

*I don't think you can break it.*

– Martin Bravenboer



# Code Generation for Axiom-based Unit Testing

This chapter presents a case-study of how the abstractions and infrastructure discussed in Part III and Part IV of this dissertation may be applied to source code analysis and interactive code generation. The motivation for the case study prototype, an interactive unit test generator, is that current unit testing methodologies often result in poor test coverage due to rather ad-hoc approaches.

Developers are encouraged to always write test cases, often as a driving force for new software features, or for systematic regression testing to avoid reintroducing known errors during software maintenance. This easily results in a development process focused around the individual test cases, rather than addressing the general requirements the cases are intended to represent. By expressing expected behaviours as axioms written in idiomatic Java code, it may be possible to improve the quality of the test code. Each axiom captures a design intent of the software as a general, machine checkable rule, rather than an individual case, and may be used to generate unit tests.

The quick and easy construction and integration of the test generator into an interactive development environment was made possible mainly due to the POM adapter technique described in Chapter 4 and the transformation runtime from Chapter 6. This chapter illustrates the applicability of the abstractions proposed in this dissertation, and, to a lesser degree, the workings of the generator tool. However, a detailed motivation and a discussion of the principal elements of the underlying test methodology are necessary before a detailed account of the transformation tool can be provided.

The results in this chapter were obtained in collaboration with Magne Haveraaen.

## 11.1 Introduction

Testing has gained importance over the past decade. Agile methods see testing as essential in software development and maintenance. Testing is given many different roles, from being the driving force of software development, to the somewhat more modest role of regression testing.

The most influential of these is probably Beck's extreme programming (XP) method [Bec98], giving rise to the mind-set of test-driven development (TDD) [Bec02]. TDD assumes that every software unit comes with a collection of tests. A new feature is added by first defining a test which exposes the lack of the feature. Then the software is modified such that all (both old and new) tests are satisfied. The process of extending software with a new feature can be summed up in five steps: (1) Add a test for a new feature which is written against the desired future API of the feature. (2) Run all tests and see the new one fail, but making certain that the unrelated features remain correct. (3) Write some code implementing the new feature, often just focusing on the minimal logic necessary to satisfy the tests written previously. (4) Rerun the tests and see them succeed. (5) Refactor code to improve the design.

A problem with a strict TDD approach is that each test is often casewise, i.e., it only tests one case in the data domain of a feature. While this opens up for implementing the logic of a new feature in small steps – or to insert dummy code for just passing the test – it does not ensure that the full feature will be implemented. For these reasons, refactoring assumes a prominent place. Using refactoring techniques, the feature implementation may be incrementally generalised from the pointwise dependencies required by the tests to the full logic required by the intended application.

Many tools have been developed to support TDD. In Java, test-driven development is most often done using JUnit [BG], a Java testing framework for unit tests.

Arguably, the focus on agile methods has taken the focus away from formal methods at large. This is somewhat unfortunate, as a substantial amount of work has been done on effectively using formal methods as a basis for testing. For example, in 1981, the DAISTS system [GMH81] demonstrated that formal algebraic specifications could be used as basis for systematic unit testing. DAISTS identified four components of a test: *Conditional equational axioms* that serve as the test oracle; the *implementation* which must include an appropriate equality function; the *test data cases*, and a *quality control* of the test data (at least two distinct values). The Dais-tish system [HS96] took these ideas into the object-oriented world by providing a DAISTS like system for C++. The notation used for the axioms were also conditional equational based, giving a notational gap between the specification functions and the C++ methods. A more recent approach which also maintains this distinction is JAX (Java Axioms) [SLA02], which merges JUnit testing with algebraic style axioms. An automation was later attempted in AJAX using ML as notation for the axioms. These experiments suggest that an algebraic approach to testing may result

in much more thorough tests than standard hand-written unit testing. The emphasis on general axioms rather than specific test cases is the key to better unit tests.

This chapter builds on these above-noted ideas by introducing and describing several novel improvements which lead up to a tool-assisted method for implementing modular axiom-based testing for JUnit and Java. The two main contributions of this chapter include:

- A case study of the practical application of program object model adapters, transformlets and the other infrastructure (Chapter 6) for expressing interactive program generation tools.
- The detailed discussion of a generator tool, JAxT (Java Axiomatic Testing) [KH], which automatically generates JUnit test cases from all axioms related to a given class.

Additionally, the tool construction resulted in several new techniques and developments related to the research in testing pursued by Haverlaen [HB05], including (1) a technique for expressing as reusable axioms the informal specifications provided with the Java standard API; (2) a flexible structuring technique for implementing modular, composable sets of axioms for an API, which mirrors specification composition, as known from algebraic theory; and, (3) a discussion of practical guidelines for writing test set generators that exercise the axioms.

## 11.2 Expression Axioms in Java

In the proposed approach, axioms are expressed as idiomatic Java code, not in a separate specification language, as is common with other approaches based on algebraic specifications. There are several benefits to this: First, the developers need not learn a new formal language for expressing the properties they want to test. Second, the axioms will be maintained alongside the code and restructuring of the code, especially with refactoring tools, will immediately affect the axioms. This reduces or prevents any “drift” between the implementation and the specifications. Third, code refactoring, source code documentation and source navigation tools may be reused as-is for expressing and developing axioms. Fourth, it becomes easy to write tools to automatically produce test cases from the axioms. This is important because axioms may easily contain errors. This makes early and frequent testing of axioms desirable.

### 11.2.1 JUnit Assertions

The approach discussed here uses axioms to express invariants – assertions – about desired properties for an abstraction. The Java `assert` mechanism allows these prop-

erties to be stated as boolean expressions. If the expression evaluates to false, the assert mechanism will fail the program by throwing an `AssertionError` exception.

For testing purposes, the JUnit system provides a wider range of assertions than what `assert` offers. A notable difference is that the failure of one assertion terminates the immediately surrounding test, but not the remainder of the test suite. This allows a full set of tests to run, even if the first test fails. In addition, the JUnit assertions provide a detailed account of the error if the assertion does not hold, making it significantly easier to trace down what the problem can be.

### 11.2.2 Java Specification Logics

In standard specification theory, such as that used in [GMH81], axioms are formed from terms (expressions) with variables (placeholders for values or objects). If a variable is not given a value in the axiom, e.g., by quantification, it is said to be free. Interpreting this in the context of a programming language, free variables of a term can be viewed as parameters to the term. This leads to the following definition:

**Definition 6** *An axiom method, or axiom, is a public, static method of type void, defined in an axiom class. The method body corresponds to an axiom expression (term), and each method parameter corresponds to the free variables of that expression (term). When evaluated, an axiom fails if an exception is thrown, otherwise it succeeds.*

It is recommended, but not required, that axioms use assertion methods in the style of JUnit, e.g.:

```
public static void equalsReflexive(Object a) { assertEquals(a,a); }
```

This axiom states the reflexive property for any object of class `Object` (or any of `Object`'s subclasses). The method `assertEquals(Object a, Object b)` is provided by JUnit and checks the equality of the values of the two objects using `a.equals(b)`. The axiom will also hold if `a` is the `null` object, since `assertEquals` safeguards against this case.

Specifying the desired behaviour of exceptions is also straightforward, albeit significantly more verbose:

```
public static void equalsNullFailure(){
    Object a = null, b = new Object();
    try {
        a.equals(b); // calling equals on the null reference
        fail(); // exception should have been raised
    } catch (NullPointerException e) {
        // OK
    }
}
```

Here, the effect of applying `equals` to a null reference is written as an axiom, named `equalsNullFailure`, with no free variables. The axiom states that, for any `a` and `b` where `a` is the null object, the expression `a.equals(b)` must raise an exception of type `NullPointerException`. (This is the Java semantics for invoking methods on null references.)

### Expressive Power

In specification theory, one often asserts the expressive power of a specification logic [MM95]. The simpler logics have less expressive power, but have better meta-properties than the more powerful logics, i.e. reasoning about the logic is less difficult. It is beyond the scope of this chapter to provide a detailed classification and comparison of Java versus other specification logics, except for the following brief remarks.

*Equational Logic* – The simplest logic for the specification of abstract data types – classes – is equational logic which asserts that two expressions are equal (for all free variables). This is intuitively captured using `assertEquals` based on the `equals` method of Java. However, there are several theoretical problems here.

First, in normal logic a term is composed of mathematical functions deterministically relating inputs to outputs. In stateful languages, such as Java, one may modify one or more arguments rather than returning a value. The function may be (semi-) non-deterministic or the result of a function may depend on an external state. Such methods are beyond standard equational logic. As long as a method is deterministic, it is mostly straight forward to reformulate the terms of an equation as a sequence of statements computing the two values to be checked against each other.

Second, the `equals` method, on which `assertEquals` is based, may not be correctly implemented. So one should treat `equals` as any other method, and hence, also make certain it satisfies certain properties: it should be deterministic; it must be an equivalence relation (reflexive, symmetric, transitive); and, it should be a congruence relation, i.e., every method should preserve equality with respect to the `equals` methods. Fortunately, these are properties that can be written as Java axioms, and then tested<sup>1</sup>. For instance, one may repeatedly evaluate `equals` on two argument objects and ascertain that the `equals` should always have the same result as long as one does not modify the objects. Interestingly, the first two of these requirements are formulated in the Java API [Jav]. The last requirement will be discussed in section Section 11.4.

Third, there may be no way of providing a relevant `equals` method for some class, e.g., a stream class. This is known as the oracle problem [GA95]. However, using properly configured test setups, these classes may be made mostly testable as well.

*Conditional Equational Logic* – A more powerful logic is conditional equational logic. This allows a condition to be placed on whether an equality should be checked

---

<sup>1</sup>Testing will never prove these properties, but will serve to instill confidence about their presence.

for. In Java axioms, this is done by using an if-statement to guard whether the `assertEquals` should be invoked. Axioms for symmetry and transitivity of the `equals` method use this pattern, e.g.:

```
public static void equalsSymmetry(Object x, Object y) {
    if (x != null && y != null)
        assertEquals(x.equals(y), y.equals(x));
}
public static void equalsTransitive(Object x, Object y, Object z) {
    if (x != null && y != null && z != null)
        if (x.equals(y) && y.equals(z))
            assertEquals(x, z);
}
```

Here, an explicit null-check is required to avoid problems with null references in the assertions.

*Quantifier-free Predicate Logic* – Quantifier free predicate logic is an even more powerful logic. It permits the expression of negations and conditionals anywhere in the logical expressions. This is trivially expressed using boolean expressions in Java.

*Full Predicate Logic* – Full predicate logic causes a problem with the quantifiers. A universal quantifier states a property for all elements of a type. There is no counterpart in programming, although supplying arbitrary collection classes and looping over all elements will be a crude (testing style) approximation. Existential quantifiers may be handled by Scholemisation – given that there are algorithms for finding the value the quantifier provides.

**Java-style Axioms** Java style axioms have a different expressive power and allow expressing properties not captured by the standard logic. For instance, the distinction between an object and its reference is easily handled by JUnit assertions. Exceptions and methods that modify their arguments, rather than returning a result, can also be dealt with easily. Further, statistical properties can be expressed, such as the well-distributedness requirement on the `hashCode()` method, or temporal properties related to processes and timings, even against physical clocks.

The drawback to this extra expressive power is that one cannot immediately benefit from the theoretical results from the more standard specification logics. That is, the general theoretical results from algebraic specifications are not directly applicable to specifications written in a “Java logic”.

This chapter will stick to the more value-oriented aspects of Java as a formal specification language, hopefully giving an indication of the intuitive relationship between this style and the standard specification logics.

## 11.3 Structuring the Specifications

All classes in Java are organised in a strict hierarchy forming a tree with the type `Object` at the top. A class may implement several interfaces. An interface may inherit several other interfaces. Further, a given class should satisfy the (informally stated) assumptions and requirements of each of its supertypes<sup>2</sup>. This is illustrated by the following simple class `Position`, which is used to index the eight-by-eight squares on a chess board:

```
public class Position implements Comparable<Position> {
    private int x, y; // range 0<=x,y<8
    public Position(int a, int b) { x=a % 8; y=b % 8; }
    public int compareTo(Position q) {
        return x-q.x; // ordering only on X-component }
    public boolean equals(Object obj) {
        final Position q = (Position) obj;
        return x==q.x && y==q.y; }
    public int getX() { return x; }
    public int getY() { return y; }
    public int hashCode() { return 3*x+y; }
    public void plus(Position q){ x=(x+q.x) % 8; y=(y+q.y) % 8; }
}
```

The method `plus` gives movements on the board, e.g., `k.plus(new Position(1,2))` for moving a knight `k`. The position class is a subclass of `Object` (which is implicitly inherited) and it implements `Comparable<Position>`. The intent is that the `Position` methods should satisfy all requirements given by its supertypes, i.e., those from the class `Object` and those from the interface `Comparable<Position>`, as well as any requirements given for `Position` itself.

### Institutions

It would be desirable to write the requirements of classes and interfaces as sets of axioms, in a modular fashion, and then allow these sets to be composed soundly. The notion of institution [ST88] provides the mathematical machinery permit expressing requirements in modules called specifications. Each specification provides a set of axioms. Operations exist for building larger, compound specifications from smaller ones. The specification for a type can be extended with new methods, thus dealing with the extension of classes or interfaces by for example using inheritance. Axioms may be added to a specification, for example to provide additional requirements for a subtype. The union of specifications may also be taken. This allows the construction

---

<sup>2</sup>In Java terminology a type encompasses both class and interface declarations.

of a compound specification for all supertype axiom sets.

The theory of institutions shows that one can safely accumulate any axioms from the supertypes as well as add new axioms for `Position`. More importantly, it shows that this accumulation will be consistent and not cause any unforeseen interaction problems, as often is the case when one considers inheritance among classes. In this sense, the modularisation and composition properties of specifications are a lot more well-behaved than that of software code. In addition, framework of institutions provides a significant freedom in organising axioms so that they become convenient to work with. The method described in this chapter uses this freedom to allow a flexible and modular organisation of axioms alongside the class and interface definitions.

### 11.3.1 Associating Axioms with Types

Axioms, in the form of static methods, are grouped into Java classes. This immediately integrates the axioms with all Java tools. During the development process, axioms will be refactored along the main code, e.g., when a method is renamed or the package hierarchy is modified. This is considerably more developer-friendly than using separate specification languages.

**Definition 7** *An axiom class is any class  $A$  which implements a subinterface of `Axioms<T>`, contains only axiom methods, its axiom set, and where  $T$  specifies which type the axioms pertain to.*

The name of a class providing axioms may be freely selected and placed in the package name space, but it must be labelled with an appropriate *axiom marker*. Labelling is done by implementing one of the predefined subinterface of `Axioms<T>` to signify whether the axiom set is required or optional for  $T$  or its subtypes.

**Definition 8** *Required axioms are defined using the `RequiredAxioms<T>` marker interface on an axiom class  $A$ , and states that all axioms of  $A$  must be satisfied by  $T$  and all its descendants.*

Using this structuring mechanism, it is possible to group the required axioms for `equals`, introduced in Section 11.2.2, into a class `EqualsAxioms` which implements `RequiredAxioms<Object>`, by defining the methods `equalsSymmetry`, `equalsTransitive` and `equalsNullFailure` in this class (to complete the specification for `equals()`, axioms for testing reflexivity and determinism are also required). Similarly, hash code axioms may be captured as follows:

```
public class HashCodeAxioms implements RequiredAxioms<Object> {
    public static void congruenceHashCode(Object a, Object b) {
        if (a.equals(b)) assertEquals(a.hashCode(), b.hashCode());
    }
}
```



```
1 public class PositionPlusAxioms implements RequiredAxioms<Position> {
2     public static void associativePlus(Position p, Position q, Position r) {
3         // compute pc = (p+q)+r;
4         Position pc = new Position(p.getX(), p.getY());
5         pc.plus(q);
6         pc.plus(r);
7         // compute p = p+(q+r)
8         q.plus(r); // destructive update
9         p.plus(q); // destructive update
10        assertEquals(pc, p);
11    }
12    public static void commutativePlus(Position p, Position q) {
13        Position pc = new Position(p.getX(), p.getY());
14        pc.plus(q);
15        q.plus(p); // destructive update
16        assertEquals(pc, q);
17    }
18 }
```

Figure 11.1: Axioms requiring that plus is associative and commutative.

Figure 11.1 shows some axioms for `Position`. Since `plus` modifies its prefix argument, a separate object `pc` is necessary to not modify `p` before it is used as an argument in the second additions (lines 9 and 15). This is not a problem for `q`, as it is not modified in the first additions. These axioms are destructive on the test data: the values of some of the arguments have been modified when the axioms have been checked (lines 8, 9 and 15).

The axioms belong to the same package as the type the axiom is associated to, so `PositionPlusAxioms` should be in the same package as `Position`. When axioms are retrofitted to an existing API, this placement may not be possible. One solution is to place the new axiom classes in an identically named package, but with `jaxt` as a prefix to the package name. Then the axioms for `Object` would go in a package `jaxt.java.lang` and the axioms for the Java standard collection classes would end up in `jaxt.java.util`.

Figure 11.2 shows how all the axioms for the supertypes, together with the axioms for `Position`, specify the design intent for `Position`.

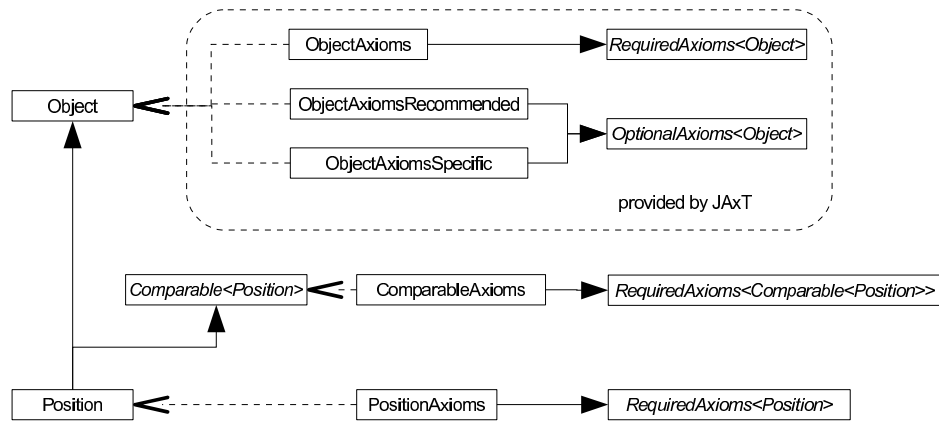


Figure 11.2: Organisation of axiom sets for `Position`. Boxes with italics represent interfaces. `X`Axioms-classes depend on, or pertain to, the classes they describe, marked by a stippled arrow. Filled arrows indicate inheritance.

### 11.3.2 Optional and Inherited-only Axioms

A close reading of the Java API documentation [Jav] shows that it not only contains requirements – the kind of axioms described in this chapter – but also a multitude of recommendations and class-specific descriptions. For instance, the description of `Comparable<T>` contains the phrase “strongly recommended” and other places use the phrase “recommended”. In the class `Object`, and some of its subclasses, such as enumerations, it is specified that reference equality is the same as value equality.

**Definition 9** *Optional axioms are defined using the `OptionalAxioms<T>` marker interface on an axiom class  $A$ , and states that all axioms of  $A$  pertain to  $T$ , but are optional for any descendant of  $T$ , unless the programmer of a subtype has requested these axioms specifically. If  $A$  is an optional axiom set of  $T$ , this set may be inherited to a descendant  $D$  of  $T$  by adding the marker interface `AxiomSet<A>` to an axiom class pertaining to  $D$ .*

The optional reference equality of `Object` is asserted by `equalsReference()` in the following axiom class, `ObjectEqualsReference`:

```

public class ObjectEqualsReference implements OptionalAxioms<Object> {
    public static void equalsReference(Object x, Object y) {
        if(x != null)
            assertEquals(x.equals(y), x == y);
    }
}

```

This axiom class implements the `OptionalAxioms<T>` interface, and therefore, the method `equalsReference()` will not be required automatically by the subtypes of

<code>RequiredAxioms&lt;T&gt;</code>	<i>required by type T and all descendants</i>
<code>OptionalAxioms&lt;T&gt;</code>	<i>required by type T, but not its descendants.</i>
<code>SubclassAxioms&lt;T&gt;</code>	<i>required by all subtypes of T but not by T</i>
<code>AxiomSet&lt;Ax&gt;</code>	<i>import axiom set Ax</i>

Table 11.1: Summary of axiom structuring mechanisms.

`Object` unless one of their axiom classes implements `AxiomSet<ObjectEqualsReference>`. Consider the following empty axiom class which states axioms pertaining to the class `EnumDemo` (not shown).

```
public class EnumDemoAxioms implements OptionalAxioms<EnumDemo>,
                                     AxiomSet<ObjectEqualsReference>
{ }
```

While the class `EnumDemoAxioms` does not specify any axioms itself (although it could), it does activate the axioms from the set `ObjectEqualsReference`; that is, the optional equality axioms are now required for `EnumDemo` (but none of its subclasses, since `EnumDemoAxioms` is itself marked optional). As expected, even though `Position` inherits directly from `Object`, the object equality axioms (`ObjectEqualsReference`) have no relevance since the axiom classes for `Position` do not implement the type `AxiomSet<ObjectEqualsReference>`.

It is sometimes necessary to state axioms that only pertain strictly to subclasses, and not originating from the base class which is exempt. This is done using subclass axioms.

**Definition 10** *Subclass axioms are defined using the `SubclassAxioms<T>` marker interface on an axiom class A, and states that all axioms of A pertain to all subtypes of T, but not T itself.*

Consider a (possibly abstract) class `c` which contains declarations and common methods for its subclasses where one may want to check as much as possible of `c` and be certain that all subclasses satisfy all axioms. By marking some axioms with `SubclassAxioms<C>`, these will not be tested on class `c` itself, but will be checked on all subclasses of `c`.

Table 11.1 summarises the structuring mechanisms for axioms. These are used in the small example class hierarchy depicted in Figure 11.3. It is important to note that as all these relationships are formally marked in the code, they can be discovered automatically by a tool, see Section 11.6.

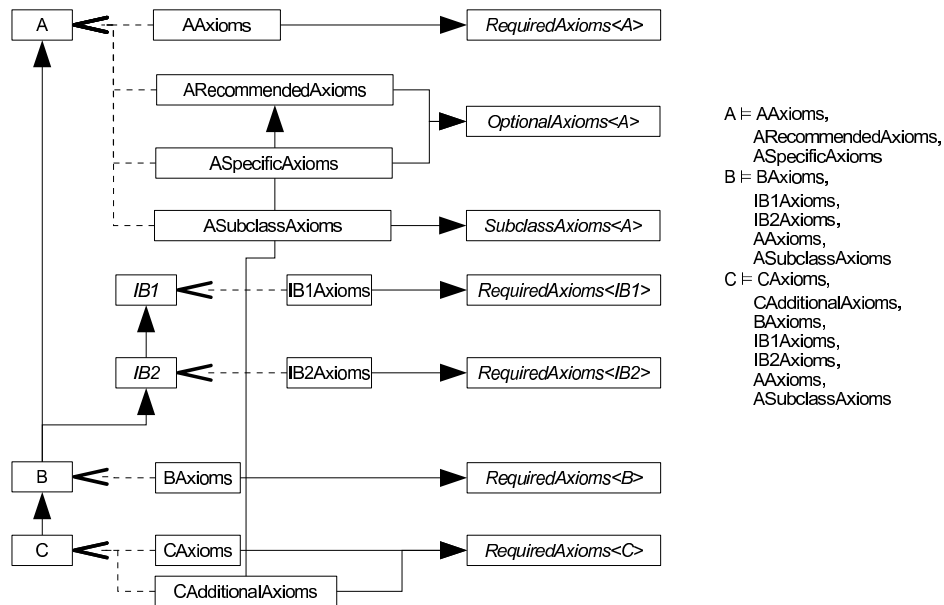


Figure 11.3: Organisation of axioms. The notation  $X \models AxL$  means that class  $X$  satisfies all the axioms in the axiom list  $AxL$ .

## 11.4 Java API caveats

Capturing the informal requirements described in the Java API documents into machine-checkable axioms is a non-trivial exercise. This section documents some of the challenges encountered in this process.

### 11.4.1 Override and Overload

Most object-oriented languages, including Java, distinguish between overriding a method and overloading a method. Method overriding occurs when a subclass re-defines a method defined in one of its superclasses, thus altering the behaviour of the method itself without changing the class interface. On the other hand, method overloading allows the same method name to be reused for different parameter lists, e.g.:

```

1 public class Test extends Object {
2     int x, y;
3     public boolean equals(Object obj){
4         return x==((Test)obj).x
5             && y==((Test)obj).y;
6     }

```

```
7 public boolean equals(Test obj){
8     return x==obj.y;
9 }
10 }
```

The class `Test` provides two overloaded methods where the first overrides the `equals` method from `Object`. Consequently, the axioms for `equals()`, defined in `Object`, will be applied to this method, while the `equals` method in lines 7-9 will not be tested. This might be surprising, but follows from the semantics demonstrated by the following method calls:

```
Object o1 = new Object();
Object c1 = new Test();
Test c2 = new Test();
o1.equals(c2); // from Object
c1.equals(c2); // from lines 3-6
c2.equals(c2); // from lines 7-9
```

The overloaded version will only be called if both arguments have the *declared* type `Test` (or a subclass thereof).

In a language with templates, like C++, one could use genericity to apply the axioms to any one of such overloaded methods. Unfortunately, this is not possible using Java generics (since the type erasure of a generic `<T> boolean equals(T o)` would be identical to `Object equals(Object o)` defined in `Object`. This is forbidden by the type system.) so the axioms for `equals` will have to be redeclared for class `Test` if they are to be applied to the second `equals` (line 7-9).

### 11.4.2 `clone` and Other Protected Methods

The `clone()` method is declared as a protected method of `Object`. This makes it problematic to specify axioms because a protected method is only accessible to subclasses. Since `clone()` is protected, it is not possible for an axiom method to invoke it on an object of type `Object` and, consequently, it cannot be adequately described. This is unfortunate, because the API documentation lists an important number of recommended (optional) axioms for `clone()`.

In principle, the same limitation holds for any protected method and is shared by any testing approach where the testing methods are defined in a class separate from the class being tested. In Java, it is possible to circumvent this limitation by declaring the testing class in the same package as the tested class – classes in the same package may invoke each other's protected methods without being in a subtype relationship.

For `clone()`, which is defined in the immutable package `java.lang`, axioms must be written specifically for every class that makes `clone()` public. The axioms will then apply to all subclasses of this class, but not to any other class that exposes the

`clone()` method. Following the previous remarks, it is not possible to circumvent this restriction using Java generic method declarations.

In the instance of `clone()`, another attractive possibility would be to state the axioms for the interface `Cloneable`, `Cloneable` is defined in the Java language specification as a marker interface used to enable the cloning algorithms encoded in the `clone()` method from `Object`. Unfortunately, `Cloneable` is an empty interface which does not (re)declare `clone()` as a public method.

### 11.4.3 The equals “congruence” relation

In the standard approaches to equational specifications, the equality test is a congruence relation. This is an equivalence relation which is preserved by all methods. Preservation means that, for any two argument lists to a method, if the arguments are pairwise equal, then the results of the two method calls also must be equal.

In Java, the `equals` method defined in `Object` is close to, but not entirely, a congruence relation. If it were, for any two objects `a` and `b`, `a.equals(b)`, then

- `a.hashCode()==b.hashCode()`, which is a required axiom in the Java API,
- `(a.toString()).equals(b.toString())`, but this is not an axiom in the Java API.

Unfortunately, this means that one of the central means for closely relating Java to equational specification theories is unavailable.

For `equals` to have the congruence property, it would have to be implemented as a form of “meta property”, requiring axioms to be written for every new method. Remember that overridden methods inherit such properties from the superclass. A tool like JAxT could easily handle this by either explicitly declaring the needed congruence axioms, or tacitly assuming them, thus generating the necessary test code even without making the axioms explicit. Even without tool support in the current iteration of JAxT, maintaining a congruence relation is a *strongly recommend* practise wherever feasible. Future releases may add facilities for handling the congruence property.

## 11.5 Testing

The previous section described how to express and organise axioms in and for Java. These axioms can be used as test oracles for ensuring implementations exhibit the intended behaviour. For the testing setup to be complete, relevant test data must be provided. During testing, each data element from the test data set will be provided for the relevant free variables of the axioms.

### 11.5.1 Test Data Generator Methods

A test data set may be as simple as the data from a typical test case, as practised by typical agile or test-driven development. For the many cases where additional testing is desired, the JAxT framework has an infrastructure that opens up for a much more systematic approach to test data.

Creating the test is the responsibility of the developer. The first step towards creating a test set is to implement a test set generator. The JAxT generator wizard, explained in Section 11.6, will provide a test set generator stub on the following form, for a class *X*:

```
public class XTestGenerator {
    public Collection<X> createTestSet()
        { return null; }
}
```

The developer must fill in the `createTestSet()` method with code that produces a reasonable test set of *X* objects. This can be data from an existing test case, or a static collection of test values.

For `Position` objects, the following test set generator method, placed in the class `PositionTestGenerator`, produces a collection of random, but valid, `Position` objects:

```
public static Collection<Position>
createTestSet() {
    final int size = 200;
    List<Position> data =
        new ArrayList<Position>(size);
    Random g = new Random();
    for(int i = 0; i < size; i++) {
        data.add(new Position(g.nextInt(8),
                               g.nextInt(8)));
    }
    return data;
}
```

The random number generator provided by the Java standard library is used to produce test data. Some authors, such as Lindig [Lin05], have reported that random test data may be more effective than most hand-crafted data sets in detecting deficiencies. In a similar approach, discussed by Claessen and Hughes [CH00], algorithms are used for deriving random test data sets based on abstract data types.

In the particular case of `Position`, the complete test data set of the 64 distinct position values could be provided, but for completeness, distinct objects with equal values due to the difference between equality on object references with equality on object values are also needed.

If the objects of a class `X` are very time-consuming to instantiate, one might consider implementing a test set generator method that extends `IncrementalTestGenerator`. This class provides a ready-made `createTestSet()` method that returns a collection which instantiates its elements on demand. The developer must implement the method `X generateNewValue(int index)`, which must produce random objects of type `X`. The `IncrementalTestGenerator` takes care of memoization.

Using `IncrementalTestGenerator` will not result in better total running times of the tests. It may, however, allow the developer to uncover errors earlier in the event of a failed axiom since no time is spent up front to generate a full test data set which will never be traversed entirely. A formulation using the `generateNewValue()` method may sometimes be cumbersome. For this reason, the use of `IncrementalTestGenerator` is optional.

### 11.5.2 Determining Test Set Quality

The `JAxT` library offers a few simple, but powerful checks for properties of test sets. For example, there is a method that checks whether the provided collection has at least two distinct data values, similar to the requirements in [GMH81]. Another method can test whether there are at least three distinct objects with equal values – necessary if a transitivity axiom is to be exercised.

The purpose of these checks is so developers can apply them to the test sets produced by their generators and ensure some degree of test set quality. The quality assurance of test sets usually occurs during the development of test set generators themselves. In general, it is not necessary to run test quality metrics as part of a test suite, provided that the developer has checked and acquired sufficient confidence in the test set generators.

Additional checks, in particular statistical metrics which can be used to judge distribution characteristics, are scheduled for inclusion into `JAxT`, but how to best integrate existing test generation approaches is still an open problem. There is a wealth of material to choose from, however, such as [DO91, TL02].

### 11.5.3 Running the Tests

When combining the test data with the axioms to run the tests, there are several issues to take into account. Some of the axioms may be destructive on the data sets, so each test data element must be generated for each use. This is normally handled by fixtures in unit testing tools, but for efficiency reasons, one may choose to have test data generation in the test methods themselves. While this entails more verbose unit test methods, since the test methods will be automatically generated from the axioms and the test data generator methods, it presents no extra burden on the user.



With automated test data generation, it becomes very easy to (accidentally) create large data sets. In general, larger test sets improve the quality of the testing, but run-times may become excessive when testing axioms with many arguments. In the example `Position` class, all tests for axioms with up to three free variables take less than a couple of seconds and are within the normal time-frame for repeated unit testing in the TDD approach. For more complicated axioms, such as checking that `equals` is a congruence for `plus`, a quadruple loop on the data set is required and takes about 30 seconds. While in the the edit-compile-run cycle, regular unit testing using large data sets is not ideal. The framework currently provides limited support for adjusting the data set sizes. Additional work independent of JAxT is required to allow the developer to flexibly tune the size, and to continuously vary the trade off between thorough testing versus short testing times.

### 11.5.4 Interpreting Test Results

Writing code and axioms, and their associated tests is error-prone. For this reason, early and frequent testing is valuable. When a test fails, it only states that there is some mismatch between the formulated axiom and the implementation of the methods used in the axiom. It is important to remember that, at least in the beginning, errors can just as easily be in the axiom as in the code. Therefore, both must be checked carefully. As always, newly written pieces of code, whether a new axiom or a new class, are typically more likely to contain errors than legacy pieces that have already been thoroughly tested.

## 11.6 Test Suite Generation

The techniques described in the previous sections are structured and formal enough for a tool to aid the developer in deriving the final unit tests and test set generators. The author has experimented with such automatic generation of unit tests from axioms by building a prototype testing tool called JAxT[KH]<sup>3</sup>. This section describes the findings from this prototyping experiment.

Below is the test class generated by JAxT for `Position`, with comments removed:

```
public class PositionTest extends TestCase {
    private Collection<Position> testSetPosition;
    public PositionTest(String name)
    { super(name); }
    protected void setUp() throws Exception {
        super.setUp();
        testSetPosition =
```

---

<sup>3</sup>JAxT stands for Java Axiomatic Testing.

```

    PositionTestGenerator.createTestSet();
}
protected void tearDown() throws Exception {
    super.tearDown();
    testSetPosition = null;
}
public void testObjectReflexiveEquals() {
    for (Position a0 : testSetPosition)
        reflexiveEquals(a0);
}
public void testComparableTransitiveCompareTo() {
    for(Position a0 : testSetPosition)
        for(Position a1 : testSetPosition)
            for(Position a2 : testSetPosition)
                transitiveCompareTo(a0, a1, a2);
}
}

```

The following sections will explain how this test case was derived.

### 11.6.1 Generating Tests from Axioms

The task of JAxT is to automatically derive unit tests for a given class *C* using those axioms from the set of all axioms associated with *C* – each axiom induces a new unit test. The set of associated axioms can be found by inspecting the axiom classes associated with *C*.

When the axioms were created, the programmer clearly specified which axioms directly pertained to *C* by placing *C*'s axioms into those classes implementing the interface `Axioms<C>`. By placing the marker `Axioms<C>` on the an axiom class *AX*, all (static) methods in *AX* are considered to be axioms for *C* and must therefore be fulfilled by a descendant of *C*.

This implies that *C* itself may have inherited axioms from a related superclass. For any (direct or indirect) superclass *P* of *C* or (direct or indirect) interface *I* of *C*, one or more axiom classes may exist with `Axioms<P>` or `Axioms<I>` markers. Methods in these classes are also considered to be axioms associated with *C*.

#### Computing Axiom Sets

In order to produce the final set of test methods for a class *C*, all applicable axiom methods must be found. As suggested in Figure 11.3, axioms for all named types provided by *C*, i.e. its superclasses, its or any of its supertypes' interfaces, are searched.

The algorithm `compute-axioms()` detailed in Algorithm 1 produces the final list of axiom methods for `C`. The resulting list is then fed into a test case generator. The test generator works as follows:

First, it computes the required axiom sets of `C`; that is, all classes which implement `RequiredAxioms<C>` or `OptionalAxioms<C>`. Next, the supertypes of `C` are traversed and, for each, the set of subclass and required axioms are collected. After this is done, an initial set of axiom classes (i.e. axiom sets) is given in  $\Xi$ . Note that the axiom classes themselves may pull in additional (optional) sets, via the `implements AxiomSet<AX>` mechanism. These optional sets are then added to  $\Xi$  and the final set of axiom sets is obtained. The methods of these axioms are the final product of `compute-axioms()`.

---

**Algorithm 1** `compute-axioms(C)`


---

```

 $\Xi := \text{required-axiom-sets-of}(C) \cup \text{optional-axiom-sets-of}(C)$ 
for  $T \in \text{supertypes-of}(C)$  do
   $\Xi := \Xi \cup \text{subclass-axiom-sets-of}(T) \cup \text{required-axiom-sets-of}(T)$ 
end for
for  $AX \in \Xi$  do
   $\Xi := \Xi \cup \text{super-axiom-sets-of}(AX)$ 
end for
return  $\bigcup_{AX \in \Xi} \text{methods-of}(AX)$ 

```

---

### User interaction

Not all the details of the generation can be inferred from the source code, such as which package the generated test class should be placed in, though reasonable defaults can be suggested. To support various working styles and project organisations, the prototype offers a graphical generator wizard that allows user to optionally specify corrections to the assumed defaults. The user accesses the GUI to select a single class or multiple classes or packages to invoke the test generator. A wizard appears that allows the user to select which classes to generate test set generators for and where to place them. Further, the user can select which package the generated test case should be placed in and whether to generate a test suite, if tests for multiple classes have been requested.

The user may also include additional axiom libraries that may contain relevant axioms for the type hierarchy at hand. For example, the JAxT library already provides axioms for `Object` and `Comparable` that may be reused as desired. This axiom inclusion feature supports reuse of axiom libraries that may be distributed separately from existing implementations. A major benefit of this design is that axioms can be easily retrofitted for existing libraries, such as the Java Standard Libraries, and such

axiom libraries can be incrementally developed without modification or access to the source code of the original library.

Thanks to JAxT, users can easily and incrementally update existing test classes, e.g., when axioms have been added or removed. By reinvoking JAxT, the test classes will be regenerated and additional test class stubs for new data types will be created. As Eclipse refactorings carry through to the axioms, the axioms will remain in sync with the code they test.

### Generation of Tests

When the user requests axiom tests for a class  $X$ , a corresponding  $X$ AxiomTests is generated (name may be customised). This is a JUnit fixture, i.e.  $X$ AxiomTests derives from `junit.TestCase`, provides a set of test-methods and may provide a `setUp()` and a `tearDown()` method.

The `setUp()` method initialises all necessary test sets, as follows:

```
setUp() {
    testSetT0 = T0Generator.createTestSet();
    ...
    testSetTn = TnGenerator.createTestSet();
}
```

The `createTestSet()` methods return `Collections` which will be traversed by the tests. The exact set of  $T_i$ Generator calls is discovered from the argument lists of the axioms exercised by this test fixture.

For each axiom  $ax(T_0, \dots, T_n)$  in axiom class  $A$  a `testAAx()` method is generated, on the following form:

```
/** {@link package.of.A#ax(T0, ..., Tn)} */
public void testAAx() throws Exception {
    for(T0 a0 : testSetT0)
        ...
        for(Tn an : testSetTn)
            A.ax(a0, ..., an);
}
```

This test will invoke the axiom  $A.ax()$  with elements from the (random) data sets `testSetTi`. The generated Javadoc for `testAAx()` will link directly to the axiom being tested.

After all tests have been run, the `tearDown()` method is executed, which takes care of releasing all test sets:

```
tearDown() {
    testSetT0 = null;
    testSetTn = null;
}
```

```
}
```

For convenience of this example, the pattern above has been idealised. The generator produces slightly different `setUp()` and test methods in cases where the data sets are not shared between all test methods. Consider the situation where a fixture has two test methods, `testA()` which uses the data set `testSetA` only, and `testB()` which uses `testSetB` only. Since `setUp()` is invoked before every test method, it is wasteful to initialise both `testSetA` and `testSetB` every time. Therefore, the generator will only put test sets which are shared among all test methods in `setUp()`. Local invocations to `createTestSet()` will be placed in the test methods for the other test sets.

### 11.6.2 Organising Generated Tests

The test generator produces two types of generated artifacts: the test set generator stubs which are meant to be fleshed out by the programmer, and the unit tests, which are meant to be executed through JUnit and never modified. For this reason, it is recommended that unit tests are generated into a separate package, such as `project.tests`, to place them separately from hand-maintained code. If there are additional, hand-written unit tests in `package.tests`, placing the generated tests into a separate package, such as `project.tests.generated`, is preferred. The test set generator stubs are clearly marked as editable in the generated comments and the test cases are marked as non-editable.

As previously stated, the generator can produce a suggested test suite based on a package or a project that lists all the tests requested in the same invocation of the test generator wizard. The purpose of these suites is to catalogue tests into categories and for the developer to be able to execute different categories at different times; some axioms may result in very long-running tests, others may not be interesting to test at each test suite execution and others still should be executed very frequently.

Both the test suites and test set generator stubs are freely editable. The generator will never overwrite these artifacts when they exist, even if the developer accidentally requests it.

### 11.6.3 Executing Tests

The current prototype tool supports two primary modes of test execution: from the command-line and through and interactive GUI.

The command-line mode is intended for integrating the tests into nightly build cycles, or other forms of continuous integration. We impose no restrictions on the management of test suites – the tool merely aids in producing suggested starting points – so varying degrees of testing may be decided on a per-project or per-build basis. By organising the generated tests into categories, it becomes easy to select which sets of axioms should be exercised at any given build.

The interactive mode reuses the JUnit framework. Once a test case (or test suite) has been generated, it can be immediately executed by the developer like any other JUnit test. Since there is a direct link in the Javadoc for every generated test method, it is trivial to understand which axiom is violated when a given test method fails.

JAxT does not provide any test coverage analysis itself, but existing solutions for JUnit, such as Cobertura [Cob] and NoUnit [NoU], are applicable.

## 11.7 Implementation

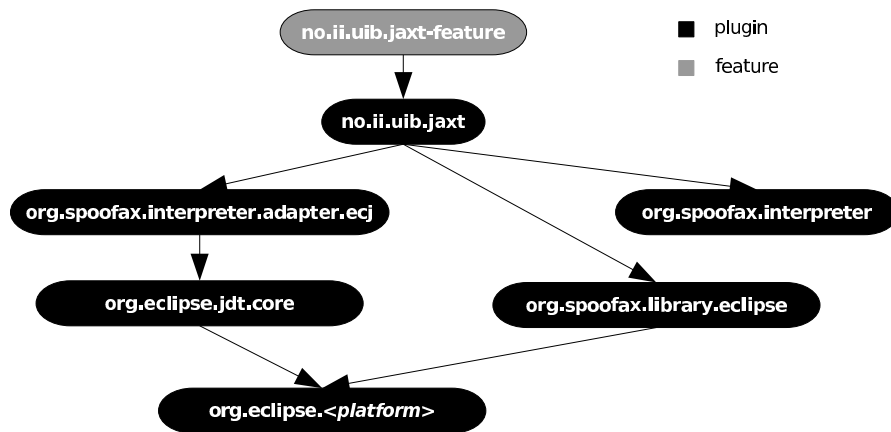


Figure 11.4: Components of the JAxT generator.

The JAxT tool is implemented as a plugin to the Eclipse development platform [Ecl]. It is divided into the component seen in Figure 11.4. The POM adapter technique, described in Chapter 4, and the Eclipse integration framework provided by Spoofox, as discussed in Chapter 9, were crucial for its construction.

The test generator wizard is very similar to the one provided by JUnit: The JUnit test generator is applied to a class  $C$  and produces a testing stub for each method in  $C$ , whereas JAxT, when applied to  $C$ , produces an immediately executable test for every axiom pertaining to  $C$ . The wizard interface collects all necessary user input. When the forms are complete, the control passes to a compiled MetaStratego transformlet which implements all the analysis and generation logic.

The script will use the Stratego/J FFI functionality to call into the Eclipse Compiler for Java and query for all necessary compilation units (.java and .class files). For compilation units that have source code, ASTs are extracted and adapted to terms using the POM adapter shown in Chapter 10. For “sourceless” compilation units, i.e. .class files, ECJ provides a read-only, high-level inspection interface. An additional POM adapter for this interface was implemented during the course of this work.

With this adapter, the JAxT transformation logic can traverse and inspect these compiler objects as well. This adapter provides a small signature for the notions of method and type bindings and by using it, many computations on the type hierarchy become very succinct and easy to express, e.g.:

```
ecj-all-supertypes-of =
  ?TypeBinding(_,_ ,_, superclass,superifaces,_ ,_)
  ; <collect-all(?TypeBinding(_,<id>,_ ,_,_,_ ,_)> [superclass | superifaces]
  ; map(!DottedName(<id>))
```

This three-line strategy will compute the transitive closure of all super classes and super interfaces (i.e. supertypes) of a given class (or interface), using the generic traversal strategy `collect-all` provided by the standard Stratego library. The strategy produces a list of `DottedName` terms, that contains the dotted names (fully qualified names) of the supertypes. Note the the super type hierarchy may in the general case be a directed acyclic graph and is rarely a tree. Even if the plain Stratego language is used, this presents no problems. Potential problems due to term building in graphs, discussed in more detail in Chapter 7, are avoided by only providing a read-only interface. Traversals will always terminate since there are no cycles.

Once the script has extracted the necessary information from the compilation units, it will will assemble ASTs for what will become the final JUnit test class and the test generator stub classes. Each of these ASTs will be written to a separate `.java` file. The Eclipse code formatter will be used to pretty-print the result. This ensures that the final result is in accordance with the user-configured settings for the Java project in which JAxT is applied.

## 11.8 Discussion and Related Work

The standard Java documentation, and that of many other software libraries, is rife with examples of formal requirements. These are typically not machine readable, nor machine checkable. As a result, these requirements are often inadvertently violated, often resulting in bugs which tend to be difficult to trace. The goal of program verification and validation techniques, including (unit) testing is to increase developer confidence in the correctness of their implementations. If a technique for formalising the library requirements was devised, and tests may be automatically generated from this formalisation, confidence that the library abstractions were used correctly would take a significant boost. This is what the JAxT, and other axiom-based techniques for test generation like JAX [SLA02], DAISTS [GMH81] and Daistish [HS96] do.

The testing approach sketched in this chapter is a continuation of the JAX tradition [SLA02] and is, in a sense, a combination of two techniques for ensuring robustness of software: test-driven development and algebraic specifications. The ideas of axioms and modular specifications are taken from algebraic specifications as

a way of succinctly describing *general* properties of an implementation, as opposed to the case-by-case approach normally advocated by TDD. Principles for integration into graphical development environments and the design of practical tools for test code generation and immediate execution of tests were inspired by TDD approaches. These principles allow the proposed method to bring instant feedback to the developer.

Any approach to (semi-)automatic test generation will eventually have to be implemented using some programming language. Experience from this case study suggest that constructing the generator in a language-general, domain-specific transformation language has several benefits compared to the author's previous experiences with implementing language processing using strictly general-purpose languages.

*Succinctness* – Compared to an implementation in a general programming programming, the transformation logic expressed in Stratego becomes compact and to the point (c.f. more detailed examples in Chapter 10). If care is taken to name the strategies and rules appropriately, most of the transformations read fairly well.

*Infrastructure Reuse* – The initial development of JAxT was very quick due to the reuse of the Eclipse Compiler infrastructure. Even if one accounts for the time taken to construct the additional POM (for type bindings), and the time spent constructing the AST POM itself, this was considerably less than the time necessary to construct a robust Java 1.5 parser from scratch, and much less than a stable type-checking front-end. The adapters were semi-automatically extracted from APIs in a matter of hours and, after a few more hours of plugging in the relevant FFI calls, the type checker was ready to be reused from within Stratego.

*Development Tools* – A weak point of most non-mainstream languages is the state of their development tools. This is also the case with Stratego which, for example, lacks an interactive debugger. The Spoofox development environment (Chapter 9) helped a lot, but additional work is required if the environment is to have a level of quality similar to that of the mainstream language environments.

*Code Templates* – Much of the generated code is composed from pre-defined templates. These are expressed using the abstract syntax of the ECJ, shown in Chapter 10. Both readability and maintainability would get a significant boost if these templates were expressed using concrete syntax, i.e. as concrete Java code. However, depending on the complexity of the templates, this would require the construction of a full Java 1.5 grammar which is embeddable into the transformation language.

## 11.9 Summary

This chapter presented a case-study of how the techniques proposed in Part III and Part IV of this dissertation are applicable to code analysis and interactive code generation. The study presented a tool-assisted approach for testing general properties of



---

classes and methods based on axioms and algebraic specifications expressed entirely in Java. It provided a detailed description of how desired program properties can be expressed as axioms written in an idiomatic Java style, including a rich and flexible mechanism for organising the axioms into composable modules (composable specifications). The proposed organisation mechanism is structured enough that the testing tool, JAxT, can automatically compute all axioms pertaining to a given class and generate JUnit test cases and test suites from the composition of these. By reexecuting JAxT periodically, the unit tests can trivially be kept in sync with the axioms, as these change.

The study also discussed design and implementation aspects of JAxT, illustrating how the techniques proposed in this dissertation may be applied in practise. The results of the study suggests that language-general, domain-specific transformation languages provide an attractive vehicle for expressing interactive language processing problems, but that additional tool support may be necessary before most programmers can be expected to benefit from the increased succinctness offered by these languages.