equals() need to be considered. One is that before dereferencing the incoming parameter, it should be checked not to be null. The other is before casting the parameter, it must be tested that it can indeed be cast to the given target type. Using two checkers, we were able to conclude for JDK 1.5 26 cases of possible null pointer dereferencing and 30 cases of inappropriate type casting that may result in an exception.

Consider the equals() of SegmentInfo (org.apache.lucene.index).

```
public boolean equals(Object obj) {
 SegmentInfo other;
 try{ other = (SegmentInfo) obj;}
 catch(ClassCastException cce){return false;}
 return other.dir==dir&&other.name.equals(name);
}
```

This equals() checks for ClassCastException but fails to check for null value of parameter obj. Thus it may throw a NullPointerException. The author of this class may be relying on a local assumption that obj is never null, but since the cost of proper implementation is so little, it would be worthwhile to fix it so that one does not have to rely on this assumption.

The org.apache.bcel.classfile.Field class from the BCEL project is shown as follows.

```
public final class Field extends FieldOrMethod{
private static BCELComparator _cmp =
  new BCELComparator() {
public boolean equals(Object o1,Object o2){
 Field THIS = (Field) o1;
 Field THAT = (Field) o2;
 return THIS.getName().equals(THAT.getName())
 &&THIS.getSignat().equals(THAT.getSignat());
}
   ... // Remainder of BCELComparator omitted
};
public boolean equals( Object obj ) {
  return _cmp.equals(this, obj);
 }
  ... // Remainder of Field class omitted
}
```

Method equals(Object, Object) in the comparator does not check for the type of o1 and o2 before casting, and thus may throw ClassCastException. Furthermore, it also dereference THIS and THAT without checking for null, and thus may result in NullPointerException. Again, the author may rely on a local assumption that o1 and o2 are of the right type, but since the cost of proper implementation is so little, it would be worthwhile to fix it so that one does not have to rely on this assumption.

As a final example, consider the equals() shown below (com.sun.org.apache.xalan.internal.xsltc.compiler.-FunctionCall.JavaType).

```
static class JavaType {
     public Class  type;
     public int distance;

     public JavaType(Class type, int distance){
            this.type = type;
            this.distance = distance;
     }
     public boolean equals(Object query){
          return query.equals(type);
     }
}
```

Objects of this class are used as a value in a Map and are compared with objects of Class directly using aJavaType.equals(aClassObject) This solution works correctly with the current Map implementation. But again, this use is local and it relies on the assumption that the Map compares two objects with JavaType as a receiver object, which would be an extra burden for a future maintainer to ensure.

4 Summary of Guidelines

Based on the analysis in Section 2 and the case study in Section 3, we recommend the following guidelines for designing, implementing, and evolving equals(). The correct implementation of equals() requires proper identification of object state and designing the right type hierarchy. In this respect, our design guidelines for equals() are consistent with established principles for designing type hierarchy [2, 4]. But our guidelines are also equals-specific and require the developer to apply the right implementation strategy.

1. Identify state for each class in a hierarchy.
2. Use inheritance for subtyping rather than implementation reuse. If implementation reuse is used for a good reason and cannot be avoided, document it.

3. Decide the right equality that is needed for an inheritance hierarchy (type-compatible, type-incompatible, or hybrid equality) and use the corresponding implementation pattern.

4. Avoid piggybacking on equals() to implement relations other than equality. Add other predicates instead.

5. Minimize the dependency on superclass. Consider implementing equals() in terms of state (e.g., accessors) rather than internal representation. Avoid calling super.equals() whenever possible.

6. Minimize implementation variations. For example, avoid using exception handling to test object types.

7. Avoid the possibility of NullPointerException and ClassCastException by following the proper patterns for implementation. Avoid relying on local assumptions since it is easy to provide a reliable implementation.

8. When an inheritance hierarchy is changed, reconsider all of the above.

9. Keep in mind that Java's design can support only one equality. When more than one equality is needed, consider other design options in addition to equals().

5 Related Work

Liskov and Guttag suggest to use reference equality for mutable objects and value equality for immutable objects [2]. For example, an IntSet in their work uses reference equality. In contrast, many data types in Java Collection Framework employ value based equality. In fact, Java's specification for equals(), in particular, the consistency rule, is more relaxing and allows a mutable class to define equals(). The tradeoff is that the developer must ensure that the consistency rule is obeyed throughout his application.

When a type-compatible equality is implemented between a supertype and a subtype, their equals() conform to LSP. When two types implement a type-incompatible equality or a hybrid equality, they may violate LSP if specifications other than the default equals contract are assumed. Some (e.g., Bloch [1]) suggest to use composition instead of inheritance in such cases. We believe that type-incompatible equality should not be the main reason for two classes to not have a subtyping relation, and that it can still be useful for two types to subtype each other when their equals() violate LSP. In contrast to [1], the hybrid equality shows that it is possible to extend an instantiable class and add an aspect while preserving the equals contract.

The implementation for the hybrid equality essentially uses a template method design pattern [5]. Use of the template method design pattern for implementing equals() has also been proposed by Cohen [7, 8] and later by Stevenson and Phillips [9]. Our work is broader than theirs in that we provide a set of guidelines that is aimed to cover all of the relevant design and implementation considerations. Unlike the analytical work in [7, 9], we conducted an empirical study with real-life projects, with findings that both validate and enrich our guidelines. Furthermore, our design and implementation of hybrid equality appears to be simpler than theirs and can be more familiar to the programmers for a quick start.

Vaziri et al. [10] extend Java with relation types with which equality can be specified in terms of object properties and equals() implementation can be generated automatically. However, the relation types neither take into account Object's behaviors for equality comparison, nor support type-compatible and hybrid equality (the generated implementation uses getClass() only). Therefore, it would help solve only a subset of equals() implementation issues.

Baker [3] attempts to define a semantics model for object equality in the context of Lisp. However, the goal seems to be to completely implement equality at the programming language level. We believe that equality is domain-specific and requires developer involvement in its definition.

IDEs such as Eclipse and NetBeans have wizards that generate equals(). NetBeans 6.0 allows for only getClass() and fails to include the state of a superclass into the equals() of a subclass. Eclipse 3.3 gives a developer the option to choose from instanceof and getClass() but does not support the implementation of hybrid equality.

The FindBugs tool [13] detects 36 equals related problems. But most of these problems are related

[13]http://findbugs.sourceforge.net/. Verified July 25, 2008.

to equals() implementation *within a class* or the invocation of equals, and only one instance is related to a type hierarchy. FindBugs does not address issues related to hybrid equality or proper implementation of equals as proposed in our guidelines. Hou et al. [6, 11] use SCL to specify and detect violation of implementation constraints for equals() *within each individual class*, and inheritance is not the focus of those work.

6 Conclusion

We have shown that the implementation of a seemingly simple method like equals() can cause many problems. It is a general design problem that also appears in other language like C#. Fortunately, its solution goes hand in hand with the established OO design methods and principles such as behavioral subtyping. In particular, the identification of design intent for the right equality is crucial for the proper implementation of equals(). Furthermore, our analysis of 4 open source projects shows that this contract is widely depended on and it can be easily implemented improperly. Our analysis of the root causes for equals()-related problems provides insights and helps create concrete guidelines on things that should be done and things that should be avoided. These guidelines will be useful not only to software developers but also tool vendors as modern IDEs like Eclipse and NetBeans still cannot generate problem-free equals(). We plan to further extend these ideas to a static checker and distribute it publicly.

In general, it appears to us that the equality design is a representative example for design extension, a software development style that is increasingly becoming common in framework and other reuse-based development. Proper design extension requires the developers to possess enough knowledge about the design, who often lack such knowledge and as a result, write suboptimal code. We plan to investigate design guidelines for other extensible designs, and to develop tools to better assist the developers in the design extension process.

Acknowledgments

The authors are grateful to the three anonymous reviewers and H. James Hoover for their constructive feedback to an earlier version of this paper.

References

[1] J. Bloch, *Effective Java: Programming Language Guide*. Addison-Wesley, 2001.

[2] B. Liskov and J. Guttag, *Program Development in Java : Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.

[3] H. G. Baker, "Equal Rights for Functional Objects or, the More Things Change, the More They Are the Same," *SIGPLAN OOPS Mess.*, vol. 4, no. 4, pp. 2–27, 1993.

[4] B. H. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1811–1841, 1994.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] D. Hou and H. J. Hoover, "Using SCL to Specify and Check Design Intent in Source Code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 6, pp. 404–423, 2006.

[7] T. Cohen, "How Do I Correctly Implement the equals() Method?" *Dr. Dobb's Journal*, May 2002.

[8] R. Smit. Introducing equivalent types in equals() implementation. Last verified: April 10, 2008. [Online]. Available: http://www.forum2.org/tal/equals.html

[9] D. E. Stevenson and A. Phillips, "Implementing Object Equivalence in Java Using Template Method Design Pattern," in *SIGCSE 2003 Technical Symposium on Computer Science Education*, Reno, Nevada, USA, Feb. 2003, pp. 278–282.

[10] M. Vaziri, F. Tip, S. Fink, and J. Dolby, "Declarative Object Identity Using Relation Types," in *21st European Conference on Object-Oriented Programming*, Berlin, Germany, Aug. 2007, pp. 54–78.

[11] D. Hou, "SCL: Static Enforcement and Exploration of Developer Intent in Source Code," in *ICSE'07 COMPANION*, 2007, pp. 57–58.