University of Bergen
Department of Informatics

Axiom Based Testing in Java

Author: Mads Bårvåg Nesse

Supervisor: Magne Haveraaen

Monday 3rd June, 2024

Abstract

Software plays a crucial role in today's society. Making sure that software is reliable is therefore very important. To make reliable software it is important to test it. Axiom-based testing gives us a way of specifying properties of programs and testing them against generated data. In this thesis we look at previous work done for implementing axiom-based testing, mainly JAxT by Karl Trygve Kalleberg and Magne Haveraaen. We present our work on creating an axiom-based testing framework for Java. The result of our work is a prototype that we have called Jaxioms.

After introducing relevant background we look at the tools we have used. Then we go over main parts of our framework, and an overview of the implementation created. We end by discussing limitations and future work that can be done for Jaxioms.

Acknowledgements

First, I would like to express gratitude to my supervisor, Magne Haveraaen, for giving me good guidance and support throughout my master's. I would also like to thank the rest of the Bergen Language Design Laboratory (BLDL) for their feedback.

Thank you also to my friends and family, and especially to Katrine for supporting me throughout this year.

Finally, thank you to everyone in the algorithm study hall and the coffee machine on the fourth floor for keeping me entertained and caffeinated.

`Mads Bårvåg Nesse`

Monday 3rd June, 2024

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Testing software has always been important. However, coming up with cases to test requires significant work. It can also be difficult to think of everything that can go wrong. Using axiom-based testing gives us a way of specifying properties of programs and then testing automatically that these properties hold. This way we do not have to manually find test cases that test meaningful parts of the program. Previous work done at Bergen Language Design Laboratory (BLDL)[7] showcase tools for axiom-based testing in both C++[4, 58] and Java[30, 29]. The tool for Java, JAxT is dependent on using the Eclipse IDE, and therefore, we present this work on creating a new tool for axiom-based testing in Java that is independent of IDEs.

We have developed a unit testing framework for Java. The framework enables axiom-based testing within Java programs. We have looked at other works relating to axiom-based testing to see what can be learned and how we can facilitate developers to integrate axioms into their development flow easily.

For Java a bunch of properties are defined for the standard API. The Liskov Substitution Principle[40] states that all subclasses of a class should conform to the properties of the class. In Java, for example Object has several properties stated. All classes in Java inherit from the Object class, meaning that any class should uphold these properties. However, it is up to the developer to remember all properties that should be upheld and make sure that they are satisfied. This can often cause mismatch from documentation to the actual program as the developers have not necessarily read the documentation of Object. And if they had, it can be hard to remember all properties that should be satisfied. Using axiom-based testing we can allow classes to inherit specifications making sure all relevant properties are tested for a class.

Another benefit we gain from axiom-based testing is extensive documentation. If all relevant properties are stated as axioms along with the classes and functions, it is easy for developers to read the properties of the programs they are using or developing. This can be a nice addition to the Javadoc.

The motivations behind creating a framework for axiom-based testing is many. We want to test specifications that are inherited through the Java hierarchy. We want it to be easy to create test data. We want to give specifications to programs in a formal manner, but keep it easy to use and understand. All in all we want to be more confident that our programs work as intended. Axiom-based testing is a good supplement to regular example-based unit testing.

When starting the development of our framework, two main questions arose.

RQ1. How can we make it easy for developers to integrate axioms into the Java software development process?

RQ2. How can we apply what was done in JAxT to the context of modern Java?

The rest of this thesis is structured as follows: In chapter two we go into theoretical background, chapter three details some of the tools we have used in our framework, chapter four gives an overview of our framework Jaxioms, chapter five gives some details to key parts of the implementation, chapter six discusses future work and limitations with our framework, and chapter seven concludes this thesis.

Chapter 2

Theoretical Background

2.1 Algebraic specifications

One way of defining software is through algebraic specifications. When giving specifications of software algebraically, you formally specify system behaviour. When defining software formally, algebraic structures can be used to give the types a basis in mathematics. An algebraic structure consists of a set called a carrier set. The structure also has operations on the set and requirements for the operations and types. The structures inherit behavior between each other.

In Figure 2.1 we visualize some algebraic structures, inheriting from each other. The root structure in the figure is called a Magma. We define a Magma as a set with a binary operation. The binary operation takes in two elements from the set and returns one element from the set. It is also required that the set is closed under the binary operation, meaning that the operation does not return any element that is not in the set. The set of the structure can be for example a set of numbers like the natural numbers, $\mathbb{N}$, and the binary operation can be for example addition or multiplication of natural numbers. When adding different specifications to the structure, we will get a new structure. For example, adding the requirement that the binary operation is associative, gives us a Semigroup. Furthermore we can add an identity element, together with the property that the binary operation on an element and the identity element gives us the original element, to get a Monoid.

$$\begin{tikzcd}
 & \text{Magma} \arrow[dl, "\text{divisibility}"'] \arrow[d, "\text{identity}"] \arrow[dr, "\text{associativity}"] & \\
\text{Quasigroup} \arrow[d, "\text{identity}"'] \arrow[dr, "\text{associativity}"] & \text{Unital Magma} \arrow[dl, "\text{invertibility}"'] \arrow[dr, "\text{associativity}"] & \text{Semigroup} \arrow[dl, "\text{divisibility}"'] \arrow[d, "\text{identity}"] \\
\text{Loop} \arrow[dr, "\text{associativity}"'] & \text{Inverse Semigroup} \arrow[d, "\text{identity}"] & \text{Monoid} \arrow[dl, "\text{invertibility}"] \\
 & \text{Group} &
\end{tikzcd}$$

Figure 2.1: Algebraic structures from Magma to Group, Figure is taken from [1]

When defining programs through algebraic specifications we use interfaces and algebras. An interface is often also referred to as a signature. Interfaces are used to specify types and mathematical operations on the types. An interface $I = \langle S, F\rangle$ consists of a set of sorts S and a set of functions F. An example of an interface using mathematical notation can be seen in Figure 2.2. In the example, we define a signature Magma, which has a set of one sort, T, and a set of one function, binop, on the sort.

$$\begin{aligned} \text{Magma} &= \langle\{T\}, & (2.1)\\ &\quad \{\text{binop} : T, T \to T\}\rangle & (2.2)\end{aligned}$$

Figure 2.2: Example interface for defining a set of one sort T, and one binary operation binop

When an interface has been defined we can see that it does not have any functionality.

In order to define behaviour we can specify *algebras*, also called *models*. Algebras gives semantics to interfaces. An algebra A for an interface $I = \langle S, F \rangle$ needs to define a set, called the *carrier set* for each sort in S and a *total function* for each function in F. An example of an algebra for the interface Magma can be seen in figure 2.3

$$[\![T]\!]_{Nat} = \mathbb{N} \tag{2.3}$$
$$[\![\text{binop}]\!]_{Nat} = a, b \mapsto a + b \tag{2.4}$$

Figure 2.3: Example algebra Nat for interface Magma defining natural numbers and basic operations on them, this algebra is a Magma, closure axiom is implied.

We can also add specifications to algebras, called axioms. An axiom is an expression on the types and operations of an algebra. Using mathematical terms we can for example define the associativity axiom that extends a Magma to a Semigroup as illustrated in Figure 2.4.

$$\forall a, b, c \in T : \tag{2.5}$$
$$\text{binop}(a, \text{binop}(b, c)) == \text{binop}(\text{binop}(a, b), c) \tag{2.6}$$

Figure 2.4: Associativity axiom, making it a Semigroup (Magma with associativity)

An alternative and perhaps more familiar way of defining interfaces and algebras is by using programming notation. The syntax we use here is similar to the Magnolia programming language, see Section 2.1.1. In Figure 2.5, you can see the same interface and the same algebra as in figures 2.2 and 2.3, but using programming notation instead of mathematical notation.

```
signature Magma {
    type T

    function binop(a : T, b : T) : T
}
concept Nat {
    type Int

    function add(a : Int, b : Int) = a + b
}
satisfaction NatModelsEx =
    Nat models Magma[T => Int, binop => add]
```

Figure 2.5: Definition of a signature (interface) and a concept (algebra) using Magnolia style syntax

2.1.1 Magnolia programming language

The Bergen Language Design Laboratory, BLDL[9] is working on an experimental programming language called Magnolia[8, 11]. The language is designed for generic programming. Magnolia also allows for writing programs that are similar to algebraic specifications. A Magnolia program consists of two "layers": the specification layer and the program layer.

The specification layer consists of signatures and concepts. A signature in Magnolia is similar to an interface, as defined in the previous section. Signatures consist of types and operations. Concepts allow us to add specifications to signatures through axioms. An example of how one can use the modules of the specification layer to define algebraic structures can be seen in Figure 2.6.

In the program layer, we have implementations and programs. In an implementation, you can add the same declarations as in a signature but also define generic implementations. A program is an implementation where the developer defines specific implementations for all generic operations and types.

One thing about Magnolia that makes it quite unique is that they provide no primitives in the language. Thus, when creating programs, you must match generic implementations with an implementation from a host language. This genericity they have called "genericity by host language" [6, p.3].

```
signature Magma = {
    type T;
    function binop(a: T, b: T): T;
}
concept Semigroup = {
    use Magma;
    axiom associativeBop(a: T, b: T, c: T) {
        assert binop(binop(a,b),c) == binop(a,binop(b,c));
    }
}
concept Monoid = {
    use Semigroup;
    function identity() : T;
    axiom identityIsNeutral(a : T) {
        assert a == binop(a, identity());
        assert binop(a, identity()) == a;
    }
}
```

Figure 2.6: The specification layer of a Magnolia program, specifying a Monoid through inheriting from a Semigroup and Magma

When creating implementations for the program layer, Magnolia uses a technique called rewriting. Rewriting is used to match the names from the specifications with names of functions in the implementations. An example of the program layer can be seen in 2.7. The examples here are based on the examples given in [6, p.7].

```
implementation PythonMonoid = external Python somelib.some_impl {
    use Monoid[T => int, binop => add, identity => zero]
}

program ExampleProgram = {
    use PythonMonoid;

    //functions and procedures that can use the python implementations
}

satisfaction ExampleSatisfiesMonoid = ExampleProgram models Monoid[T
    ↪ => int, binop => add, identity => zero]
```

Figure 2.7: The program layer of a Magnolia program, giving Monoid an implementation through an external python implementation somelib.some_impl is a custom python program wrapping around the python implementations used.

2.2 Object-Oriented Programming

A popular paradigm of programming is object-oriented programming. Object-oriented programming is, as the name suggests, programming revolving around objects. When programming object-oriented, you write classes and then instantiate the classes. Instances of classes are referred to as objects. A class typically consists of fields, which are special variables that belong to the class, and methods, which are functions that modify or access the fields of the class.

Encapsulation is a key concept of object-oriented programming. The goal is to encapsulate data and methods that pertain to that class. In other words, you hide values that should not be modified unexpectedly. This is done in order to have control over how the data is accessed. We add private modifiers when data or methods should not be accessed and public modifiers when we allow access. An example is if we have a class called `Position`, see Figure 2.8, which is a position in 2D space. The class has two integer fields representing an x-value and a y-value. We want the values to be between 0 and 8 for both fields. If the fields are open for modification, people might change the value to be outside our range. Instead, we should provide a public setter method that ensures the new value is within the range.

```java
public class Position {

    public int x; //should be private, not public
    public int y;

    public Position(int x, int y) {
        this.x = x % 8;
        this.y = y % 8;
    }

    //when modifying x and y use setters
    public void setX(int x) {
        this.x = x % 8;
    }
    ...
}
```

Figure 2.8: A simple position class representing a position in 2D space

Another key concept of object-oriented programming is inheritance. Classes can inherit from one or multiple parent classes. This promotes reusability and allows for specialization because you can override methods to differentiate behavior for child classes from their parent.

With the inheritance of object-oriented programming comes another concept, polymorphism. This concept revolves around being able to use a superclass or interface when code is called from a subclass. This is useful when multiple subclasses are all inheriting from superclass A. The classes will then all have the methods from A, but some of them might override it with specific behaviour. As a programmer, you can then treat them all as type A and call the methods without being concerned with which implementation should be used.

There is also a principle of Object-oriented programming called the Liskov substitution principle. Introduced by Liskov in the KeyNote "Data Abstraction and Hierarchy"[40]. The principle states that an instance of a subclass B should satisfy all properties of class A if it is a class A subclass. This means that if a program depends on an object of type A, supplying an object of type B would not break the program's functionality.

2.3 The Java Programming Language

The Java language was originally released by Sun Microsystems in 1995[50]. Oracle has since taken over as developer of Java[14]. Java is a "write-once, run anywhere" programming language, meaning that code you have written can be run on any machine running the Java Virtual Machine. Java is designed around the Liskov substitution principle; all subclasses in Java should inherit properties from their parent classes. Even though Java came out in 1995, it is still (2023) the fourth most used programming language on the open-source platform GitHub[39]. Yet it is worth mentioning that it has decreased in popularity, as can be seen in the GitHub report from 2022[55]. According to the report, it was number two from 2014 to 2018 and number three from 2019 to 2022. Another language running on the Java Virtual Machine, Kotlin[34], is becoming popular. In the 2022 report[55], it had an increase in popularity of 22.9%. Kotlin was the 12th most popular programming language in 2023[39].

2.3.1 Properties in the Java Language API

In the Java language API[20] multiple properties are stated for different classes and methods. This was identified in the work on JAxT[30, 7]. Object has several properties specified for both its `equals` and `hashCode` methods. The following properties should all hold for the `equals` method of all subclasses of `Object`. It is reflexive, in other words `x.equals(x)` should return true. Equals is also symmetric, meaning that `x.equals(y)=> y.equals(x)`. It should also be transitive, if `x.equals(y)` and `y.equals(z)` then `x.equals(z)`. Furthermore, it should be consistent, so multiple calls to `x.equals(y)` should always yield the same result. The last property stated for `Object.equals` is that `x.equals(null)` should always return false. For the `hashCode` method, it is stated that it should be consistent; during an execution of a program, `hashCode` should return the same result consistently. Also `hashCode` is congruent on `equals` meaning that if they are equal according to the `equals` method, then `hashCode` should produce the same integer for both of them.

Another place we can find axiomatic style properties in the API is in Comparable[16]. One requirement is that the `compareTo` method should be consistent with the signum method from `java.lang.Integer`[18], i.e. that `signum(x.compareTo(y))== -signum(y.compareTo(x))` for all x and y. The compareTo method should also be transitive, i.e. `(x.compareTo(y)>` `↪ 0 && y.compareTo(z)> 0)` implies that `x.compareTo(z)> 0`.

These properties are all nice to have; however, while the documentation states that these should hold, the programmer implementing a class will be responsible for ensuring that they are actually held when they override the methods. All classes in Java inherit from the class `java.lang.Object`[46], meaning that all classes in Java should follow the `Object` properties.

2.4 Unit Testing

There are multiple ways of testing that computer programs work as intended. One of the most common techniques of testing software is unit testing. Unit tests are small programs that execute a portion of the program under test, checking expected behavior or testing that unexpected values produce errors. These tests are a vital part of software development. Rerunning them when the code is updated can indicate whether the program works as expected after introducing changes.

A difficult aspect of unit testing is finding good examples of values that produce expected results and those that should produce errors. It is especially important to test the values that are most likely to break the program, this could be edge-case values that rarely occur.

2.4.1 SUnit

In 1996, Kent Beck created a framework called SUnit for the language Smalltalk[5]. The original paper described the testing strategy and the SUnit framework. This framework was the ancestor of so-called XUnit testing frameworks, which have been adapted for multiple languages.

2.4.2 JUnit

On an airplane to the 1997 OOPSLA conference, Kent Beck and Erich Gamma created JUnit. It is built on the same principles as SUnit but for Java instead of Smalltalk. After JUnit gained popularity, several other frameworks for different languages arose, causing the term XUnit to refer to any framework deriving from the principles of SUnit[26]. Two of these ports are CppUnit[54] for C++ and NUnit for .NET[10]

Since JUnit was created in 1997, it has undergone several overhauls, but the main ideas remain the same. The current major release of JUnit, which we are building on, is JUnit 5[37]. According to a survey done in 2023 by Jetbrains [35], see Figure 2.9, JUnit is the most used unit testing framework for Java, with 84% of users responding that they use JUnit for unit testing.

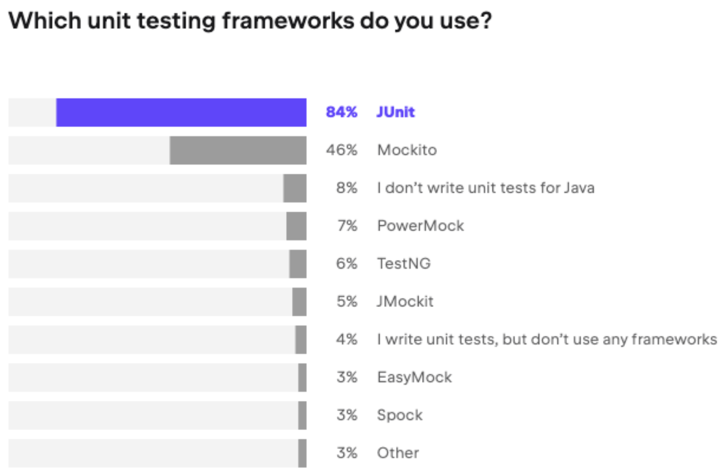


Figure 2.9: Part of Jetbrains 2023 developers survey[35]

2.5 Axiom-based testing

Axiom-based testing allows developers to specify algebraic-style axioms. Axiom-based testing provides developers with a systematic way of specifying the behavior of a program using properties. This way, the developer will only specify the properties that should hold instead of coming up with all different test cases that can go wrong. Data can then be generated based on the properties to try and test as many cases as possible.

In different works, Axiom-based testing is also referred to as property-based testing. Axiom-based testing can be supported either on the language level, such as for Magnolia, see Section 2.1.1, where they provide a way of specifying axioms, which you can then use to test, or it can be supported through external libraries or frameworks for programming languages. The following sections describe some common tools for Axiom-based testing.

2.5.1 QuickCheck

One of the first frameworks to popularize property-based testing was QuickCheck[12]. QuickCheck was created by Koen Claessen and John Hughes from Chalmers University of Technology, as a framework for the language Haskell. They proposed a way of supplying testable criteria through formal specification, and they designed a way of specifying properties in Haskell code.

A standard example for QuickCheck is checking that the reverse of a list returns the reverse. When we have a reverse function, see Figure 2.10, we can write a property

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' [x] = [x]
reverse' (x:xs) = reverse' xs ++ [x]
```

Figure 2.10: A standard reverse function in haskell.

```
prop_reverse :: [Int] -> Bool
prop_reverse xs = reverse' (reverse' xs) == xs
```

Figure 2.11: A property that checks that reverse function 2.10 returns the reverse of a list of Ints.

which is a function that should take as input the arguments of the function in Figure 2.11. While our reverse function can take in any type in the list, we need to supply a specific type for QuickCheck to generate values. In Figures 2.12 and 2.13 we show a reverse function that does not reverse the list as it should, and hence QuickCheck should find an error. Running the example by first importing QuickCheck and then running the standard quickCheck function gives the result in Figure 2.14. As can be seen, the failing function gives the result `Failed ! Falsified ( after 3 tests and 3 shrinks )`. This means it ran 3 tests before it found a counter-example that shows the property is not correct. It also performed three shrinks, which is a QuickCheck technique for reducing the failing counter-example to its simplest form.

2.5.2 JAxT

A framework for integrating axioms into Java exists already. This framework, named Java Axiom Testing - JAxT [30], was developed at Bergen Language Design Laboratory by Karl Trygve Kalleberg and Magne Haveraaen. JAxT is a plugin created for the Integrated Development Environment (IDE) Eclipse.

When using JAxT, the developer provides axioms in accompanying classes. They then use Eclipse to generate JUnit-style test classes, which is similar to generating normal JUnit test classes in Eclipse, by clicking on the JAxT specific “generate test class” option.

```
reverseFailing :: [a] -> [a]
reverseFailing [] = []
reverseFailing [x] = [x]
reverseFailing (x:xs) = reverseFailing xs ++ [x,x]
```

Figure 2.12: A reverse function in haskell that does not return the reverse of a list.

```
prop_reverseFailing :: [Int] -> Bool
prop_reverseFailing xs = reverseFailing (reverseFailing xs) == xs
```

Figure 2.13: A property that checks the failing reverse function 2.12 returns the reverse of a list of Ints

```
ghci> import Test.QuickCheck
ghci> quickCheck prop_reverse
+++ OK, passed 100 tests.
ghci> quickCheck prop_reverseFailing
*** Failed! Falsified (after 3 tests and 3 shrinks):
[0,0]
```

Figure 2.14: Running quickcheck on the two properties we defined in Figures 2.11 and 2.13

JAxT also allows for inheriting specifications, and provide axioms that were identified from the Java API, see Section 2.3.1.

JAxT is used to specify axioms for class X in a separate file called XAxioms.java. It also requires the user to specify a generator for class X. JAxT can then do one of two things: either check a specific class, X, in isolation or check a subclass hierarchy for a class or interface.

The user also supplies a test data generator that should give random instances of the class under test. JAxT has functionality for creating a test data generator template that the user can fill out.

2.5.3 Catsfoot

For C++11 there was a proposal which added concepts to C++. The concepts were to provide a way of describing requirements on types, similar to concepts in Magnolia, see Sectio 2.1.1. One of the things introduced in the concepts was the ability to specify axioms on concepts [25]. However, due to it being too complex, this proposal was scrapped [56].

In the paper Testing with Axioms in C++ 2011 [4], Bagge et al. introduced a framework called Catsfoot[58] that added concepts in a similar manner to that in the proposal. They also added support for checking that implementations of concepts adhere to the axioms provided by testing them on random data. The paper also discusses having a library that supplies templates for algebraic structures or common concepts such as indexable, searchable, and sorted. This can be beneficial as it provides you with “free tests”, since you can inherit tests from concepts, and then the tool can be used to generate tests for your specific use case.

2.5.4 Axiom Based Testing for Fun and Pedagogy

In his paper "Axiom Based Testing for Fun and Pedagogy"[29], Haveraaen advocates using Axiom-based testing to teach new students programming. He claims that the intuition that students acquire over 12 years of school has a mismatch with the technicalities of programming. Students are taught numbers in the mathematical sense where number types such as natural numbers $\mathbb{N}$ and integers $\mathbb{Z}$ are sets of infinite size. However, when it comes to computer languages such as Java, the numbers are represented in a different way. They are bound to computer bits. Therefore, by utilizing axiom-based testing, the gap between their intuition and how computers work can be bridged.

As argued by Haveraaen, Axiom-based testing is a nice way to help programmers learn to program without breaking properties defined for the classes they are working on. It can also aid in ensuring that programmers overriding methods do not break any contracts that these methods should uphold. Such as for example the properties of `Object.equals()` as described in Section 2.3.

2.5.5 Parameterized Tests

Parameterized tests in JUnit[38] offer a lot of the features from property-based testing. The developer specifies that a given test method should be a parameterized test using annotations. They also need to specify an argument provider that provides arguments that will be used to test the method. In the example in Figure 2.15 there are two different ways of defining arguments, the first being as an argument to the `@ValueSource` annotation. The second way we show is by using a method that returns a stream of arguments.

```java
public class ParameterisedExample {
    @ParameterizedTest
    @ValueSource(ints = {2,4,6,8,10,29064}) //providing arguments
       ↪ through annotation argument
    void testIsEvenMethod(int a) {
        assertTrue(isEven(a));
    }

    @ParameterizedTest
    @MethodSource("intProvider")
    void testIsEvenMethodWithMethodSource(int a) {
        assertFalse(isEven(a));
    }

    // providing arguments through method
    public static Stream<Arguments> intProvider() {
        return Stream.of(
                Arguments.of(1),
                Arguments.of(3),
                Arguments.of(5),
                Arguments.of(7),
                Arguments.of(9),
                Arguments.of(2905)
        );
    }

    public static boolean isEven(int a) {
        return a % 2 == 0;
    }
}
```

Figure 2.15: Example of JUnit style parameterized tests

2.5.6 Categories of properties

Hypothesis[42] is a library that allows for property-based testing in Python. Hypothesis uses strategies to generate data that you then use in properties, which are declared similarly to QuickCheck.

In the paper “How Developers Implement Property-Based Tests”[13], they look at how different repositories using Hypothesis use property-based testing. The paper looks at a total of ten categories of Property-based testing, eight of which were identified in a blog post, directed towards functional programming in F#[24] by Scott Wlaschin[60]. The remaining two properties are from the paper “Falsify your Software: validating scientific code with property-based testing”[28].

The properties from the blog post are “Different paths, same destination”, “There and back again”, “Some things never change”, “The more things change, the more they stay the same”, “Solve a smaller problem first”, “Hard to prove, easy to verify”, “The test oracle”, and “Model-based testing”. The two properties that they used from [28] were “Outputs within expected bounds” and “Metamorphic properties”. Following is a description of each of the different categories.

Different paths, same destination

This category relates to the properties stating that the order of application should not matter. From mathematics, this would translate to commutative functions, see Figure 2.16.

```
concept CommutativeExample {
    type T;
    add(a : T, b : T) : T
    axiom addIsCommutative (T a, T b) {
        assert add(a,b) == add(b,a)
    }
}
```

Figure 2.16: An example of a property in the category Different paths, same destination

There and back again

In this categories properties state that applying the inverse function of a function takes you back to the original value. An example of a property in this category is a decode function that is the inverse of an encode function, See Figure 2.17.

Figure 2.17: An example of a property in the category There and back again

```
concept InverseFunction {
    type T;
    encode(a : T) : T
    decode(a : T) : T
    axiom invIsInverse(a : T) {
        assert a == decode(encode(a))
    }
}
```

Some things never change

For this category, the properties describe some invariant that is preserved after some kind of transformation. An example of this is that an element is still in a list after sorting the list, or that the sorted list is the same size as before sorting, see Figure 2.18.

```
concept ListPropertiesStayTheSame {
    type List, Int, E;
    sort(a : List) : List
    size(a : List) : Int
    contains(a : List, e : E) : boolean

    axiom sortKeepsSize(a : List) {
        assert size(a) == size(sort(a));
    }
    axiom sortKeepsElementsInList(a : List, e : E) {
        assert contains(a, e) => contains(sort(a), e);
    }
}
```

Figure 2.18: Two examples of properties in the category Some things never change

The more things change, the more they stay the same

Here the properties describe functions that do the same thing regardless of how many times the function is applied. This is also referred to as idempotence. A notable example in this category is that a filter of a collection, should yield the same filtered collection regardless of how many times the filter was applied, see Figure 2.19.

```
concept FilterList {
    type List, C;
    filter(a : List, condition : C) : List
    axiom filterTwiceSameAsOnce(a : List, c : C) {
        assert filter(filter(a, c), c) == filter(a, c)
    }
}
```

Figure 2.19: Example of a property stating that filtering a list twice is the same as filtering once

Solve a smaller problem first

In this category the properties divide the problem up by recursively defining a property that should for example hold for the first element of the list, and that the rest of the list should also have the same property. An example can be that for a list that should only contain positive numbers, the first element should be a positive number, and the rest of the list should have the same property, see Figure 2.20.

```
concept ListShouldBePositive {
    type List, Int;

    isPositive(i : Int) : boolean;
    head(l : List) : Int;
    tail(l : List) : List;

    axiom listContainsOnlyPositives(l : List) {
        assert allPositive(l);
    }
    //incorrect to have this in a concept in Magnolia, but have it
       ↪ like that for illustrating
    function allPositive(l : List) : boolean {
        return isPositive(head(l)) && allPositive(tail(l));
    }

}
```

Figure 2.20: An example of a property stating that a list should only have positive numbers

Hard to prove, easy to verify

The category Hard to prove, easy to verify contains properties that is computationally heavy to prove, but checking if a solution is correct is relatively easy, see Figure 2.21.

```
concept Maze {
    type Maze, Path, Node;
    findPath(m : Maze, start : Node, goal : Node);
    contains(p : Path, node : Node);

    axiom findPathReturnsPathThroughMaze(m : Maze, start : Node goal :
       ↪ Node) {
        p : Path = findPath(m, start, goal);
        assert contains(p, start) && contains(p, goal);
    }


}
```

Figure 2.21: Example of a property that checks something that is easy to verify i.e. checking if a collection contains a node

The Test Oracle

The two final categories described in [60] are Test Oracle and Model-based testing. They are quite similar in the way they work. For test oracle you use an alternative version of the algorithm that is to be tested to verify the result. An example of this could be that you want to check that an optimization of a sorting algorithm still sorts the list, so you use a known implementation of sorting to check that it sorts the list.

Model-based testing

For Model-based testing you create some model that functions as a bare minimum implementation of the program you want to test and apply all the same functions to the model. At the end you verify that the state of the model and the system under test are the same.

Outputs within expected bounds

The first of the categories described in "Falsify your Software: validating scientific code with property-based testing"[28] is "Outputs within expected bounds". This category is pretty self explanatory, the property states that some value should be within a bound. For instance, a price or length function should always return a positive value. This can also be called data invariants.

Metamorphic properties

Properties in this category don't directly say something about the result because they don't necessarily know the result. Instead, they state that given an input, the output should behave in a certain way. Using filter as an example, like in "The more things change, the more they stay the same", we can say that filtering a list should return a list whose length is smaller than or equal to the original list. We don't know what, if anything, will be filtered away, we know that filtering a list should not increase the length of the list, see Figure 2.22.

```
concept FilterList {
    type List, C;
    length(a : List) : Int
    filter(a : List, condition : C) : List
    axiom filterDoesNotIncreaseList(a : List, c : C) {
        assert length(filter(a, c)) == length(a)
    }
}
```

Figure 2.22: Example of a metamorphic property, filtering a list does not increase the size of the list

Property Category	No of test cases	Percentage
Different paths, same destination	0	0%
There and back again	32	37.2%
Some things never change	6	6.9%
The more things change, the more they stay the same	0	0.0%
Solve a smaller problem first	0	0.0%
Hard to prove, easy to verify	0	0.0%
Test oracle	29	33.7 %
Model-based testing	3	3.4%
Outputs within expected bounds	5	5.8%
Metamorphic properties	3	3.4%
Other	8	9.3%

Table 2.1: The distribution of test cases for each category, from [13].

Category usage

Findings from "How Developers Implement Property-Based Tests"[13], see Table 2.1, indicate that most of these categories were not very widely used for the repositories they looked at in the paper. For the dataset, they chose the top 30 repositories using Hypothesis and again narrowed the study to look at 86 property-based tests from these repositories. Of these test cases, 32 fall into the category "There and back again", and 29 of them in the category "Test oracle", meaning these two categories alone make up over 70% of the total tests. Meanwhile, the categories "Different paths, same destination", "The more things change, the more they stay the same", "Solve a smaller problem first", and "Hard to prove easy to verify" all have 0 cases in the data set looked at. They attribute the lack of test cases in these categories to developers not knowing what properties in the unused categories are. The high number in the first two could be due to their familiarity with developers, and thus, they find them easier to implement.

Chapter 3

Tools Background

To make it easy for developers to integrate axioms into the development process, we wanted the step from not writing axioms to writing them to be as little as possible. The idea is that to start writing axioms, the developer adds the framework and starts writing them with minimal additional steps. It should also integrate with the current way they do software testing. This is so we do not add unnecessary overhead regarding the number of different tools they are using.

The idea for our Axiom-based testing tool is therefore a framework with the following specifications:

- Should work regardless of IDE
- Should generate JUnit test classes based on axiomatic specifications
- Should utilize Parameterized testing from JUnit
- Should automate the process so that minimal manual work is required
- Should integrate with some data generation

It is also desirable to have as few dependencies on other third-party software as possible. This ensures that we are not reliant on other people to keep their code maintained. However, removing all dependencies on third-party software proved hard but might be achieved with some additional work.

We tried to see if we could implement Axiom-based testing as an annotation processor, instead of relying on Eclipse. Utilizing an annotation processor gives us the benefit of being able to run the framework as part of the compilation process. This ensures that the test classes we generate as part of annotation processing can then be run as part of the compilation.

Following is a description of how we chose the different tools we use for our framework, Jaxioms.

3.1 IDE Independence

Our main goal was to not depend on a specific IDE. The previous tool created by Haveraaen and Kalleberg, JAxT, relied on Eclipse's plugin ecosystem for generating test classes. Optimally, we think the user should not depend on the Eclipse IDE or any other IDE when using the framework.

In a survey from 2010 by Eclipse[23], Eclipse had the majority of the market share for IDEs with a 53.7% share, compared to 1.2% from JetBrains'[33] IDE IntelliJ. This survey might have some bias, seeing that it was Eclipse themselves that made the survey. A more recent survey[35] found that Eclipse is no longer as popular an IDE as it used to be, as IntelliJ was the most popular in the survey. However, as discussed in the survey, this survey also suffers from bias because it was done by JetBrains, and thus, respondents are likely to be JetBrains users. Findings from a newer survey done by Stack Overflow[51] in 2023 also confirm that IntelliJ has the most users. This survey reported that 26.8% of respondents use IntelliJ from JetBrains, while only 9.9% use Eclipse.

Seeing that the trend of which IDE developers are using to develop Java is fluctuating and might be something else entirely in 10 years, we figured that not relying on an IDE at all is the way to go forward. What we also gain from this is that if we want developers to use our framework, they should be able to adapt it to their current workflow.

3.2 Maven

Maven[3] is a build system for Java created by Apache. In the Developer Ecosystem survey done by JetBrains in 2023[35], 74% of developers said they use Maven. To use Maven developers specify a build file "pom.xml" which contains information for Maven to build the Java project. The build file contains information like compiler plugins, dependencies on other maven projects, and naming and versioning for the project. When Maven is run the dependencies are downloaded as Java Archives (JARs).

For our prototype, we require that users have Maven installed. We wanted to use Maven because it is the most used build system for Java. We have only been using Maven version 3.6.3 and are aware that some issues arise with using other versions, which is discussed further in Chapter 6.

3.3 Choosing testing framework

Another aspect to consider is how developers write tests. As mentioned in Section 2.4.2, JUnit is the most popular testing framework for Java[35]. How can we write JUnit tests in a modern Java context? Parameterized tests are a tool for testing in JUnit that offers many of the same benefits as property-based testing. With Parameterized tests, users can specify properties that should hold and then test them with specified arguments. The tests can also have a method specified that supplies arguments to use.

Building upon an existing testing framework gives us the benefit of utilizing their way of reporting test results. We generate our axioms as JUnit parameterized tests and then leave it up to the build tool to discover the tests. This discovery is done the same way as discovering other JUnit tests.

An argument could be made that we should not rely on a testing framework such as JUnit either, as that creates a new dependency that might deter some developers. However, removing our dependency on JUnit would require some additional work. For our prototype, it is enough to depend on the most used framework, as that will cater to most developers.

3.4 JavaParser

JavaParser[32] is an open-source library for interacting with Java source code. It allows the user to parse source code as an AST. When the source code has been parsed, the library provides the user with a mechanism for navigating the tree, as well as the ability to manipulate the tree. The mechanism for manipulating the tree can be used for code generation, and JavaParser is, therefore, often used as part of refactoring tools.

```java
package org.example;

import java.util.function.Function;

public class ParserExample {
    public String someStringMethod() {
        return "This is a string";
    }
    public boolean someBooleanMethod() {
        return true;
    }
    public Function<Integer,String> someFunctionMethod() {
        return (Integer i) -> "This is a function";
    }
}
```

Figure 3.2: Example file org.example.ParserExample.java

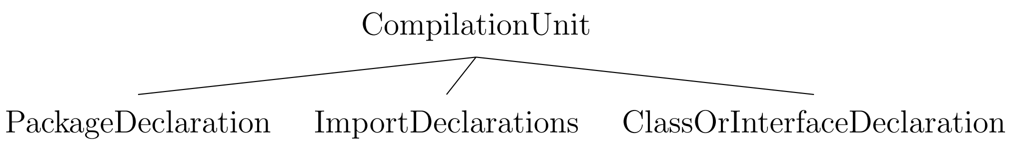


Figure 3.1: The root node, plus the first layer of a typical JavaParser AST

When parsing Java source code using JavaParser, an abstract syntax tree is returned. The root node of the AST will typically be a CompilationUnit when using the generic parse methods. Roughly speaking a CompilationUnit can be seen as the file in which Java code lives. Thus the CompilationUnit will have three types of children, that we can also recognize Java source code files to have. It will have a package declaration, some import statements, and a class or interface declaration. The root node plus the first layer of child nodes of a typical Javaparser AST can be seen in Figure 3.1.

In Figure 3.2 we can see an example of a class. When parsing this using JavaParser we will get a CompilationUnit representing the file `ParserExample.java`, which will have three child nodes. The first child node will be of type PackageDeclaration which again will have a child node that is of type `Name`. We then also have another node representing the import of the `Function` interface. The import declaration has a name telling us what is being imported, and two additional properties, `isStatic` and `isAsterisk`, telling us whether it is a static import, and whether it is an import for everything in a package.

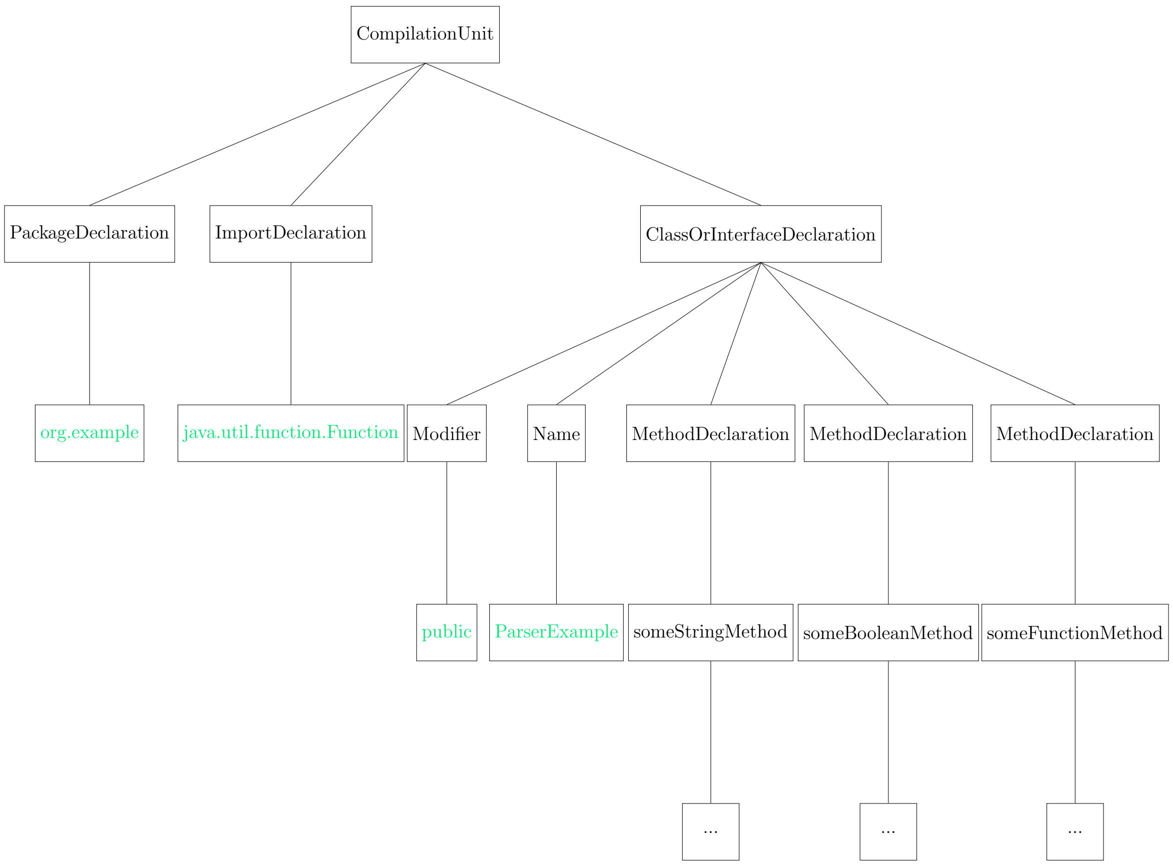


Figure 3.3: Result of parsing the class in Figure 3.2. Green nodes are leaves with no children

The third and most important child node we have is the `ClassOrInterfaceDeclaration` for `class ParserExample`. The `ClassOrInterfaceDeclaration` node will again have five children. Three of them will be nodes for each of the three methods, and we will also have two leaf nodes, one telling us the access modifier of the class, which will be public, and the other representing the name of the class. When parsing the example file we get the AST that can be seen in Figure 3.3.

An important thing to note with Javaparser is that it is a parser, not a compiler, and thus offers no guarantee that parsed code will compile [53, p. 4].

3.5 Annotations in Java

Annotations in Java[49] are special interfaces that are a form of metadata on classes. The purpose of annotations is to give information to the compiler and for compile-time

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyAnnotation {
    String value();
    int number();
    Class<?> clazz();
}

//using the annotation
@MyAnnotation(value = "some string",
              number = 1,
              clazz = Object.class)
public void someMethod(){
    //...
}
```

Figure 3.4: Example of defining and using a custom annotation

processing. Which elements of a Java program can be annotated is restricted by the `ElementType`[17] enum. They include constructors, fields, methods, local variables, types, and packages.

The Java standard API includes some predefined annotations, for example `@Override`, indicating that the method should override a method from a parent class and give a compile-time error if it does not override a method. There is also `@Deprecated`, indicating that a method or class is no longer in use and thus should not be used. Uses of deprecated methods will generate warning messages from the compiler.

An example of a custom annotation - `MyAnnotation` can be seen in Figure 3.4. This annotation is for annotating a method and includes three arguments: value, number, and clazz. The retention policy defined in the `@Retention` annotation means that it is available for querying through Java reflection at runtime.

In order to process the annotations an annotation processor[19] is used. Annotations are processed as part of the compilation. Use cases for annotation processors include code generation, documentation, validation, and configuration.

3.6 Choice of language

Kotlin[34], which is being developed by JetBrains, is another language that targets the Java Virtual Machine for compiling. When we started our implementation, the obvious

```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public int getAge() {return age;}

    public String getName() {return name;}

    public void setAge(int age) {this.age = age;}

    public void setName(String name) {this.name = name;}
}
```

```kotlin
class Person (private var age : Int, private var name : String) {
    fun getAge() : Int = age

    fun getName() : String = name

    fun setAge(age: Int) {this.age = age}

    fun setName(name: String) {this.name = name}
}
```

Figure 3.5: Two equivalent classes in Java(top) and Kotlin(bottom)

choice would be to write it in Java. However, Kotlin[34] also compiles down to bytecode similarly to Java. Because of this, you can write Java and Kotlin code side by side. We wanted to experiment with writing Kotlin code and Java code together. Therefore, the main part of our framework, code regarding the processing of annotations and generation of files, is written primarily in Kotlin. In many cases, Kotlin benefits from more concise and shorter code, as shown in the example in Figure 3.5.

3.7 Summary

What we ended up doing was building our framework as a Java annotation processor. When you compile Java source code, there is a step before compilation itself that processes all annotations present in the code, and then generates JUnit[37] style test classes that are automatically run during Maven's test phase. The framework uses JavaParser[32] for reading from and writing to AST representations of Java code.

The project is structured as a multi-module Maven project. The modules are; common, generator, and processors. The processors module is the main module of the project, containing an annotation processor and different utility classes relating to the processors. The generator module is the module for the data generators, containing models for creating custom generators and some default generators. The common module is meant for common methods and models used between the other modules.

For our prototype we require that the user has installed Maven and has some knowledge of how to use it. We have verified that it works with version 3.6.3 of Maven, but other versions might also work. The project is built around Java 17, and the user needs to have Java Development Kit (JDK) 17 installed and set as the current version of Java. All other dependencies should be downloaded through Maven when the project is installed.

Chapter 4

Jaxioms framework

In this thesis, we have been working on answering the research questions posed in Chapter 1. The result of our work is a framework that we have called Jaxioms. It is a prototype for integrating axioms into Java development by creating axioms as annotated static methods. The axioms are specified on classes or interfaces and can be inherited down to child classes.

The framework revolves around Java annotations. We have defined four custom annotations. The annotations we have defined are:

- `@Axiom`: the main annotation, specifies properties of the class that they are defined in. The `@Axiom` annotations are attached to static methods that represent the properties.
- `@AxiomForExistingClass`: an annotation that has the same functionality as `@Axiom`, but the properties are specified for the class whose name is given as an argument to the annotation.
- `@InheritAxioms`: an annotation specified on classes. This annotation indicates that the class should inherit all axioms from its parent classes and interfaces it implements.
- `@DefinedGenerator`: an annotation specified on classes that is a generator for another class. Will fail if it does not extend the `Generator<T>` abstract class from our framework.

One of the first plans was to use the annotation to retrieve the name of the class that should be tested, and then use the Java reflection API[21] to get the class and traverse the hierarchy looking for all axioms that should be applied. The problem with doing it this way is that since the annotation processor is run during the compilation step, the Java reflection API cannot access the classes since they have not been compiled yet.

To overcome this issue we opted for using JavaParser[32] to parse the source files to be able to read interesting properties such as which class(es) it inherits from. Parsing the source files also makes it so we can pass around the methods and classes as source code, and then write it to files to be compiled. This is utilized when we store axioms for use in subsequent runs of the framework. Axioms that have previously been collected are stored in a JSON file so that we can get axioms for the parent classes of classes that should inherit specifications. The parsed methods are also used when we write all relevant axioms to test classes.

4.1 Common Test Templates

As argued by Bagge et al. in the paper "Testing with Axioms in C++ 2011"[4], having a library that supplies common templates such as the algebraic structures, see Figure 2.1, is beneficial, as you get "free tests". Therefore, we supply algebraic structures with our framework. The structures we support currently are Semigroup, Monoid, and Group. We define them similarly to how we define a signature and a concept in Magnolia, but we adapt it to use the Java syntax by defining interfaces. Using our own `@Axiom` annotation, we indicate the axioms. The definition of the interface for Group can be seen in Figure 4.1. We can then use the interface similarly to how we did it in Figure 2.7 by using Java syntax to inherit the interface.

We also provide axioms for some of the properties defined in the Java language API[20], see Section 2.3.1. The axioms from the Java specification we support are for the `equals` and `hashCode` methods from `java.lang.Object` and for the `compareTo` method from `java.lang.Comparable<T>`. The axioms in their entirety can be seen in Figures B.1 and B.2 in the Appendix.

4.2 A Position Example

To illustrate how we can use our framework, we provide an example. We have a class `Position` representing a position or vector in two-dimensional space; see Figure A.1. We

```java
public interface Group<T> {

    T binaryOperation(T a);

    T inverse();

    T identity();

    @Axiom
    static <T extends Group<T>> void associativeBinaryOperation(T a, T
    ↪ b, T c) {
        T ab = a.binaryOperation(b);
        T bc = b.binaryOperation(c);
        assertEquals(ab.binaryOperation(c), bc.binaryOperation(a));
    }

    @Axiom
    static <T extends Group<T>> void neutralAxiom(T a) {
        assertEquals(a, a.binaryOperation(a.identity()));
    }

    @Axiom
    static <T extends Group<T>> void inverseAxiom(T a) {
        assertEquals(a.identity(), a.binaryOperation(a.inverse()));
    }
}
```

Figure 4.1: Interface representing the algebraic structure Group (from Figure 2.1)

```java
@ParameterizedTest()
@DisplayName(value = "inverseAxiom < {0} >")
@MethodSource("factoryinverseAxiom") //connecting factory to test
void inverseAxiom(org.example.Position a) {
    assertEquals(a.identity(), a.binaryOperation(a.inverse()));
}
```

Figure 4.2: Parameterised test generated for the inverse axiom in Group, see Figure 4.1

```java
public static Stream<Arguments> factoryinverseAxiom() {
    Generator<Position> clazzGenerator = new PositionGenerator();
    List<Arguments> clazzStream = new ArrayList();
    for (int i = 0; i < 100; i++) {
        clazzStream.add(Arguments.of(Named.of("Argument 1:",
           ↪ clazzGenerator.generate())));
    }
    return clazzStream.stream();
}
```

Figure 4.3: Factory for supplying generated positions to the parameterised test

want to add some binary operation to it, and we want it to have the properties defined in the algebraic structure group, see Figure 2.1. We, therefore, specify that Position should implement the interface `Group<T>` defined in the Jaxioms framework, see Figure 4.1. Comparing two positions is also something we want to be able to do for this class, so we also implement the `java.lang.Comparable` interface, which gives us the `compareTo` method. Adding the `@InheritAxioms` annotation specifies that all axioms from parent classes and interfaces should be inherited down to `Position`.

To be able to generate the axioms, we also define a generator for our class. To do that, we extend the abstract class `Generator<T>`, see Figure A.1. The generator class is a shell wrapping around the `Random`[48] class. For our implementation of the generate method, we return a new position with two random integers.

Now, we can run Jaxioms by compiling using Maven. What happens now is a test class is generated for our `Position` class. In the test class, we collected all of the axioms that `Position` inherit. The axioms have been converted into JUnit 5 parameterized tests. An example of an axiom converted into a parameterized test can be seen in Figure 4.2. Factory methods supplying instances of `Position` using our `PositionGenerator` have also been created in the same test class, which is demonstrated in Figure 4.3.

As we can see in Figure 4.3, 100 test cases are performed for each of the axioms. The result we get from running tests on all axioms can be seen in Figure 4.4; 1100 test cases

```
[INFO] Running annotations.no.uib.ii.jaxioms.PositionGeneratedTest
[INFO] Tests run: 1100, Failures: 0, Errors: 0, Skipped: 0, Time
   ↪ elapsed: 0.264 s -- in
   ↪ annotations.no.uib.ii.jaxioms.PositionGeneratedTest
```

Figure 4.4: Maven output from compiling the Position class.

```java
@DefinedGenerator
public class PositionGenerator extends Generator<Position> {
    private final Position3DGenerator position3DGenerator = new
        ↪ Position3DGenerator();
    @Override
    public Position generate() {
        if (random.nextBoolean()) return
            ↪ position3DGenerator.generate();
        return new Position(random.nextInt(), random.nextInt());
    }
}
```

Figure 4.5: Modified PositionGenerator, returning instances generated for the subclass about half the time.

passed the tests; in other words, we did not find anything wrong with the eleven axioms that Position has.

Extending our `Position` class is also possible. We might want to have a position that has three dimensions instead of two. We can then specify a class `Position3D`, see Listing A.2, which extends `Position`. We also define a generator for this class and, in the same way, specify that it should inherit axioms from its parents.

Now that we have a subclass for Position, we can modify our PositionGenerator allowing the generator also to return instances from the subclass generator, see Figure 4.5. Running the framework with both our `Position` class and its subclass `Position` gives us the two generated test classes that are seen in Listings C.1 and C.2 in the Appendix.

4.3 Using the framework

To use the framework, we first need to install it. Installing it by cloning and compiling is the most straightforward way. The project source code is located on GitHub[44], and can be downloaded there. The project is also available as a compressed archive (zip file) attached to this thesis. After downloading navigate to the correct folder, and run the command `mvn clean install`.

We also provide a Maven artifact hosted on GitHub[44], the repository can also be downloaded this way but requires some setup. Instructions on how to do this are in the `README` file in the GitHub repo for the project.

4.3.1 Configuring the project

After installing the Maven artifact, we can add it to a project. To do this, we add it to the Maven build script. We have only tested on other maven projects, so we can not guarantee that it will work for other build systems. In addition to the dependency on Jaxioms, we require adding two different JUnit dependencies. Lastly, we need to add some configuration to the `build` tag of the maven build script. In the `build` tag, we add a `testSourceDirectory` which points to the folder we generate our tests in. This, in addition to adding a plugin called `maven-surefire`[43], assures that the tests we generate will be run automatically during the test phase of the Maven build.

4.3.2 Tutorial

Now that the framework has been compiled and configured, we can add some axioms for testing. We define axioms by giving a static void method to a class. The method should employ some JUnit style assertion. We provide some static methods for `assertEquals` and `assertTrue` that can be used instead of JUnits methods, which we convert to JUnit during processing. After we have created an asserting method, we then annotate it using the `@Axiom` annotation. The tests will be generated and run when we run the framework by executing the Maven command `mvn clean install`. Following is a tutorial showcasing how to use our framework.

In the example in Figure 4.6 we have a property for the class Person which states that two instances of the class with the same name and age should be equal. The property has one parameter of type `Person` as input and we check that a new `Person` instantiated with the same name and age should be equal according to `Person.equals`. When we try to run this example we will get something like in Figure 4.7 as a result.

```java
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        if (name.length() > 8)this.name = name.substring(0,7);
        else this.name = name;
        if (age < 0) this.age = 0;
        else if (age > 150) this.age = 150;
        else this.age = age;
    }

    @Axiom
    public static void equalNameAndAgeShouldBeEqual(Person p) {
        Person q = new Person(p.getName(), p.getAge());
        assertEquals(p,q);
    }
    //getters
}
```

Figure 4.6: An example Person class

Figure 4.7: Output from running person example with Jaxioms framework

```
[ERROR]   PersonGeneratedTest.equalNamesAndAgeShouldBeEqual:25
    ↪ expected: <org.example.Person@7e5afaa6> but was:
    ↪ <org.example.Person@63a12c68>
...
```

When we look at the output we can see that this error is caused by the equals method, inherited from Object, it is checking for equality of reference and not for value. The obvious solution to this is to override the equals method. Overriding the equals method with something like in 4.8 gives us the output in 4.9, meaning that 100 tests were run, and all of them passed.

```java
@Override
public boolean equals(Object o) {
    Person person = (Person) o;
    return name.equals(person.name) && age == person.age;
}
```

Figure 4.8: An equals method for the Person class

```
[INFO] Results:
[INFO]
[INFO] Tests run: 100, Failures: 0, Errors: 0, Skipped: 0
```

Figure 4.9: Output from running person example with Jaxioms framework

In the output folder under `target/generated-sources` we see that the framework also generated a generator. These generated generators are quite experimental, so defining generators ourselves is the best course of action. This way, we know exactly what is generated for the classes we want to test. The way we define generators is to have a class that extends the abstract class `Generator<T>` from the Jaxioms generator package. We also need to add the annotation `@DefinedGenerator` to ensure the framework can discover and use the generator we created. An example of a generator for `Person` can be seen in Figure 4.10.

```java
@DefinedGenerator
public class PersonGenerator extends Generator<Person> {
    @Override
    public Person generate() {
        //randomly generated person
    }
}
```

Figure 4.10: A generator for the Person class

Now we have run some tests to check that the equals method holds the property we specified. However, as discussed in Section 2.3.1 the equals method has a lot of other properties that should always hold. Jaxioms provide these as axioms together with the framework. To test them on our equals method, we must specify that `Person` should inherit axioms from its parents. In this case the only parent present is `Object`, which is the parent of all classes in Java.

The way we specify that `Person` will inherit axioms from its parents is again with another annotation. We add the annotation `@InheritAxioms`, which can be seen in Figure 4.11. This annotation tells the framework that all axioms that it has discovered for parents of Person should be rewritten as axioms for Person as well.

Optimally we might want to inherit specifications by default. Meaning that once the Jaxioms framework is added to the project all axioms should be inherited. However, since

```java
@InheritAxioms
public class Person {
    // ...
}
```

Figure 4.11: Specifying that Person should inherit axioms from parent(s)

```
[ERROR]   PersonGeneratedTest.equalsNullIsFalse:94 - NullPointer
    ↪ Cannot read field "name" because "person" is null
...
```

Figure 4.12: Output of compiling Person with the @InheritAxioms annotation

it is implemented as an annotation processor we need an entry point where we trigger the axiom generation. One such entry point could be with the `@Axiom` annotation, and if that is present it should also inherit axioms from parents. While this would be doable it requires that a custom class has an axiom of itself in order to inherit axioms. Another issue is that we do not know which classes actually exist. In the annotation processing the classes from the project that is being worked on are not yet compiled, and thus a tool such as `java.lang.reflect`[21] are not able to find the classes.

We now see from the output in Figure 4.12 that the equals method we created earlier does not uphold the properties from `Object.equals`. Specifically the equalsNullIsFalse property fails. We can see from the equals we created in Figure 4.8 it does not do any null checks on the person parameter. Therefore, when the axiom tries to check that `p.equals(null)` it will get a `NullPointerException` because the equals method we defined tries to get `null.name`.

This is one of the problems with developers implementing equals functions. They might not always think about every property from the Java specification that their overriding methods should uphold. When they override equals, they should still ensure that the equals method conforms to Java's specification. Another thing we need to remember is that we also need to give a `hashCode` method. According to the Java API:

> "It is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes."[46]

We can easily imagine why, as also commented by Haveraaen in the paper "Axiom Based Testing for Fun and Pedagogy"[29]. If we add a person into a `HashMap<Person, T>`,

Figure 4.13: equals and hashCode methods generated by IntelliJ IDE

```java
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    Person person = (Person) o;
    return age == person.age && Objects.equals(name, person.name);
}

@Override
public int hashCode() {
    int result = Objects.hashCode(name);
    result = 31 * result + age;
    return result;
}
```

we might want to retrieve the person by another instance of the same person. This might be due to, for example, loading data into a class, which will create a new instance but still refer to the same person. If we do not override `hashCode`, we will not take into account the data that is evaluated for equality in the equals method, and thus the data might be "lost".

A good practice can be to let the IDE generate `equals` and `hashCode` for you. When we do that using IntelliJ's generator, we get the methods in Figure 4.13, which satisfies all axioms we provided.

In the paper "Axiom Based Testing for Fun and Pedagogy"[29], Haveraaen also discussed two different options for comparing objects with `equals`. Option one is where the objects are only equal if they are of the same class. Option two allows more flexibility because sub-classes can be equal to a super-class. For option two, it is, according to Haveraaen, necessary for both equals and hashCode to be declared `final` for the symmetry and transitivity properties to hold. This is because to keep the symmetry property the subclass needs to use the superclass equals method if the object its comparing against is of superclass. Which in turn breaks transitivity because two unequal subclasses can be equal in the superclass equals method. Having final on the equals method make it so that comparison of subclasses also use the superclass equals.

To make sure that `equals` and `hashCode` are consistent we also add the axiom `equalsCongruenceHashCode`, which checks that if two objects are equal, their hash code will also be equal, see Figure 4.14. A problem with this axiom, however, is that there

```java
@AxiomForExistingClass(className = "java.lang.Object")
public static void equalsCongruenceHashCode(Object o, Object p) {
    if (o.equals(p)) {
        assertEquals(o.hashCode(), p.hashCode());
    }
}
```

Figure 4.14: Axiom for `Object` "If two objects are equal according to the equals method, then calling the hashCode method on each of the two objects must produce the same integer result."[46]

is no guarantee that two generated objects will be equal. And if the generator is just random, they will rarely be equal. Thus, for the current prototype of Jaxioms this axiom is almost always true. We leave it up to future work to refine the generators so that it can take into account conditional axioms which will both generate values where the condition holds, and values where it does not.

Here, we have shown a simple use for our framework, Jaxioms. The test class generated for `Person` can be seen in Listing C.3. The final use case is specifying axioms for existing classes. This is done by using the annotation `@AxiomForExistingClass(className="somename")`, as can be seen for defining an axiom for Object in Figure 4.14. Being a prototype, some kinks still make the process of using the framework a bit difficult, but for the scope of this thesis, it is sufficient.

Chapter 5

Implementation

The following chapter details key parts of how we implemented Jaxioms. The implementation we created consists of several steps, it is invoked whenever the framework is installed and the user starts compiling a project that has one of the supported annotations present. The process is illustrated in Figure 5.1 and can be described as follows.

- The framework is invoked by the user when they have one or more of the supported annotations in a class and compiles with Maven.
- First, Jaxioms collect all axioms available in the program. These axioms can either be user-defined, come from a library, or be included in the Jaxioms framework itself.
- Afterwards, all new axioms and user-defined generators are processed. Meaning that the allowed annotations are processed, and axioms and generators resulting from this are collected.
- Now, in order to test axioms, we need generators for each of the types. Some generators might have been defined by the user, and some are generated, and a reference to them is stored in a map.
- Then, after we have a collection of all axioms and a reference to available generators, we need to generate test classes testing the axioms. We generate a test class for each class that has any defined axioms.

The program revolves around an annotation processor that processes four custom annotations, as described in Chapter 4. The framework is invoked whenever a user compiles a project which depends on the processors artifact. The skeleton of the main part of the program is illustrated in Figure 5.2.

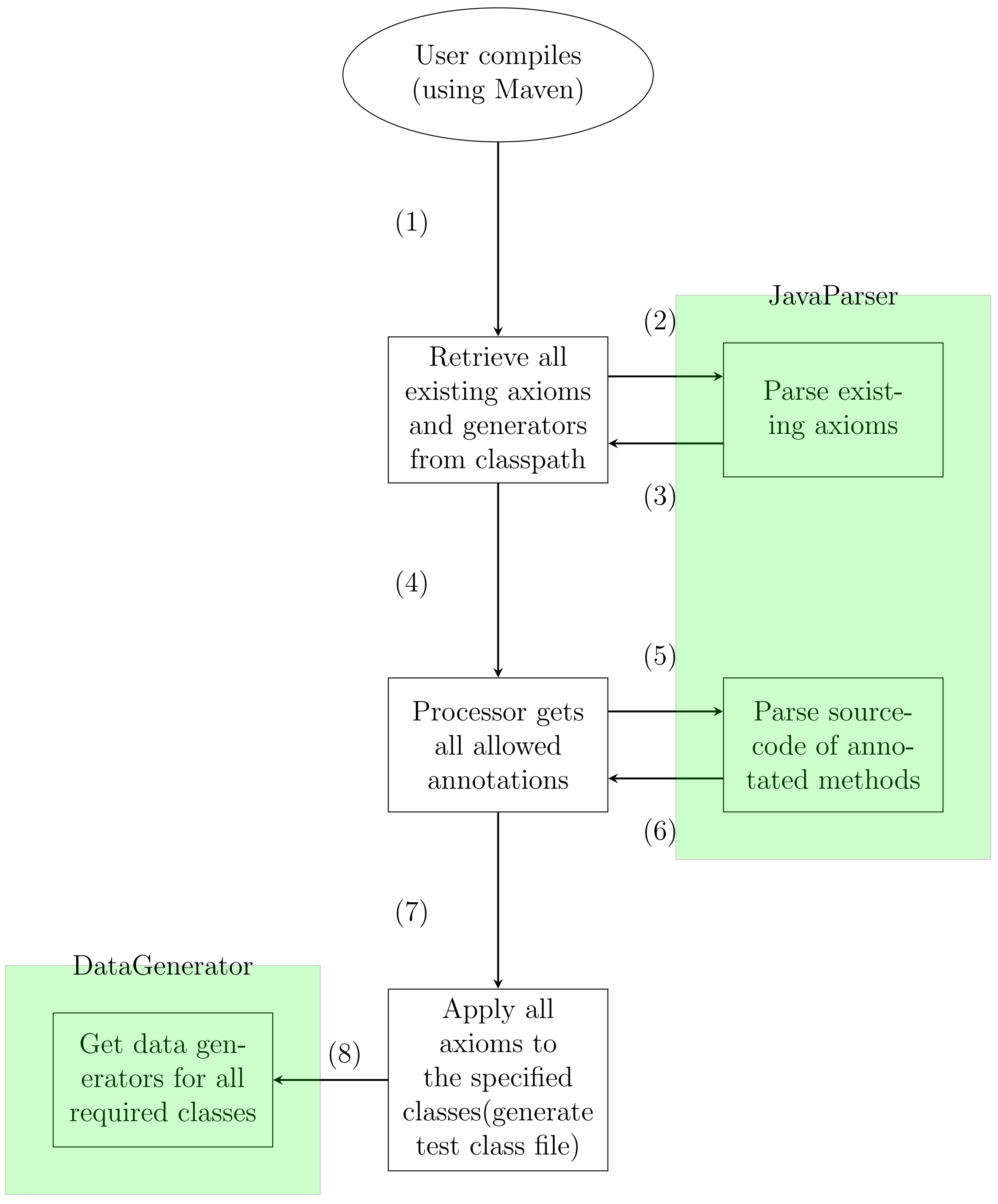

User compiles (using Maven)
(1)
JavaParser
(2)
Retrieve all existing axioms and generators from classpath
Parse existing axioms
(3)
(4)
(5)
Processor gets all allowed annotations
Parse source-code of annotated methods
(6)
(7)
DataGenerator
Get data generators for all required classes
(8)
Apply all axioms to the specified classes(generate test class file)

Figure 5.1: Sketch of the flow of the program

5.1 Collecting Axioms

When the framework has been invoked we start by collecting all available axioms. This include axioms defined by the developer in the project they are working on now, as well as axioms included in libraries, or the Jaxioms framework itself. All axioms are then stored in a map, where the keys are the qualified names of the classes that have axioms defined.

The existing axioms are retrieved by reading files that we have created on previous runs. We have created an index file containing one string for each class with previously defined axioms. The string is a filename that points to a JavaScript Object Notation[31], JSON, file for each class that has axioms defined. Then for each of the file names in the index file, a JSON file is attempted loaded. The resulting JSON is parsed as a list of axiom declarations. At the end, a map of all axioms that were found is returned to the main process. We chose to have an index file containing all the axioms as we needed to have them available at the start of the process.

5.2 Processing Annotations

Now that we have collected the previously defined axioms we can process the annotations that triggered the framework. In Figure 5.2 we illustrate the main part of the program. It consists of a class that is of the type`Processor`[19], which is an interface for annotation processors defined in Java's annotation processing package[15]. In our case, we extend an abstract subclass of `Processor`, which means we do not have to implement all the functions defined in `Processor`.

Our class `AxiomProcessor`, see Figure 5.2, has two annotations also from the annotation processing package[15]. `SupportedAnnotationTypes` defines which annotations we are accepting to this processor. The arguments to `SupportedAnnotationTypes` can be custom annotations like the ones we created, but they can also be from the Java standard library or other libraries. `SupportedSourceVersion` indicates what version of Java we support in this framework. We decided on Java 17 as it was the most recent long-time supported version at the time we started implementation[22]. However, supporting more than one version of Java would be beneficial, and should be considered for future work.

```
@SupportedAnnotationTypes("no.uib.ii.annotations.Axiom",
    "no.uib.ii.annotations.InheritAxioms",
    "no.uib.ii.annotations.AxiomForExistingClass",
    "no.uib.ii.annotations.DefinedGenerator")
@SupportedSourceVersion(SourceVersion.RELEASE_17)
class AxiomProcessor : AbstractProcessor() {
    //fields etc.
    override fun process(annotations: MutableSet<out TypeElement>?,
       ↪ roundEnv: RoundEnvironment?): Boolean {
        val predefinedAxioms = loadAxiomsFromFiles()
        annotations?.forEach(
            fun(annotation: TypeElement) {
                val elementsAnnotatedWith =
                   ↪ roundEnv?.getElementsAnnotatedWith(annotation)
                try{
                    when (annotation.toString()) {
                        "no.uib.ii.annotations.AxiomForExistingClass"
                           ↪ -> {
                            UserDefinedProcessing
                            .processAxiomForExistingClass(
                                elementsAnnotatedWith,
                                ...)
                        }
                        "no.uib.ii.annotations.Axiom" -> {
                            UserDefinedProcessing
                            .processAxiom(
                                elementsAnnotatedWith,
                                ...)
                        }
                        "no.uib.ii.annotations.InheritAxioms" -> {
                            UserDefinedProcessing
                            .applyAxiomsFromParent(
                                elementsAnnotatedWith,
                                ...)
                        }
                        "no.uib.ii.annotations.DefinedGenerator" -> {
                            generatorProcessing
                            .processGenerator(
                                elementsAnnotatedWith,
                                ...)
                        }
                    }
                }catch (e: Exception){
                    processingEnv.messager.printMessage(
                        Diagnostic.Kind.WARNING,
                        "Error processing annotation")
                }
            }
        )
        if (roundEnv?.processingOver()!!) {
            testClassGenerator.generateTestClassesForAxioms(...)
            fileUtils.writeGeneratorList(availableGenerators)
            fileUtils.writeAxiomsToPropertyFile(axiomDeclarations)
        }
        return false
    }
}
```

Figure 5.2: Main component of the annotation processor, ties the program together

```kotlin
val typeElement = element.enclosingElement as TypeElement //casting
    ↪ class holding the axiom method to a type
val axiomMethod = processAxiomMethod(element, typeElement, filer,
    ↪ typeUtils) //processing the axiom method (element) in the class
    ↪ (typeElement)

val existingAxiomsForClass = axiomDeclarations.getOrDefault(
    typeElement.qualifiedName.toString(),
    ArrayList()
); // all existing axioms for the class
existingAxiomsForClass.add(axiomMethod)
axiomDeclarations[typeElement.qualifiedName.toString()] =
    ↪ existingAxiomsForClass
```

Figure 5.3: Processing the `@Axiom` annotation

For the processing we override the `process` function of the interface. The annotations are given to us by the Java compiler as a `MutableSet` of type `TypeElement`. The `RoundEnvironment` of the processor allows us to access information about the current round of processing, such as all elements annotated by a given annotation. Using this we can for each of the annotations find all elements that are annotated by a given annotation, and then dispatch the elements along with some other variables we might need. At the end of processing we return a boolean value of false. This value says whether or not we "claim" this annotation. Meaning that since we return false other annotation processors are allowed to process these annotations.

5.2.1 Processing @Axiom

The Axiom annotation is the standard way of defining an Axiom. For a given class, the user writes properties, which are annotated by `@Axiom`. The properties should take in some parameters that will be used to generate input data. The property should also do some JUnit style assert. The key parts of processing `@Axiom` can be seen in Figure 5.3. For each of the new elements that is annotated with this annotation, we find the method body by parsing the source file of the class to which the method belongs. We then add the axiom to our map, where the key is the name of the class where the method was defined.

```
val typeElement = element as TypeElement
var axioms = axiomDeclarations.getOrDefault(
    typeElement.qualifiedName.toString(),
    ArrayList()
)
typeElement.interfaces.forEach {
    val e = typeUtils?.asElement(it) as TypeElement
    axiomDeclarations[e.qualifiedName?.toString()]?.forEach { axiom ->
        axioms.add(axiom.copy())
    }
}

val e = typeUtils?.asElement(typeElement.superclass) as TypeElement
axiomDeclarations[e.qualifiedName.toString()]?.forEach { axiom ->
    axioms.add(axiom.copy())
}

axioms = convertGenericAxioms(axioms, typeElement, typeUtils)
axioms = convertParentAxioms(axioms.toMutableList(), typeElement,
  ↪ filer).toMutableList()
axiomDeclarations[typeElement.qualifiedName.toString()] = axioms
```

Figure 5.4: Collecting axioms for parent(s) and converting them for child class

5.2.2 Processing @InheritAxiom

For the `@InheritAxiom` annotation, differ a bit from `@Axiom`. In Figure 5.4 the main part of processing this annotation is shown. The element that has the annotation is a class so we cast that to a `TypeElement`. We then get all the axioms gathered for this class, as well as all axioms for the interfaces and superclass. Compilation might be done twice for everything to work. This is due to the axioms for the parent of this class might not have been stored yet.

After having collected all axioms for the class, we need to convert the axioms that are for the parent class and those for generic parameters. In brief, this is done by replacing all instances of the type that should be converted in the parsed method to the child class. This process should be refined as minimal checks for whether the classes are in the correct scope, and thus which imports are present, are done. We leave this up to future work.

5.2.3 Processing @AxiomForExistingClass

The third of our annotations, `@AxiomForExistingClass`, is processed similarly to `@Axiom` and is shown in Figure 5.5. This annotation requires a string parameter `className`. The axiom

```
val typeElement = element.enclosingElement as TypeElement
val annotation =
    ↪ element.getAnnotation(AxiomForExistingClass::class.java)
val axiomMethod = processAxiomMethod(element, typeElement, filer,
    ↪ typeUtils)
axiomMethod.setGeneric(true)
axiomMethod.setQualifiedClassName(QualifiedClassName(annotation.className))
val existingAxiomsForClass = axiomDeclarations.getOrDefault(
    annotation.className,
    ArrayList()
);
existingAxiomsForClass.add(axiomMethod)
axiomDeclarations[annotation.className] = existingAxiomsForClass
```

Figure 5.5: Processing the `@AxiomForExistingClass` annotation

is stored for the class name in this string, instead of getting it from the class that the axiom method is defined in, as for `@Axiom`. For the time being, defining axioms for existing classes only works for classes that are supposed to be a superclass or interface. It is not supported yet, future work needs to look at how to generate data for classes that we do not have the source code for. One option is to always require that the user supplies a generator by using `@DefinedGenerator`.

5.2.4 Processing @DefinedGenerator

The final annotation we have is `DefinedGenerator`. This is used as an indicator that there is a generator present and thus we needn't try to generate one. We also check that the class annotated by this method is a subclass of the abstract class `Generator<T>` so that we know it implements a method `T generate()`.

5.3 Generating Generators

Inspired by the work done for creating a language based around property-based testing in [27], we did some experimenting on generating generators. This is done when the program has found all axioms and generators and there are no generators for some of the classes that have axioms. The framework tries to generate the generator based on a public constructor in the class. If there are no constructors the generator will be a meaningless shell as no meaningful data can be added to the objects. There is no logic

```
var extendsGenerator = false;
val classDeclaration =
   ↪ FileUtils.getClassOrInterfaceForTypeElement(element as
   ↪ TypeElement, filer)
for (type in classDeclaration.extendedTypes) {
    if (type.name.toString() == "Generator") {
        val typeArguments = type.typeArguments.orElseThrow {
            UnexpectedError("Type arguments for Generator not found")
        }
        if (typeArguments.size != 1) {
            throw UnexpectedError("Generator must have exactly one
               ↪ type argument")
        }
        val typeArgument = typeArguments[0]
        //val id = cu.imports.find { id ->
           ↪ id.nameAsString.endsWith(typeArgument.asString()) }
        dataGenerator.addGenerator(typeArgument.asString(),
           ↪ classDeclaration.fullyQualifiedName.orElseThrow {
            UnexpectedError(
                "Fully qualified name not found"
            )
        })
        extendsGenerator = true;
    }
}
if (!extendsGenerator) {
    throw UnexpectedError("Class ${classDeclaration.nameAsString} must
       ↪ extend Generator<T>")
}
```

Figure 5.6: Processing the `@DefinedGenerator` annotation

to decide which constructor to use. In case there are multiple constructors, it will return a generator based on the first constructor that is in the list for the `CompilationUnit` that JavaParser has parsed for this class. If no generators are defined for the arguments of the constructor chosen, the generator will also fail.

The work done on generating generators is very experimental. It is recommended that the users define generators themselves by extending the `Generator<T>` class from the generator package and adding the `@DefinedGenerator` annotation to let the framework discover the generator.

5.4 Generating Files

5.4.1 Generating Test Classes

In the end, when we have hopefully gotten generators for all relevant classes, we will generate the test class. For each of the classes that have a generator defined or generated, we create a test class located in `target/generated-sources/`. The code for creating the classes can be seen in Figure 5.7. First, we add a list of imports for all the dependencies. At the time the imports are more or less a static list of the dependencies we need. However, it would be beneficial to get the imports more dynamically in the future. The class contains the axioms as well as a factory method which acts as a bridge between the axiom and the generator.

What we end up with is a JavaParser CompilationUnit which represents the test class. It can be visualized like in Figure 5.8. The `CompilationUnit` contains a class definition which holds all our axiom methods, represented in the figure by “someAxiomMethod”. It also has all the factory methods, represented by “factorysomeAxiomMethod”. The reason that the package name has “annotations” in front of it is that files generated using the `Filer` interface of Java’s annotation processor package[15] are stored in this folder. When we have achieved a CompilationUnit it is easy to get the source code of the Java file it would represent by calling the `toString` method.

5.4.2 Writing to Support Files

When all is done, we store the axioms that we collected as JSON objects in files, as well as an index file pointing to each of the JSON files. We also store a generator index that tells us what generators are available and what the package name and class name are.

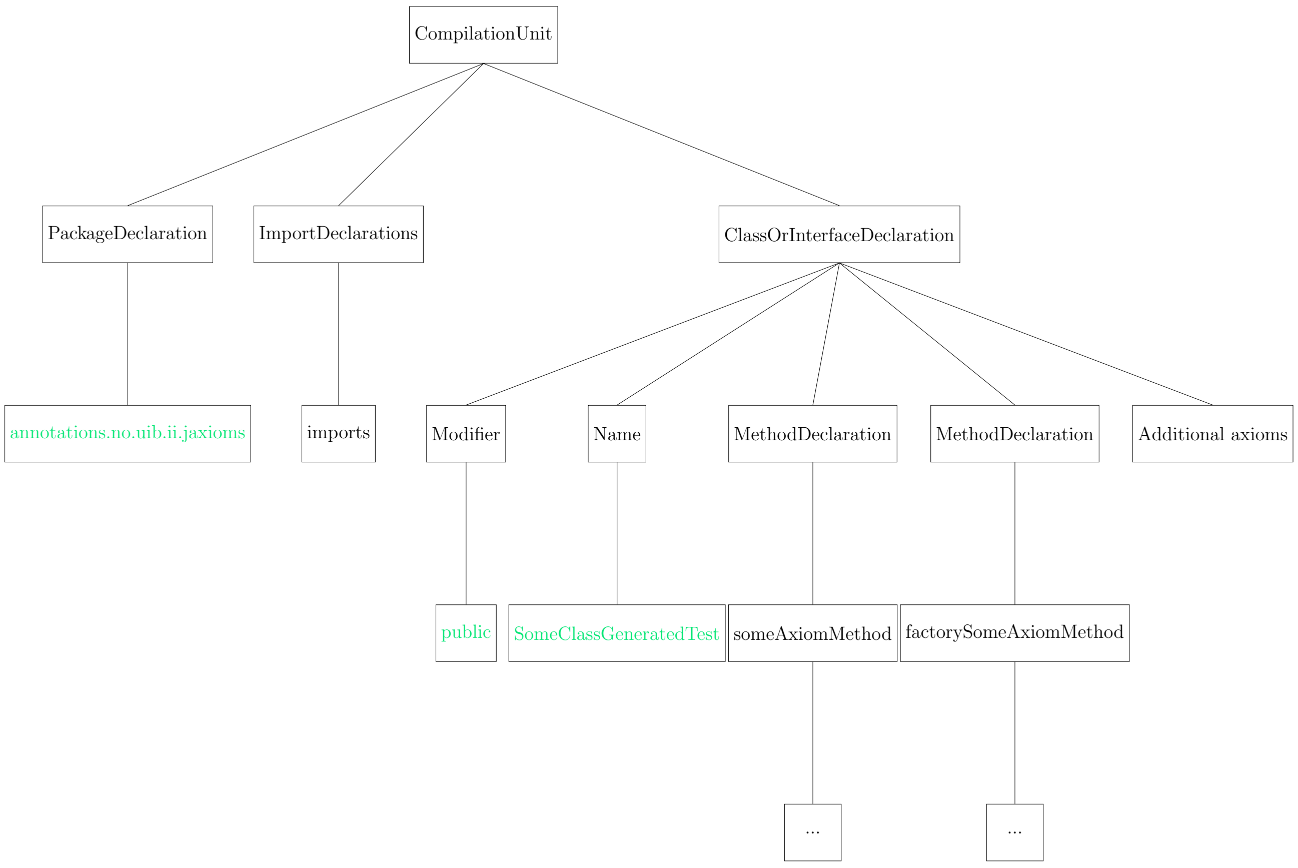
```kotlin
var cu: CompilationUnit = CompilationUnit("annotations.no.uib.ii.jaxioms");

val methods: List<MethodDeclaration> =
    axiomDeclarations.map { axiomDeclaration -> axiomDeclaration.getMethod() }

imports.forEach(fun(import: ImportDeclaration) {
    cu.addImport(import)
})
var classDeclaration = cu.addClass("${className}GeneratedTest");
methods.forEach(fun(method: MethodDeclaration) {
    val args = (1..method.parameters.size).joinToString(",") { "{${(it - 1)}}" }
    classDeclaration.addMember(
        MethodDeclaration().setBody(method.body.orElseThrow())
            //set various properties from method
            .addAnnotation("ParameterizedTest")
            .addSingleMemberAnnotation("MethodSource", "\"factory${method.name}\"")
    )
    classDeclaration.addMember(
        getStreamMethod(className, method.parameters.size, "factory${method.name}")
    )
})
filer.createSourceFile("no.uib.ii.jaxioms.${className}GeneratedTest")
    .openWriter().use { writer ->
    writer.write(cu.toString())
}
```

Figure 5.7: Generating a test class, using a JavaParser[32] CompilationUnit to keep track of the file before writing its string representation to the file

Figure 5.8: A JavaParser tree representing the test class we generate



5.5 Summary

Following is a summary of key parts of our implementation. We have the `@Axiom` and `@AxiomForExistingClass` annotations for defining axioms. These are annotations on methods that are then used to generate test classes for classes that should have these specifications. We also have two “supporting annotations”, `@InheritAxioms` and `@DefinedGenerator`, mainly for being able to discover files and process them as part of annotation processing. It is important to be able to process the files in annotation processing so that the tests can be run as part of compilation, and not require an additional (manual) step. After we have collected all axioms and figured out where to apply them, we generate files. These files include test classes containing axioms, as well as some supporting files to be used on subsequent runs of the framework.

Chapter 6

Discussion

6.1 Breaking Encapsulation

As mentioned in Section 2.2, one of the principles of object-oriented programming is encapsulation. Fields should as a rule of thumb be private. When we write axioms in the classes the fields being private is no issue. However, issues arise when we copy the axiom to the test class. Because the test class can not access private fields in other class files. In our prototype, we have made it a requirement to have fields that you access in axioms either be public or use getter methods in the axioms.

Having public fields can break encapsulation by allowing for external modification. To fix this, an extension could be made to the framework that, for instance, requires that values accessed in axioms have getter methods. Another option for fixing this, which we suggest, is instead of including the entire method in the test class file, we can call the static method from the class that owns the axiom. With our current implementation, there need to be some changes, but they should be doable.

The change is fairly easy for generating the test classes. Instead of copying the entire method, we add a call to the static method from the class where the axiom is defined. We must also ensure that an import is added. The way we do it currently is seen in Figure 6.1, and our suggested fix is in Figure 6.2.

In Section 2.5.6 we discussed the different categories of properties described by Corgozinho et al. in “How Developers Implement Property-Based Tests”[13]. For our framework, all of the categories are supported. The only thing to ensure is that the functions called

```java
@ParameterizedTest()
@MethodSource("factoryequalsIsReflexive")
public void equalsIsReflexive(org.example.Position o) {
    assertEquals(o, o); //body copied from
      ↪ ObjectAxioms.equalsIsReflexive
}
```

Figure 6.1: Example of how the current implementation looks, axiom body copied over to generated test class

```java
@ParameterizedTest()
@MethodSource("factoryequalsIsReflexive")
public void equalsIsReflexive(org.example.Position o) {
    ObjectAxioms.equalsIsReflexive(o);
}
```

Figure 6.2: Example how it would look after applying our suggestion

from axioms should, for this as well, be accessible from the external test class. This is especially important for the categories such as “Test oracle” and “Model-based testing”, if the tests rely on external classes, they must be in scope for the caller. If we implement the suggestion above this is no problem as the classes should be in scope in the place where the axiom is defined.

6.2 Storing generated classes

Another issue we need to address is where to store the classes we generate. For our current prototype, we store them in the target folder along with other generated files. However, this might be problematic if we want to keep the files at a later point. We also wanted to include a test report that states what files were used to run the tests but to do this, we would have to have a way of storing the generated files to be able to reproduce the test.

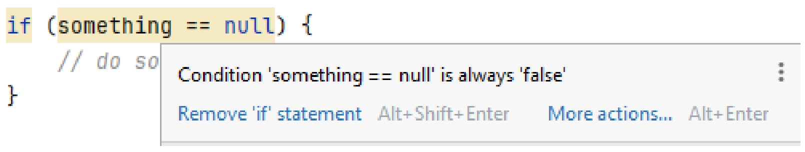
For storing files, we use the `Filer` interface from the annotation processing package[15]. This interface writes the files to the default source and class output folders, which is the target folder for Maven. If we wanted to write the files somewhere else, we would have to use another API for writing files. However, Java’s documentation for the annotation processor[19], states “The Filer interface discusses restrictions on how processors can operate on files." which means that it would probably be a bad idea to store the files in another location.

6.3 Better Data Generation

For our prototype, we provide some basic functionality for generating generators for custom classes. This work should be extended so that the data generated is more meaningful. As argued by Haveraaen in [29], random testing avoids bias towards test data as it can span the entire data space. This makes it harder for developers to create code that passes tests by accommodating the test data at hand. While random tests give a good data distribution, generating random data will rarely generate equal data. Haveraaen argues in [29] that the data generator could skew the random data towards generating data that is equal more often than not.

6.4 Static analysis

Another improvement to our framework could be to utilize static analysis to cut back on the need for testing. For example, IDEs give you a lot of static analysis through tips that improve your code. The recommendations can be like in the example in Figure 6.3. Adding static analysis to our framework could cut back on the amount of test cases done. If we by using static analysis while generating axioms can see that an axiom can never fail, there is no point in running tests on this axiom for this test run.



```
if (something == null) {
    // do so
}
```

Condition 'something == null' is always 'false'

Remove 'if' statement Alt+Shift+Enter
More actions... Alt+Enter

Figure 6.3: IntelliJ gives a hint that name can never be null and this check is redundant

6.5 Using Java reflections

For our prototype, we must explicitly specify that a class should inherit specifications from its parent(s). It might be beneficial to do this automatically, but this would require more work. We would have to load all classes in the classpath and then create a tree structure that connects all the relevant types. This way, we can traverse our tree to find subclasses of a class with some specifications. Another way would be to rely on a third-party framework for scanning the class hierarchy. Reflections[52] is one such framework. However, an initial investigation led us to believe that this framework has the same limitation as the built-in Java reflection: classes must be compiled to be found.

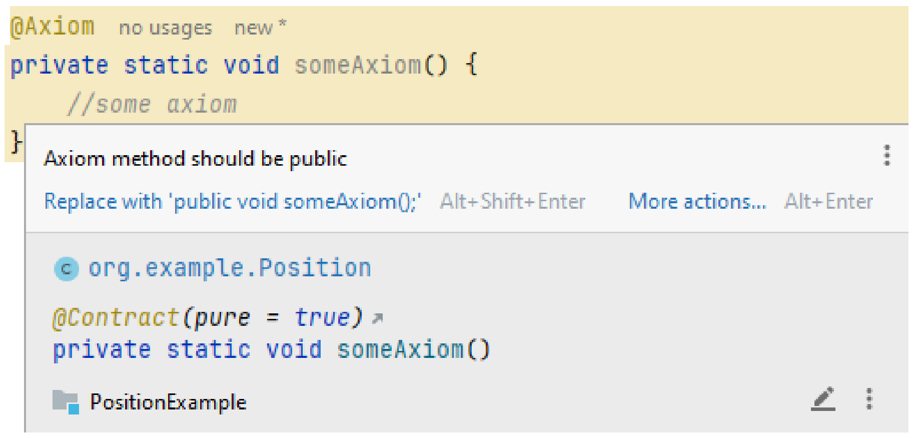
A possible approach to utilizing reflections is to run the reflection code in the generated test class. In other words, instead of generating test classes as we do now, we generate code to find subclasses of a type and call the axiom with the subclasses found.

Liskov substitution principle, as discussed in Section 2.2, is one of the principles for object-oriented programming. Classes that inherit specifications are tested for this principle, as the tests will fail if they do not follow axioms specified in the parent classes. It would also be beneficial to have the generators supply instances of the subclasses for a given class to check that subclasses conform to the properties.

Seeing as this principle requires that the subclass can be used as a substitute for the superclass, we should check that the parameters of the axioms can have different types. i.e., be subclasses alongside superclasses to check that it satisfies this principle. In order to check that classes conform to this principle we could for our axioms generate not only the class itself but also known subclasses.

This way, the axioms would still have the superclass for its parameters, but the generators can supply instances of subclasses to uncover possible bugs. For the current version of our framework, this is supported if the user creates the generator themselves, as can be seen in Figure 4.5. For the generated generator, it is not supported. If the subclass has been discovered by our framework and has a defined generator the change is to add to the generator an instance of a subclass generator. We would also decide on how we should distribute the cases. For a previous example in Figure 4.5, we added a subclass generator and said that we use that one `if (random.nextBoolean)`. To support this for subclasses that are not discovered by our framework we would have to change some key parts of our implementation. It would require that we could find all known subclasses. As discussed in the previous section, this is not possible without relying on some other mechanism like “Java reflections”[52].

The reflections project is no longer being maintained but is available with an open license. This means that if we want to add features from reflections, we could depend on the unmaintained version and hope that it will not break in future updates of Java.



```
@Axiom  no usages  new *
private static void someAxiom() {
    //some axiom
}
```

Axiom method should be public

Replace with 'public void someAxiom();' Alt+Shift+Enter   More actions... Alt+Enter

```
org.example.Position
@Contract(pure = true)
private static void someAxiom()
```

PositionExample

Figure 6.4: Example of a custom inspection in IntelliJ, replacing the private modifier with a public

Another approach would be to copy their code and include it in the Jaxioms framework. The downside to this approach would be that it requires more effort from the maintainers of Jaxioms.

6.6 IDE Integration

Even though we wanted to be independent of an IDE, having some optional integration with IDEs could prove useful. For example, if we create a more robust way of handling exceptions, perhaps there is a way to show suggestions in the code based on which exception was thrown. If we inherit axioms from parents, maybe a suggestion for adding `equals` and `hashCode` methods could come up.

IntelliJ offers some functionality for adding custom code inspection[36]; these can then be bundled together in a plugin. Figure 6.4 shows a prototype for one of the code inspection corrections that would be nice to have, if an axiom method is not public, we could supply a suggestion allowing the developer to change it directly

If we create a plugin for Jaxioms, we want to keep it optional. A thing to consider is also that optimally, to accommodate as many different workflows as possible, we should create plugins for more than one IDE. This, again, leads to a lot of work that might be unfeasible.

6.7 Compatibility

For our prototype, we found it sufficient to use one version of Java. The work does not rely on Java 17 specific features, but we have defined that our annotation processor only supports Java 17. Adding support for other versions might introduce some problems that need resolving. One thing that needs to be done to support more Java versions is to configure the JavaParser based on which version is used, examples of how to do this can be seen in JavaParser's tutorial[59]. JavaParser supports Java versions 1 through 18.

The Maven version we used was version 3.6.3. Initially, we did not predict that this would pose any issues, but testing on different computers that ran a newer version of Maven (3.9.7) revealed that there was an issue with running on this version. This was resolved once we downloaded version 3.6.3. We should therefore look at what has changed from version 3.6.3 to version 3.9.7 and how we can fix it so that we do not rely on a specific version. It should also be made available for other build systems than Maven.

6.8 New features

6.8.1 Data invariant support

Haveraaen and Kalleberg also provided an additional plugin in "JAxT and JDI - The Simplicity of JUnit Applied to Axioms and Data Invariants"[30], called the Java Data Invariant plugin (JDI). The JDI plugin adds support for verifying data invariants of Java classes. To use it the user creates a method `dataInvariant()` which is some Java statements about the fields of the class. JDI is then invoked to insert checks throughout the code. It rewrites the source code to insert calls to `dataInvariant()` at method entries and exits. The program is then compiled, and breaches of the invariant give exceptions.

If we want to add data invariant checks to our framework or make a new framework that supports this, it could be done by having an annotation `@DataInvariant`. This annotation can generate a test class that calls all public methods of the class that has the annotated method, see Figure 6.5. We can then insert data invariant checks before and after the method calls. We could have the data invariants as boolean methods, which would then be verified for truth, or we could keep them in the same style as our axioms and have JUnit assertions directly in the method.

```java
public class Person {
    public String name;
    public int age;

    @DataInvariant
    public void dataInvariant() {
        if (age < 0 || age > 150) throw new
           ↪ InvariantBreachException("Not a valid age");
    }
}
public class PersonGeneratedTests {
    @ParameterizedTest()
    @MethodSource("factorySetAgeDataInvariant")
    public void setAgeDataInvariant(Person p, int age) {
        p.dataInvariant();
        p.setAge(age);
        p.dataInvariant();
    }
}
```

Figure 6.5: Sketch of how the data invariant "extension" to Jaxioms could look

One option would be to generate a separately compiled class file that is equivalent to the class but with invariant checks. This would, however, defeat the purpose as you won't get the invariant checks in the regular class, and you would still need to have a test class file that checks that invariants hold.

Another option is not to create invariant checks in a separate file but instead do like in JDI[30] and insert the checks throughout the generated code. This requires more work than what we have done with Jaxioms because the annotation processor can not modify the source code directly. Something similar to this is done by a library for Java called Lombok[41] where they have an annotation `@Builder` which adds a builder to the class compiled from the source class annotated by this annotation. Future works should include looking into how they do this and whether techniques similar to Lombok's can be utilized for Jaxioms.

Having a data-invariant method could also prove useful for generating generators. We could analyze the data invariant and only generate values that fall within its valid range.

6.8.2 Supply manual test data

A feature that would be nice to have would be the ability to supply manual test data. This could be done by a Comma-Separated Values (CSV) file or similar. The way it could be defined is by having an argument for the `@Axiom` annotation that is a file path to a file containing data. This would be beneficial because the developers might know what values can cause issues or values that have caused issues in the past. And they can thus ensure that these values do not cause issues again. The manual data would be in addition to generated test data so that we do not lose the advantage of random testing. JUnit parameterized tests give a way of adding CSV sources to the parameterized test by adding the `@CSVSource` annotation, which we could use when generating test classes.

In addition to supplying manual test data, it would be useful to keep data from previous runs. We could for example along with the axioms store the random seed that was used, and what generator was used. This way if the tests fail we can rerun it on the same seed when the code has been changed to see if it fixed the test fails. If it succeeds and there has not been any changes to the code there is no need to re-run it on the same random seed. If only a few of the test cases fail it could also be beneficial to store the values that caused the issue for the future. Meaning that problematic data values can be tested against more often.

6.8.3 Rewriting

We discussed how we can specify algebraic structures that have axioms that can be inherited in Section 4.1. One downside to inheriting these specifications is that we lose the ability to give our classes as meaningful method names. For the example we give in Section 4.1, we have a `Position` class, which inherits a binary operation from `Group<T>`, see Figure 4.1. Since this is done by implementing an interface we have to use the same name for the method. However, we might want to rename the `binaryOperation` method to have a more meaningful name like `add`.

We recall that Magnolia, see Section 2.1.1, uses a technique called rewriting for re-defining the names of specifications you inherit. We could create a similar feature using annotation processing to allow this. We would probably have to turn off some warnings or exceptions in the IDE to avoid red squiggly lines when not implementing the methods from an interface. The rewriting could be done similarly as discussed earlier for data invariants by rewriting the class before compiling it.

Another way of solving this is to use an annotation to specify that it should implement an interface, with rewrites as arguments to the annotation, instead of using the regular `implements Interface`. An example of this can be seen in Figure 6.6. Then, for the

```java
@ImplementAndRewrite(clazz=Group.class,
                     rewrites = {"binaryOperation => add",
                                 "identity => origin",
                                 "inverse => negative"})
public class Pos {

    //fields and constructor

    public Pos add(Pos a) {
        return new Pos(x + a.x, y + a.y);
    }

    public Pos negative() {
        return new Pos(-x, -y);
    }

    public Pos origin() {
        return new Pos(0,0);
    }
}
```

Figure 6.6: Suggestion for how rewriting could look

processing of the annotation, we would have to rename the methods of the `Pos` class to the methods of `Group`. We would also have to specify that `Pos` implements `Group`. A downside to this is that we would lose the IDEs built-in "implement methods" feature. We could, however, create a similar feature ourselves as part of the IDE integration discussed in Section 6.6. A sketch of how this could be done is illustrated in Listing 6.6

6.8.4 Specifying configuration properties

Letting the developers specify properties for the framework would be a nice addition to allow for some customization. Top-level properties that should pertain to the framework as a whole could be specified in a separate file and read at the start of annotation processing. We could also add fields for properties in the annotations themselves to allow for customization on a per-axiom level. A property that would be relevant is the number of test cases to test each axiom for. Another property could be allowing the user to customize whether to store test runs and where to store generated classes.

6.8.5 Better error handling

An issue we have had while developing the framework is that exception handling is not done in a good manner. As it was created by adding new things and experimenting with what works and what does not, we did not always keep in mind that error-prone situations should be reported with proper exceptions. As an afterthought, we added some custom exceptions to try and mitigate this, but there is still some work left on this. Our suggestion is to have an exception handler as a wrapper around the program that handles all exceptions. This way, we can create custom exceptions that include important values for specific error situations and report them to the user in a good way. We could also allow the users to specify that errors from Jaxioms should not crash the program. It could instead give warnings that the axioms are not upheld. This could be done by letting the user specify in a property file, as discussed in the previous section, whether Jaxioms failing should stop compilation, or give warnings in the log. Exception handling in the annotation processor should use the interface `Messager` from Java's annotation package[15], which gives a way of reporting errors and warnings.

6.8.6 Better test reporting

The way reporting works at the moment is with the Maven surefire plugin[43]. It generates a test report for each of the test class files that is run. When the tests fail we get a stack trace for what went wrong, and when they succeed we get the number of tests that passed. It would be helpful to have some better reporting even if the tests succeed so that we can see which values were tested for the properties. Some work was done to try this, which is why our parameterized tests have `@DisplayName` annotations. We did not get this working properly though.

We could use JUnit assertEquals string messages as well to get more meaningful feedback when the tests fail. One of the things to consider is that these messages come with quite a bit of overhead. When doing an assert like `assertEquals(a,b, a.toString()+ ↪ "is not equal to "+ b.toString());` the error message will be calculated even if there is no error. This might seem like a small problem, and it is in the case of few asserts, but when running many, the time difference is noticeable. In Figure 6.7, we did an experiment of the time difference running `assertEquals(a,b, a.toString()+ "is not equal ↪ to "+ b.toString());` versus using `assertEquals(a,b);`. We ran the experiment 100 times, each time executing the assert 1 000 000 times. While these values are exaggerated,

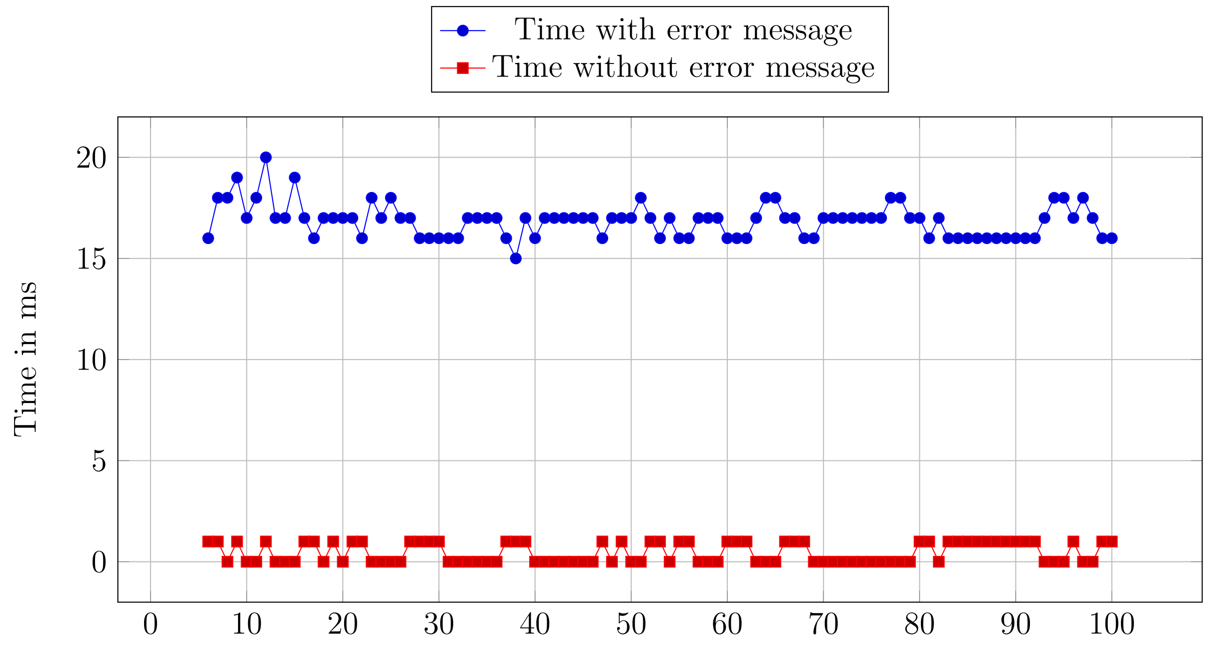


Figure 6.7: Experiment timing different assert statements

they show the point that using error messages requires significantly more time. Note that since the assertions themselves are basic in the experiment, there might be even more of a difference when doing assertions on more complicated functions. What could be done is to have a check before executing the assert, so that if they are equal, we assert without the message. If they are not equal we assert with the message string.

The Java language provides a class for giving test failure - `java.lang.AssertionError`[2], and then it is up to the developers of testing frameworks to bridge the gap by giving some custom exception that extends either AssertionError or RuntimeException. This means there is no guarantee that IDEs and build tools can display the failed test well. JUnit, therefore, created a new open-source project called the "Open Test Alliance for the JVM"[57]. This project aims to provide a standard way of throwing exceptions during testing so that, for instance, IDEs can discover the test results. This work would prove beneficial if we wanted to get rid of the dependency on JUnit. We could utilize the exceptions in this library and give customized results for our framework.

Chapter 7

Conclusion

In this thesis, we present our work on Jaxioms, a framework for testing with axioms in Java. Jaxioms allows for specifying axioms on classes in Java. We also allow for inheriting axioms from superclasses and interfaces. Jaxioms also provide some axioms for properties specified in the Java language API. We also provide some interfaces for algebraic structures.

Our approach utilizes Java's built-in mechanism for processing annotations. It also uses JavaParser to parse source code and create a representation of classes that can be written to a file. We have shown how developers can use Jaxioms to integrate Axiom-based testing into their testing workflow. Our framework provides a proof-of-concept for how Axiom-based testing can be implemented in the context of modern Java. With some further refining as described in Chapter 6, it can become a fully-fledged framework for integrating axioms in Java.

We recall the questions we asked in Chapter 1.

RQ1. How can we make it easy for developers to integrate axioms into the Java software development process?

RQ2. How can we apply what was done in JAxT to the context of modern Java?

To answer RQ1, we have looked at what workflows are most familiar to developers. We looked at which IDE is most popular. Though the validity of the surveys[23, 35] we found can be questioned, it looks like the current most popular IDE for Java is JetBrain's

IntelliJ[35, 51]. Which testing framework is most widely used was also considered. What was found in [35] is that JUnit is by far the most used. We also looked at which build system developers use, which, according to[35], is Maven. Based on these findings, we wanted a framework that is, at the very least, JUnit and Maven compatible. Relying on an IDE was something we did not want to do, and thus, our framework can be used from the command line. To make it easy for developers to integrate axioms into the Java software development process we provide a framework independent of any IDE. This way the developers can incorporate it into their current development process.

For the second question, we examined how to apply JAxT's different features in a modern context. JAxT allowed for specifying axioms for classes, as well as inheriting specifications from parents. We also provide support for specifying axioms for classes. And we provide a way of specifying axioms for existing classes. We also allow for inheriting specifications. One of the things that JAxT supported that we have not done is the different kinds of axioms. The motivation behind these was from the Java specification where the formulation some places were that something was recommended, but not necessary.

We have shown that it is possible to utilize annotation processing for Axiom-based testing. Even though we made it independent of any IDE, our prototype relies on some third-party software, which is sub-optimal. Other limitations to our framework, and suggestions for the future, were discussed in Chapter 6.

Bibliography

[1] Algebraic structures between magmas and groups.
URL: `https://commons.wikimedia.org/wiki/File:Magma_to_group4.svg`. Accessed: Wed May 8 2024.

[2] Assertionerror (java se 17 and jdk 17).
URL: `https://docs.oracle.com/en/java/javase/17/docs//api/java.base/java/lang/AssertionError.html`. Accessed: 2 Jun 2024.

[3] Apache. Welcome to apache maven.
URL: `https://maven.apache.org/`. Accessed: 2 Jun 2024.

[4] Anya Helene Bagge, Valentin David, and Magne Haveraaen. Testing with axioms in c++ 2011. *Journal of Object Technology*, 10:10:1–32, 2011. ISSN 1660-1769. doi: 10.5381/jot.2011.10.1.a10.
URL: `http://www.jot.fm/contents/issue_2011_01/article10.html`.

[5] Kent Beck. Simple smalltalk testing: With patterns.
URL: `https://live.exept.de/doc/online/english/tools/misc/testfram.htm`. Accessed: February 24, 2024.

[6] Magne Haveraaen Benjamin Chetioui, Jaakko Järvi. Revisiting language support for generic programming: When genericity is a core design goal. *The Art, Science, and Engineering of Programming*, 7, 2022.
URL: `https://doi.org/10.22152/programming-journal.org/2023/7/4`.

[7] BLDL. Axiom-based testing, .
URL: `https://bldl.ii.uib.no/testing.html`. Accessed: May 21, 2024.

[8] BLDL. Design of a mouldable programming language (dmpl), .
URL: `https://bldl.ii.uib.no/dmpl.html`. Accessed: May 28 2024.

[9] BLDL. Bergen language design laboratory (bldl), .
URL: `https://bldl.ii.uib.no/`. Accessed: May 28 2024.

[10] Rob Prouse Charlie Poole. Nunit.org.
URL: `https://nunit.org`. Accessed: Fri 31 May 2024.

[11] Benjamin Chetioui. magnoliac: A magnolia compiler, June 2022.
URL: `https://doi.org/10.5281/zenodo.6615298`.

[12] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, sep 2000. ISSN 0362-1340. doi: 10.1145/357766.351266.
URL: `https://doi.org/10.1145/357766.351266`.

[13] Arthur Lisboa Corgozinho, Marco Tulio Valente, and Henrique Rocha. How developers implement property-based tests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 380–384, 2023. doi: 10.1109/ICSME58846.2023.00049.
URL: `https://www.researchgate.net/publication/373829403_How_Developers_Implement_Property-Based_Tests`.

[14] Oracle Corporation. Java software — oracle, .
URL: `https://www.oracle.com/java/`. Accessed: 1 Jun 2024.

[15] Oracle Corporation. Package javax.annotation.processing, .
URL: `https://docs.oracle.com/en/java/javase/17/docs/api/java.compiler/javax/annotation/processing/package-summary.html`. Accessed: 1 Jun 2024.

[16] Oracle Corporation. Comparable (java se 17 and jdk 17), .
URL: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Comparable.html`. Accessed: 29 May 2024.

[17] Oracle Corporation. Elementtype (java se 17 and jdk 17), .
URL: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/annotation/ElementType.html`. Accessed: 1 Jun 2024.

[18] Oracle Corporation. Integer (java se 17 and jdk 17), .
URL: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Integer.html#signum(int)`. Accessed: 29 May 2024.

[19] Oracle Corporation. Processor (java se 17 and jdk 17), .
URL: `https://docs.oracle.com/en/java/javase/17/docs/api/java.compiler/javax/annotation/processing/Processor.html`. Accessed: 1 Jun 2024.

[20] Oracle Corporation. Overview java se 17 jdk 17 2023, Jul 2023.
URL: `https://docs.oracle.com/en/java/javase/17/docs/api/`.

[21] Oracle Corporation. Package java.lang.reflect, Jul 2023.
URL: `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/reflect/package-summary.html`.

[22] Oracle Corporation. Oracle java se support roadmap, 3 2024.
URL: `https://www.oracle.com/java/technologies/java-se-support-roadmap.html`. Accessed: May 24 2024.

[23] Eclipse Foundation. Eclipse survey 2010 report (final), 2010.
URL: `https://www.eclipse.org/org/community_survey/Eclipse_Survey_2010_Report.pdf`. Accessed: May 18, 2024.

[24] F# Software Foundation. F# software foundation, .
URL: `https://fsharp.org/`. Accessed: Fri 31 May 2024.

[25] The Standard C++ Foundation, .
URL: `https://isocpp.org/wiki/faq/cpp0x-concepts-history#cpp0x-concepts`. Accessed: Sat Apr 27 2024.

[26] Martin Fowler. Xunit.
URL: `https://martinfowler.com/bliki/Xunit.html`. Accessed: February 24, 2024.

[27] Triera Gashi, Sophie Bosio, Joachim Kristensen, and Michael Kirkedal Thomsen. pun: Fun with properties; towards a programming language with built-in facilities for program validation. 09 2023.
URL: `https://www.researchgate.net/publication/373838243_textsfpun_Fun_with_Properties_Towards_a_P`
`in_Facilities_for_Program_Validation`.

[28] Zac Hatfield-Dodds. Falsify your software: validating scientific code with property-based testing. pages 162–165, 01 2020. doi: 10.25080/Majora-342d178e-016.
URL: `https://www.researchgate.net/publication/343232178_Falsify_your_Software_validating_scienti`
`based_testing`.

[29] Magne Haveraaen. Axiom based testing for fun and pedagogy. In Antonio Cerone and Markus Roggenbach, editors, *Formal Methods – Fun for Everybody*, pages 27–57, Cham, 2021. Springer International Publishing. ISBN 978-3-030-71374-4.
URL: `https://link.springer.com/chapter/10.1007/978-3-030-71374-4_2`.

[30] Magne Haveraaen and Karl Trygve Kalleberg. JAxT and JDI: The simplicity of JUnit applied to axioms and data invariants. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 731–732, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-220-7. doi: 10.1145/1449814.1449834.
URL: `https://dl.acm.org/doi/10.1145/1449814.1449834`.

[31] ECMA International. Ecma-404 the json data interchange syntax.
URL: `https://ecma-international.org/publications-and-standards/standards/ecma-404/`. Accessed on: 1 Jun 2024.

[32] JavaParser. Javaparser: Analyse, transform and generate your java codebase.
URL: `https://javaparser.org/`. Accessed: February 24, 2024.

[33] JetBrains. Jetbrains: Essential tools for software developers and teams, .
URL: `https://jetbrains.com`. Accessed: May 31, 2024.

[34] JetBrains. Kotlin programming language, .
URL: `https://kotlinlang.org/`. Accessed: 1 Jun 2024.

[35] JetBrains. Java programming - the state of developer ecosystem in 2023 infographic — jetbrains: Developer tools for professionals and teams, 2023.
URL: `https://www.jetbrains.com/lp/devecosystem-2023/java/`. Accessed: January 21, 2024.

[36] JetBrains. Create custom inspections, April 2024.
URL: `https://www.jetbrains.com/help/idea/creating-custom-inspections.html`. Accessed on: 27.05.2024.

[37] JUnit. Junit 5, .
URL: `https://junit.org/junit5/`. Accessed: February 24, 2024.

[38] JUnit. Annotation type parameterizedtest, .
URL: `https://junit.org/junit5/docs/5.0.2/api/org/junit/jupiter/params/ParameterizedTest.html`. Accessed: 2 Jun 2024.

[39] GitHub Staff Kyle Daigle. Octoverse: The state of open source and rise of ai in 2023, November 2023.
URL: `https://github.blog/2023-11-08-the-state-of-open-source-and-ai/`. Accessed: May 28 2024.

[40] Barbara Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN Not.*, 23(5):17–34, jan 1987. ISSN 0362-1340. doi: 10.1145/62139.62141.
URL: https://doi.org/10.1145/62139.62141.

[41] Project Lombok. Project lombok, 2024.
URL: https://projectlombok.org. Accessed: 2 Jun 2024.

[42] David R. MacIver. What is hypothesis?, 2016.
URL: https://hypothesis.works/articles/what-is-hypothesis/. Accessed: Sat 11 May 2024.

[43] Maven. Maven surefire plugin.
URL: https://maven.apache.org/surefire/maven-surefire-plugin/. Accessed 1 Jun 2024.

[44] Mads Bårvåg Nesse. Axiom based testing in java, 2024.
URL: https://github.com/madsnesse/axiom-based-testing-java. Accessed 1 Jun 2024.

[45] Mads Bårvåg Nesse. Jaxioms example, 2024.
URL: https://github.com/madsnesse/PositionExample. Accessed 1 Jun 2024.

[46] Oracle Corporation. Java™ platform, standard edition –api specification, version 17, .
URL: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object). Accessed on: 20.02.2024.

[47] Oracle Corporation. Comparable (java se 17 and jdk 17), .
URL: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Comparable.html. Accessed on: 20.02.2024.

[48] Oracle Corporation. Random (java se 17 and jdk 17), .
URL: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Random.html. Accessed on: 30.05.2024.

[49] Oracle Corporation. Lesson: Annotations the java™ tutorials - learning the java language, .
URL: https://docs.oracle.com/javase/tutorial/java/annotations/. Accessed on: 31.05.2024.

[50] Oracle Corporation, .
URL: https://www.java.com/en/download/help/whatis_java.html. Accessed: Tue May 28 2024.

[51] Stack Overflow. Stack overflow developer survey 2023, 2023.
URL: `https://survey.stackoverflow.co/2023/#section-most-popular-technologies-integrated-development-environment`. Accessed: May 18, 2024.

[52] ronmamo. Java runtime metadata analysis.
URL: `https://github.com/ronmamo/reflections`.

[53] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. *JavaParser: Visited.* http://leanpub.com/javaparservisited, july 2023.
URL: `http://leanpub.com/javaparservisited`.

[54] Eric Sommerlade, Michael Feathers, Jerome Lacoste, Baptiste Lepilleur, Bastiaan Bakker, and Steve Robbins. Cppunit - the unit testing library.
URL: `https://people.freedesktop.org/~mmohrhard/cppunit/index.html`. Accessed: Fri 31 May 2024.

[55] Unknown GitHub Staff. The top programming languages, 2022.
URL: `https://octoverse.github.com/2022/top-programming-languages`. Accessed: May 28 2024.

[56] Bjarne Strousrup. `https://www.drdobbs.com/cpp/the-c0x-remove-concepts-decision/218600111`. Accessed: Tue May 28 2024 through `https://jacobfilipp.com/DrDobbs/articles/DDJ/2009/0908/0908bs01/0908bs01.html` (Archived dr dobbs article).

[57] JUnit 5 team. Open test alliance for the jvm.
URL: `https://github.com/ota4j-team/opentest4j`. Accessed: 2 Jun 2024.

[58] David Valentin. Catsfoot: Introduction.
URL: `https://catsfoot.sourceforge.net/index.html`. Accessed: Sat Apr 27 2024.

[59] Danny van Bruggen. Setting java 8, 9, 10, etc, 2018.
URL: `https://javaparser.org/setting-java-8-9-10-etc/`. Accessed 29 May 2024.

[60] Scott Wlaschin. Choosing properties for property-based testing, December 2014.
URL: `https://fsharpforfunandprofit.com/posts/property-based-testing-2/`. Accessed: Fri May 3 2024.

Appendix A

Examples

```java
public abstract class Generator<T>{

    public Random random;

    public abstract T generate();

    public Generator() {
        this.random = new Random();
    }
    public Generator(Random r) {
        this.random = r;
    }
}
```

Listing A.1: Abstract class for defining generators, also available at [44]

```java
@InheritAxioms
public class Position implements Comparable<Position>, Group<Position>
    ↪ {
    private final int x;
    private final int y;

    public Position(int x, int y) {
        this.x = x;
        this.y = y;
    }
    //From Comparable
    @Override
    public int compareTo(Position o) {
        return Integer.compare(this.x + this.y, o.x + o.y);
    }
    //From Group
    @Override
    public Position binaryOperation(Position position) {
        return new Position(this.x + position.x, this.y + position.y);
    }

    @Override
    public Position inverse() {
        return new Position(-this.x, -this.y);
    }

    @Override
    public Position identity() {
        return new Position(0,0);
    }

    //From Object
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass() ) return false;

        Position position = (Position) o;
        return x == position.x && y == position.y;
    }

    @Override
    public int hashCode() {
        int result = x;
        result = 31 * result + y;
        return result;
    }

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }

    //Getters
    ...
}
```

Figure A.1: Position class, representing a position in two-dimensional space, also available at [45]

```java
@InheritAxioms
public class Position3D extends Position {
    private final int z;

    // constructors

    @Override
    public Position identity() {
        Position p = new Position(0,0).identity();
        return new Position3D(p.getX(), p.getY(), 0);
    }

    @Override
    public Position inverse() {
        Position p = new Position(this.getX(), this.getY()).inverse();
        return new Position3D(p.getX(), p.getY(), -this.z);
    }

    @Override
    public Position binaryOperation(Position p) {
        Position q = super.binaryOperation(p);
        if (p instanceof Position3D) {
            return new Position3D(q.getX(), q.getY(), this.getZ() +
                ↪ ((Position3D) p).getZ());
        }
        return q;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        if (!super.equals(o)) return false;

        Position3D that = (Position3D) o;
        return z == that.z;
    }

    @Override
    public int hashCode() {
        int result = super.hashCode();
        result = 31 * result + z;
        return result;
    }
}
```

Figure A.2: A class representing a position in three dimensions, extending Position from Figure A.1, also available at [45]

Appendix B

Axioms for Object and Comparable

```
package axioms;

/**
 * Axioms for the {@link Object} class.
 * As defined in {@link <a
    ↪ href="https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/la
 * Code inspired by JAxT {@link <a
    ↪ href="https://www.ii.uib.no/mouldable/testing/jaxt/index.html">...</a>}
 */
public class ObjectAxioms {

    /**
     * It is reflexive: for any non-null reference value x,
        ↪ x.equals(x) should return true.
     */
    @AxiomForExistingClass(className = "java.lang.Object")
    public static void equalsIsReflexive(Object o) {
        assertEquals(o,o);
    }
    /**
     * It is symmetric: for any non-null reference values x and y,
        ↪ x.equals(y)
     * should return true if and only if y.equals(x) returns true.
     */
    @AxiomForExistingClass(className = "java.lang.Object")
    public static void equalsIsSymmetric(Object x, Object y) {
        assertEquals(x.equals(y), y.equals(x));
    }
    /**
```

```
     * It is transitive: for any non-null reference values x, y, and
        ↪ z,
     * if x.equals(y) returns true and y.equals(z) returns true,
        ↪ then x.equals(z) should return true.
     */
    @AxiomForExistingClass(className = "java.lang.Object")
    public static void equalsIsTransitive(Object x, Object y, Object
        ↪ z) {
        if (x.equals(y) && y.equals(z)) {
            assertEquals(x, z);
        }
    }
    /**
     * For any non-null reference value x, x.equals(null) should
        ↪ return false.
     */
    @AxiomForExistingClass(className = "java.lang.Object")
    public static void equalsNullIsFalse(Object x) {
        assertEquals(false, x.equals(null));
    }

    /**
     * If two objects are equal according to the {@code equals}
        ↪ method,
     * then calling the {@code hashCode} method on each of the two
        ↪ objects must produce the same integer result.
     */
    @AxiomForExistingClass(className = "java.lang.Object")
    public static void hashCodeCongruenceOnEquals(Object x, Object
        ↪ y) {
        if (x.equals(y)) {
            assertEquals(x.hashCode(), y.hashCode());
        }
    }

}
```

Listing B.1: Axioms for `java.lang.Object`, as described in [46]. Definitions based on [30, 7], also available at [44]

```
package axioms;

import no.uib.ii.annotations.AxiomForExistingClass;
```

```java

import static no.uib.ii.StaticMethods.assertEquals;
import static no.uib.ii.StaticMethods.assertTrue;

/**
 * Axioms for the {@link Comparable} interface.
 * As defined in {@link <a
    ↪ href="https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/la
 * Code inspired by JAxT {@link <a
    ↪ href="https://www.ii.uib.no/mouldable/testing/jaxt/index.html">...</a>}
 */
public class ComparableAxioms {

    /**
     * The implementor must ensure signum(x.compareTo(y)) ==
        ↪ -signum(y.compareTo(x)) for all x and y.
     * (This implies that x.compareTo(y) must throw an exception if
        ↪ and only if y.compareTo(x) throws an exception.)
     */
    @AxiomForExistingClass(className = "java.lang.Comparable")
    public static <T extends Comparable<T>> void
       ↪ compareToConsistentWithSignum(T x, T y) {
        try {
            var left = java.lang.Integer.signum(x.compareTo(y));
            assertEquals(left,
               ↪ -java.lang.Integer.signum(y.compareTo(x)));
        } catch (Exception e) {
            try {
                var right =
                   ↪ -java.lang.Integer.signum(y.compareTo(x));
            } catch (Exception ex) {
                assertTrue(true);
            }
            assertTrue(false);
        }
    }

    /**
     * The implementor must also ensure that the relation is
        ↪ transitive:
     * (x.compareTo(y) > 0 && y.compareTo(z) > 0) implies
        ↪ x.compareTo(z) > 0.
     */
    @AxiomForExistingClass(className = "java.lang.Comparable")
    public static <T extends Comparable<T>> void
```

```
        ↪ compareToTransitive(T x, T y, T z) {
        if (x.compareTo(y) > 0 && y.compareTo(z) > 0) {
            assertEquals(true, x.compareTo(z) > 0);
        } else assertTrue(true);
    }

    /**
     * It is strongly recommended, but not strictly required that
        ↪ (x.compareTo(y)==0) == (x.equals(y)).//TODO create
        ↪ support for recommended axioms
     * Generally speaking, any class that implements the Comparable
        ↪ interface
     * and violates this condition should clearly indicate this
        ↪ fact. The recommended language is
     * "Note: this class has a natural ordering that is inconsistent
        ↪ with equals."
     */
    @AxiomForExistingClass(className = "java.lang.Comparable")
    public static <T extends Comparable<T>> void
        ↪ compareToEqualsConsistent(T x, T y) {
        assertEquals(x.compareTo(y) == 0, x.equals(y));
    }
}
```

Listing B.2: Axioms for `java.lang.Comparable`, as described in [47]. Definitions based on [30, 7], also available at [44]

Appendix C

Generated Code

```java
public class PositionGeneratedTest {

    @ParameterizedTest()
    @DisplayName(value = "compareToConsistentWithSignum < {0},{1} >")
    @MethodSource("factorycompareToConsistentWithSignum")
    public void compareToConsistentWithSignum(org.example.Position
    ↪ x, org.example.Position y) {
        assertEquals(java.lang.Integer.signum(x.compareTo(y)),
            ↪ -java.lang.Integer.signum(y.compareTo(x)));
    }

    public static Stream<Arguments>
        ↪ factorycompareToConsistentWithSignum() {
        Generator<Position> clazzGenerator = new PositionGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
                ↪ clazzGenerator.generate()), Named.of("Argument
                ↪ 2:", clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "compareToTransitive < {0},{1},{2} >")
    @MethodSource("factorycompareToTransitive")
    public void compareToTransitive(org.example.Position x,
    ↪ org.example.Position y, org.example.Position z) {
        if (x.compareTo(y) < 0 && y.compareTo(z) < 0) {
            assertEquals(true, x.compareTo(z) < 0);
        } else
            assertTrue(true);
    }

    public static Stream<Arguments> factorycompareToTransitive() {
        Generator<Position> clazzGenerator = new PositionGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
               ↪ clazzGenerator.generate()), Named.of("Argument
               ↪ 2:", clazzGenerator.generate()),
               ↪ Named.of("Argument 3:",
               ↪ clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "inverseAxiom < {0} >")
    @MethodSource("factoryinverseAxiom")
    void inverseAxiom(org.example.Position a) {
        assertEquals(a.identity(), a.binaryOperation(a.inverse()));
    }

    public static Stream<Arguments> factoryinverseAxiom() {
        Generator<Position> clazzGenerator = new PositionGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
               ↪ clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsIsReflexive < {0} >")
    @MethodSource("factoryequalsIsReflexive")
    public void equalsIsReflexive(org.example.Position o) {
        assertEquals(o, o);
    }

    public static Stream<Arguments> factoryequalsIsReflexive() {
        Generator<Position> clazzGenerator = new PositionGenerator();
        List<Arguments> clazzStream = new ArrayList();
```

```java
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
               ↪ clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsIsSymmetric < {0},{1} >")
    @MethodSource("factoryequalsIsSymmetric")
    public void equalsIsSymmetric(org.example.Position x,
   ↪ org.example.Position y) {
        assertEquals(x.equals(y), y.equals(x));
    }

    public static Stream<Arguments> factoryequalsIsSymmetric() {
        Generator<Position> clazzGenerator = new PositionGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
               ↪ clazzGenerator.generate()), Named.of("Argument
               ↪ 2:", clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsIsTransitive < {0},{1},{2} >")
    @MethodSource("factoryequalsIsTransitive")
    public void equalsIsTransitive(org.example.Position x,
   ↪ org.example.Position y, org.example.Position z) {
        if (x.equals(y) && y.equals(z)) {
            assertEquals(x, z);
        }
    }

    public static Stream<Arguments> factoryequalsIsTransitive() {
        Generator<Position> clazzGenerator = new PositionGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
               ↪ clazzGenerator.generate()), Named.of("Argument
               ↪ 2:", clazzGenerator.generate()),
               ↪ Named.of("Argument 3:",
               ↪ clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsNullIsFalse < {0} >")
    @MethodSource("factoryequalsNullIsFalse")
    public void equalsNullIsFalse(org.example.Position x) {
        assertEquals(false, x.equals(null));
    }

    public static Stream<Arguments> factoryequalsNullIsFalse() {
        Generator<Position> clazzGenerator = new PositionGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
               ↪ clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "hashCodeCongruenceOnEquals < {0},{1} >")
    @MethodSource("factoryhashCodeCongruenceOnEquals")
    public void hashCodeCongruenceOnEquals(org.example.Position x,
   ↪ org.example.Position y) {
        if (x.equals(y)) {
            assertEquals(x.hashCode(), y.hashCode());
        }
    }

    public static Stream<Arguments>
       ↪ factoryhashCodeCongruenceOnEquals() {
        Generator<Position> clazzGenerator = new PositionGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
               ↪ clazzGenerator.generate()), Named.of("Argument
               ↪ 2:", clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }
}
```

Listing C.1: Generated test classes for Position A.1

```java
public class Position3DGeneratedTest {

    @ParameterizedTest()
    @DisplayName(value = "binaryOperationWithPositionReturnsPosition
    ↪ < {0} >")
    @MethodSource("factorybinaryOperationWithPositionReturnsPosition")
    public void
    ↪ binaryOperationWithPositionReturnsPosition(org.example.Position3D
    ↪ p) {
        org.example.Position q = new org.example.Position(1, 1);
        org.example.Position newP = p.binaryOperation(q);
        assertEquals(org.example.Position.class, newP.getClass());
    }

    public static Stream<Arguments>
        ↪ factorybinaryOperationWithPositionReturnsPosition() {
        Generator<Position3D> clazzGenerator = new
            ↪ Position3DGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
                ↪ clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "compareToConsistentWithSignum < {0},{1} >")
    @MethodSource("factorycompareToConsistentWithSignum")
    public void compareToConsistentWithSignum(org.example.Position
    ↪ x, org.example.Position y) {
        assertEquals(java.lang.Integer.signum(x.compareTo(y)),
            ↪ -java.lang.Integer.signum(y.compareTo(x)));
    }

    public static Stream<Arguments>
        ↪ factorycompareToConsistentWithSignum() {
        Generator<Position3D> clazzGenerator = new
            ↪ Position3DGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
                ↪ clazzGenerator.generate()), Named.of("Argument
                ↪ 2:", clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "compareToTransitive < {0},{1},{2} >")
    @MethodSource("factorycompareToTransitive")
    public void compareToTransitive(org.example.Position x,
    ↪ org.example.Position y, org.example.Position z) {
        if (x.compareTo(y) < 0 && y.compareTo(z) < 0) {
            assertEquals(true, x.compareTo(z) < 0);
        } else
            assertTrue(true);
    }

    public static Stream<Arguments> factorycompareToTransitive() {
        Generator<Position3D> clazzGenerator = new
            ↪ Position3DGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
                ↪ clazzGenerator.generate()), Named.of("Argument
                ↪ 2:", clazzGenerator.generate()),
                ↪ Named.of("Argument 3:",
                ↪ clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "inverseAxiom < {0} >")
    @MethodSource("factoryinverseAxiom")
    void inverseAxiom(org.example.Position a) {
        assertEquals(a.identity(), a.binaryOperation(a.inverse()));
    }

    public static Stream<Arguments> factoryinverseAxiom() {
        Generator<Position3D> clazzGenerator = new
            ↪ Position3DGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
                ↪ clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsIsReflexive < {0} >")
    @MethodSource("factoryequalsIsReflexive")
    public void equalsIsReflexive(org.example.Position o) {
        assertEquals(o, o);
    }

    public static Stream<Arguments> factoryequalsIsReflexive() {
        Generator<Position3D> clazzGenerator = new
            ↪ Position3DGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
                ↪ clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsIsSymmetric < {0},{1} >")
    @MethodSource("factoryequalsIsSymmetric")
    public void equalsIsSymmetric(org.example.Position x,
    ↪ org.example.Position y) {
        assertEquals(x.equals(y), y.equals(x));
    }

    public static Stream<Arguments> factoryequalsIsSymmetric() {
        Generator<Position3D> clazzGenerator = new
            ↪ Position3DGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
                ↪ clazzGenerator.generate()), Named.of("Argument
                ↪ 2:", clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsIsTransitive < {0},{1},{2} >")
    @MethodSource("factoryequalsIsTransitive")
    public void equalsIsTransitive(org.example.Position x,
    ↪ org.example.Position y, org.example.Position z) {
        if (x.equals(y) && y.equals(z)) {
            assertEquals(x, z);
        }
    }

    public static Stream<Arguments> factoryequalsIsTransitive() {
        Generator<Position3D> clazzGenerator = new
           ↪ Position3DGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
               ↪ clazzGenerator.generate()), Named.of("Argument
               ↪ 2:", clazzGenerator.generate()),
               ↪ Named.of("Argument 3:",
               ↪ clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsNullIsFalse < {0} >")
    @MethodSource("factoryequalsNullIsFalse")
    public void equalsNullIsFalse(org.example.Position x) {
        assertEquals(false, x.equals(null));
    }

    public static Stream<Arguments> factoryequalsNullIsFalse() {
        Generator<Position3D> clazzGenerator = new
           ↪ Position3DGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
               ↪ clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }

    @ParameterizedTest()
    @DisplayName(value = "hashCodeCongruenceOnEquals < {0},{1} >")
    @MethodSource("factoryhashCodeCongruenceOnEquals")
    public void hashCodeCongruenceOnEquals(org.example.Position x,
   ↪ org.example.Position y) {
        if (x.equals(y)) {
            assertEquals(x.hashCode(), y.hashCode());
        }
    }

    public static Stream<Arguments>
        ↪ factoryhashCodeCongruenceOnEquals() {
        Generator<Position3D> clazzGenerator = new
            ↪ Position3DGenerator();
        List<Arguments> clazzStream = new ArrayList();
        for (int i = 0; i < 100; i++) {
            clazzStream.add(Arguments.of(Named.of("Argument 1:",
                ↪ clazzGenerator.generate()), Named.of("Argument
                ↪ 2:", clazzGenerator.generate())));
        }
        return clazzStream.stream();
    }
}
```

Listing C.2: Generated test classes for Position3D A.2

```java
public class PersonGeneratedTest {

    @ParameterizedTest()
    @DisplayName(value = "equalNameAndAgeShouldBeEqual < {0} >")
    @MethodSource("factoryequalNameAndAgeShouldBeEqual")
    public void equalNameAndAgeShouldBeEqual(org.example.Person p) {
        org.example.Person q = new Person(p.name, p.age);
        assertEquals(p, q);
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsIsReflexive < {0} >")
    @MethodSource("factoryequalsIsReflexive")
    public void equalsIsReflexive(org.example.Person o) {
        assertEquals(true, o.equals(o));
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsIsSymmetric < {0},{1} >")
    @MethodSource("factoryequalsIsSymmetric")
    public void equalsIsSymmetric(org.example.Person x,
    ↪ org.example.Person y) {
        assertEquals(x.equals(y), y.equals(x));
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsIsTransitive < {0},{1},{2} >")
    @MethodSource("factoryequalsIsTransitive")
    public void equalsIsTransitive(org.example.Person x,
    ↪ org.example.Person y, org.example.Person z) {
        if (x.equals(y) && y.equals(z)) {
            assertEquals(x, z);
        }
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsIsConsistent < {0},{1} >")
    @MethodSource("factoryequalsIsConsistent")
    public void equalsIsConsistent(org.example.Person x,
    ↪ org.example.Person y) {
        assertEquals(x.equals(y), x.equals(y));
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsNullIsFalse < {0} >")
    @MethodSource("factoryequalsNullIsFalse")
    public void equalsNullIsFalse(org.example.Person x) {
        assertEquals(x.equals(null), false);
    }

    @ParameterizedTest()
    @DisplayName(value = "equalsHashCodeCongruence < {0},{1} >")
    @MethodSource("factoryequalsHashCodeCongruence")
    public void equalsHashCodeCongruence(org.example.Person a,
    ↪ org.example.Person b) {
        if (a.equals(b)) {
            assertEquals(a.hashCode(), b.hashCode());
        }
    }
}
```

Listing C.3: Parts of the generated tests for Person4.6

Appendix D

Example Project

Listing D.1: Pom.xml for example project, also available at [45]

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            ↪ http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>jaxioms-test</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>17</maven.compiler.source>
        <maven.compiler.target>17</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <junit.version>5.10.1</junit.version>
        <jaxioms.version>1.0.7-STABLE</jaxioms.version><!--  newest
           ↪ jaxioms version    -->
        <surefire.version>3.2.5</surefire.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>no.uib.ii</groupId>
            <artifactId>processors</artifactId>
            <version>${jaxioms.version}</version>
        </dependency>
        <dependency>
            <groupId>no.uib.ii</groupId>
            <artifactId>generator</artifactId>
            <version>${jaxioms.version}</version>
        </dependency>
        <dependency>
            <groupId>no.uib.ii</groupId>
            <artifactId>common</artifactId>
            <version>${jaxioms.version}</version>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>${junit.version}</version>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-params</artifactId>
            <version>${junit.version}</version>
        </dependency>
    </dependencies>
    <build>
        <testSourceDirectory>
            ${project.build.directory}/generated-sources <!--  enable
               ↪ automatically running generated tests   -->
        </testSourceDirectory>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId> <!--  run
                   ↪ tests with surefire plugin, allowing for
                   ↪ reporting     -->
                <artifactId>maven-surefire-plugin</artifactId>
                <version>${surefire.version}</version>
            </plugin>
        </plugins>
    </build>
</project>
```

Glossary

API Application Programming Interface, interface for other programs or other parts of program to interact with through calling functions..

AST Abstract Syntax Tree, tree representation of a program.

Axiom-based testing Also known as Property-based testing, testing software by defining properties that should always hold for certain parts of the program.

IDE Integrated development environment, a text editor with extra features to enable easy development.

JAxT A framework created for Axiom-based testing as an Eclipse plugin by Karl Trygve Kalleberg and Magne Haveraaen [30, 7].

Property-based testing Property-based testing, sometimes used synonymously as Axiom-based testing, but focuses on properties of functions instead of properties of classes.