A report about Program Verification using JML and openJML

Francesco Paolo Di Lorenzo
student ID: 1712990
e-mail: dilorenzo.1712990@studenti.uniroma1.it

Contents

Introduction

This report shows how to use program verification tools like `openJML` on a Java class.
`Program verification` aims to use formal proofs to demonstrate that programs behave according to formal specifications [1].
In the first section there is a high-level description of the tool.
The second section describes the formal specifications that the class subject to verification should verify.
The third section, shows and describes the output of the tool and the corrections made to solve all the warnings reported by openJML. The fourth section shows the final version of the program.
The fifth section suggests further changes to be made to the Java class to make the class safe from multiple write accesses, from multiple threads, to shared resources.

1 OpenJML

OpenJML is a program verification tool for Java programs that allows you to check the specifications of programs annotated in the `Java Modeling Language`.
It can be used in operating systems such as Linux, Mac OS X and Windows, as long as a JDK for java 8 is installed.
OpenJML can be installed as a command line tool, as an Eclipse plugin or programmatically from a user's Java program [2].
For this verification it was used as an Eclipse plugin, which incorporates the functionality of the command-line tool and allows you to interactively explore the results of failed checks.
As mentioned before OpenJML allows to check the specifications of programs annotated in the **J**ava **M**odelling **L**anguage: JML is a behavioral interface specification language for formally specifying,using annotations, the behavior of java classes and methods [3].
Assertions will be added to the code (preconditions, postconditions and invariants), specified in comment and the properties to be respected are specified using java syntax, in particular as boolean expressions.
Pre-condition for method can be specified using `requires`.
Post-condition for method can be specified using `ensures`.
Invariants are properties that must be maintained by all methods and are implicitly included in all pre- and post-conditions [7].

2 Formal specifications of Taxpayer class

OpenJML, in this formal verification, was used to verify a Java class (called Taxpayer), used in an information system at the tax office to record information about individuals.
The tax office requires certain specifications to be verified,this is a summary of these specifications which can be easily divided into 3 parts:

1. Persons are either male or female. **(Part 1)**
2. Persons can only marry people of the opposite sex.**(Part 1)**
3. The tax system uses an income tax allowance (deduction): Every person has an income tax allowance on which no tax is paid.
   There is a default tax allowance of 5000 per individual and only income above this amount is taxed. Married persons can pool their tax allowance, as long as the sum of their tax allowances remains the same. For example, the wife could have a tax allowance of 9000, and the husband a tax allowance of 1000. **(Part 2)**
4. The new government introduces a measure that people aged 65 and over have a higher tax allowance, of 7000. The rules for pooling tax allowances remain the same.**(Part 3)**

Note: These specifications mostly show which properties this class must check. Some properties may arise as a result of others and even if they are not mentioned in the specifications they must be verified

3 OpenJML output description

This section quickly describes the java class to be checked, shows the warnings obtained from the execution of openJML on it and the choices made for the elimination of these warnings.
Opening the java class on Eclipse, the first thing that catches the eye is the compiler that complains about some compilation errors.
This is the situation:

```
Taxpayer(boolean babyboy, Taxpayer ma, Taxpayer pa) {
  age = 0;
  isMarried = false;
  this.isMale = babyboy;
  this.isFemale = !babyboy;
  mother = ma;
  father = pa;
  spouse = null;
  income = 0;
  tax_allowance = DEFAULT_ALLOWANCE;
  /* The line below makes explicit the assumption that a↩
      new Taxpayer is not
   * married to anyone yet*/
  //@ assume (\forall Taxpayer p; p.spouse != this);
}
```

`Both mother and father "cannot be resolved to a variable"`, so a good choise(bearing in mind that this class is used in an information system) is to make mother and father class fields:

```
...
Taxpayer mother;
Taxpayer father;
...
Taxpayer(boolean babyboy, Taxpayer ma, Taxpayer pa) {
  age = 0;
  isMarried = false;
  this.isMale = babyboy;
  this.isFemale = !babyboy;
  mother = ma;
  father = pa;
  spouse = null;
```

```
    income = 0;
    tax_allowance = DEFAULT_ALLOWANCE;
    /* The line below makes explicit the assumption that a↩
        new Taxpayer is not
     * married to anyone yet*/
    //@ assume (\forall Taxpayer p; p.spouse != this);
  }
```

Now you can run openJML. As mentioned above running openJML as eclipse plugin,allows you to interactively explore the results of failed checks. There are two outputs, one in which the result of a static check is shown and through the use of colors the valid and invalid methods are highlighted:

```
<package-name>
 Taxpayer
   [INVALID] divorce()[0.127 z3_4_3]
   [INVALID] haveBirthday()[0.239 z3_4_3]
   [VALID]   marry(Taxpayer)[0.125 z3_4_3]
   [INVALID] Taxpayer(boolean,Taxpayer,Taxpayer)[0.251 z3_4_3]
   [INVALID] transferAllowance(int)[0.127 z3_4_3]
```

Another one in which it is explained through a trace because these methods do not pass the formal verification. At the end of the second output we have a summary like this:

```
Completed proving methods in Taxpayer [1.27 secs]
Summary:
    Valid:         1
    Invalid:       4
    Infeasible:    0
    Timeout:       0
    Error:         0
    Skipped:       0
   TOTAL METHODS: 5
   Classes:        0 proved of 1
   Model Classes: 0
   Model methods: 0 proved of 0
   DURATION:         1.4 secs
[2.26] Completed
```

This summary shows that, for the moment, only 1 method out of 5 is valid. First you need to add all those annotations and code changes that allow you to validate the class for static analysis.
After that all the necessary modifications and annotations will be added to make the class valid with regards to the specifications required by the tax office.

3.1 Static analysis

This section shows which warnings are shown by the static analysis of the class and what changes have been made to make it valid.
As seen before only one method out of 5 is considered valid by openJML. The tool produces warnings on these methods:

- Taxpayer.Taxpayer(boolean,Taxpayer,Taxpayer)
- Taxpayer.divorce()
- Taxpayer.haveBirthday()
- Taxpayer.transferAllowance(int)

Although the tool does not produce warnings regarding the Taxpayer.marry(Taxpayer) qmethod, it has been modified as it was not logically correct:

```
void marry(Taxpayer new_spouse) {
 spouse = new_spouse;
 isMarried = true;
}
```

The method does not update the information on the marriage just occurred as regards the taxpayer referenced by the input parameter `new_spouse` and therefore also by the `spouse` variable.
This is the correct version:

```
void marry(Taxpayer new_spouse) {
  spouse = new_spouse;
  spouse.spouse = this;
  spouse.isMarried = true;
  isMarried = true;
}
```

3.1.1 Taxpayer.Taxpayer(boolean,Taxpayer,Taxpayer)

This is the constructor of the Taxpayer class.

```
37  Taxpayer(boolean babyboy, Taxpayer ma, Taxpayer pa) {
38    age = 0;
39    isMarried = false;
40    this.isMale = babyboy;
41    this.isFemale = !babyboy;
42    mother = ma;
43    father = pa;
44    spouse = null;
45    income = 0;
46    tax_allowance = DEFAULT_ALLOWANCE;
47  }
```

```
Taxpayer.Taxpayer(boolean,Taxpayer,Taxpayer): The prover cannot establish an
    assertion (PossiblyNullAssignment) in method Taxpayer
```

OpenJML mainly complains that within the constructor there is a *PossiblyNullAssignment*. A PossibilyNullAssignment is when the value assigned to a variable or field declared NonNull must not be null [4].
This happens because the fields that refer to objects (arrays etc ..) are considered in JML as `non_null` by default [5].
To solve it is necessary to use the `/*@ nullable @*/` annotation in order to point out that to a variable can also be assigned a null value.
Here we are talking about the `spouse` variable which is of the Taxpayer type, so just add this annotation before the type of the variable, to solve this warning.
In fact, just think that there are two other variables that refer to objects (mother, father), so it's a good idea (even if no warnings occur on these two), do the same.

```
/*@ nullable @*/ Taxpayer mother;

 /*@ nullable @*/ Taxpayer father;

 /* Reference to spouse if person is married, null ↩
    otherwise */
 /*@ nullable @*/ Taxpayer spouse;
 }
```

```
37  Taxpayer(boolean babyboy, Taxpayer ma, Taxpayer pa) {
38    age = 0;
39    isMarried = false;
40    this.isMale = babyboy;
41    this.isFemale = !babyboy;
42    mother = ma;
43    father = pa;
44    spouse = null;
45    income = 0;
46    tax_allowance = DEFAULT_ALLOWANCE;
47  }
```

```
<package-name>
 Taxpayer
  [INVALID] divorce()[0.127 z3_4_3]
  [INVALID] haveBirthday()[0.239 z3_4_3]
  [VALID]   marry(Taxpayer)[0.125 z3_4_3]
  [VALID] Taxpayer(boolean,Taxpayer,Taxpayer)[0.251 z3_4_3]
  [INVALID] transferAllowance(int)[0.127 z3_4_3]
```

Now the costructor seems to be ok.

```
  Completed proving methods in Taxpayer [1.24 secs]
Summary:
    Valid:         2
    Invalid:       3
    Infeasible:    0
    Timeout:       0
    Error:         0
    Skipped:       0
   TOTAL METHODS: 5
   Classes:       0 proved of 1
   Model Classes: 0
   Model methods: 0 proved of 0
   DURATION:          1.3 secs
[2.54] Completed
```

3.1.2 Taxpayer.divorce()

```
void divorce() {
  spouse.spouse = null;
  spouse = null;
  isMarried = false;
 }
```

```
Taxpayer.divorce(): The prover cannot establish an assertion (
    PossiblyNullDeReference) in method divorce
```

OpenJML mainly complains that within this method there is a *PossiblyNullDeference*. A PossibilyNullDeference is when the program can potentially deference a null pointer, raising a NullPointerException [6].
This can happen because `spouse` variable my assume null value. The goal is to ensure that the execution of this method is always safe and that it cannot lead to such problems. One way to do this is to use `//@invariant` and `//@ requires` annotations.
So we can add this invariant:

```
//@ invariant this.isMarried <==> this.spouse != null;
```

and this requirement:

```
//@ requires this.isMarried;
```

With that invariant we make sure that before and after the execution of each method,if a taxpayer is married then the spouse variable cannot be null and vice versa if the spouse variable is not null then the taxpayer must be married.
The requires serves to guarantee that before the execution of the method `divorce()`, the taxpayer is married and consequently the spouse variable (thanks to the invariant) is different from null.
After adding these changes, running openJML again gets this:

```
<package-name>
 Taxpayer
  [INVALID] divorce()[0.127 z3_4_3]
  [INVALID] haveBirthday()[0.239 z3_4_3]
  [VALID]   marry(Taxpayer)[0.125 z3_4_3]
  [VALID] Taxpayer(boolean,Taxpayer,Taxpayer)[0.251 z3_4_3]
  [INVALID] transferAllowance(int)[0.127 z3_4_3]
```

```
Completed proving methods in Taxpayer [1.04 secs]
Summary:
    Valid:         2
    Invalid:       3
    Infeasible:    0
    Timeout:       0
    Error:         0
    Skipped:       0
   TOTAL METHODS: 5
   Classes:       0 proved of 1
   Model Classes: 0
```

```
   Model methods: 0 proved of 0
   DURATION:          1.1 secs
[1.64] Completed
```

The `divorce()` method is still considered invalid by openJML. This happens because there are problems in the method code.

```
void divorce() {
  spouse.spouse = null;
  spouse = null;
  isMarried = false;
 }
```

The method simply nullifies the references of the spouse variable for both spouses and sets only one of them as "unmarried". So to maintain the consistency between the two now ex-spouses, it is necessary to set the variable `isMarried` to false also for the taxpayer from which one divorces, before canceling the reference. This is the correct version of the method, considered valid also by openJML:

```
...

//@ invariant this.isMarried <==> this.spouse != null;
...
//@ requires isMarried;
void divorce() {
  spouse.isMarried=false;
  spouse.spouse = null;
  spouse = null;
  isMarried = false;
 }
```

This is the current openJML output:

```
<package-name>
 Taxpayer
  [VALID] divorce()[0.127 z3_4_3]
  [INVALID] haveBirthday()[0.239 z3_4_3]
  [VALID]   marry(Taxpayer)[0.125 z3_4_3]
  [VALID] Taxpayer(boolean,Taxpayer,Taxpayer)[0.251 z3_4_3]
  [INVALID] transferAllowance(int)[0.127 z3_4_3]
```

```
Completed proving methods in Taxpayer [0.99 secs]
Summary:
    Valid:         3
    Invalid:       2
    Infeasible:    0
    Timeout:       0
    Error:         0
    Skipped:       0
   TOTAL METHODS: 5
   Classes:       0 proved of 1
   Model Classes: 0
   Model methods: 0 proved of 0
   DURATION:          1.1 secs
[1.71] Completed
```

Now even the `divorce()` method is considered valid.

3.1.3 Taxpayer.haveBirthday()

```
void haveBirthday() {
  age++;
 }
```

```
Taxpayer.haveBirthday(): The prover cannot establish an assertion (
    ArithmeticOperationRange) in method haveBirthday: overflow in int sum
```

OpenJML mainly complains that within this method there is an *Arithmetic-OperationRange*. The main problem is that the program can potentially slip into *Integer Overflow*.
An Integer Overflow is the condition that occurs when the result of an arithmetic operation, such as multiplication or addition, exceeds the maximum size of the integer type used to store it [8].
The goal is to ensure that the execution of this method is always safe and that it cannot lead to *Integer Overflows*. One way to do this is to use `//@ invariant` and `//@ requires` annotations like these:

```
//@ invariant age >=0 && age <= Integer.MAX_VALUE;
```

and this requires:

```
//@ requires age+1 <= Integer.MAX_VALUE;
```

With that invariant we make sure that before and after the execution of each method,the integer variable `age` can only takes values between 0 and the maximum possible value for an integer, defined in the variable `Integer.MAX_VALUE`.

The requirement serves to guarantee that before the execution of the method `haveBirthday()`,increasing the `age` variable value by 1 does not lead to *Integer Overflow*.
After adding these changes, running openJML again gets this:

```
<package-name>
 Taxpayer
   [VALID] divorce()[0.127 z3_4_3]
   [VALID] haveBirthday()[0.239 z3_4_3]
   [VALID]   marry(Taxpayer)[0.125 z3_4_3]
   [VALID] Taxpayer(boolean,Taxpayer,Taxpayer)[0.251 z3_4_3]
   [INVALID] transferAllowance(int)[0.127 z3_4_3]
```

```
Completed proving methods in Taxpayer [1.62 secs]
Summary:
    Valid:         4
    Invalid:       1
    Infeasible:    0
    Timeout:       0
    Error:         0
    Skipped:       0
   TOTAL METHODS: 5
   Classes:       0 proved of 1
   Model Classes: 0
   Model methods: 0 proved of 0
   DURATION:          1.7 secs
[2.23] Completed
```

Now even the `haveBirthday()` method is considered valid.

3.1.4 Taxpayer.transferAllowance(int)

```
/* Transfer part of tax allowance to own spouse */
void transferAllowance(int change) {
 tax_allowance = tax_allowance - change;
 spouse.tax_allowance = spouse.tax_allowance + change;
}
```

We get several warnings from this method:

```
Taxpayer.transferAllowance(int): The prover cannot establish an assertion (
    ArithmeticOperationRange) in method transferAllowance: underflow in int
        difference
Taxpayer.transferAllowance(int): The prover cannot establish an assertion (
    ArithmeticOperationRange) in method transferAllowance: overflow in int
        difference
```

The block above refers to the first instruction of the method, one below refers to the second.

```
Taxpayer.transferAllowance(int): The prover cannot establish an assertion (
    PossiblyNullDeReference) in method transferAllowance
Taxpayer.transferAllowance(int): The prover cannot establish an assertion (
    ArithmeticOperationRange) in method transferAllowance: underflow in int
        sum
Taxpayer.transferAllowance(int): The prover cannot establish an assertion (
    ArithmeticOperationRange) in method transferAllowance: overflow in int
        sum
```

Most of these warnings refer to arithmetic operations. In fact the difference that is executed in the first instruction of the method, can lead to *Integer Underflow*.
The sum that occurs in the second statement can instead lead to an Integer Overflow.
Again in the second instruction, accessing the `spouse` variable can lead to a *PossibilyNullDeference*, since one could try to access the `spouse` variable when the taxpayer is not married. One way solve this situation is to use the `//@ requires` annotation:

```
 /*@
 requires change >= 0 && change <= tax_allowance;
 requires isMarried;
 requires change + spouse.tax_allowance <= Integer.↩
    MAX_VALUE;
@*/;
```

In this way we guarantee that the mathod always receives as input a non-negative integer value (`change` variable) and not larger than the value from which it must be subtracted (`tax_allowance`) in the first instruction of the method.
We also guarantee that the taxpayer is married and consequently that the `spouse` variable is not null.
In this way we are sure that in the second instruction a null variable is not deferenced.
Finally with the last requirement we guarantee that the sum that is executed in the second instruction does not lead to an *Integer Overflow*.
Keeping in mind the invariants added to correct the previously analyzed method, some invariants have been added to control the values that the integer variables of the class can take:

```
//@ invariant this.tax_allowance >= 0 && tax_allowance <=↩
    Integer.MAX_VALUE;
//@ invariant   this.income >= 0 && this.income <= Integer↩
   .MAX_VALUE;
```

After these changes, running openJML again gets this:

```
<package-name>
 Taxpayer
  [VALID] divorce()[0.127 z3_4_3]
  [VALID] haveBirthday()[0.239 z3_4_3]
  [VALID]   marry(Taxpayer)[0.125 z3_4_3]
  [VALID] Taxpayer(boolean,Taxpayer,Taxpayer)[0.251 z3_4_3]
  [VALID] transferAllowance(int)[0.127 z3_4_3]
```

```
Completed proving methods in Taxpayer [1.41 secs]
Summary:
    Valid:         5
    Invalid:       0
    Infeasible:    0
    Timeout:       0
    Error:         0
    Skipped:       0
   TOTAL METHODS: 5
   Classes:       1 proved of 1
   Model Classes: 0
   Model methods: 0 proved of 0
   DURATION:          1.5 secs
[2.11] Completed
```

Now all the methods of the class are considered valid.

3.2 Client specifications

Until now the invariants, the requirements and the necessary modifications have been added so that the class is valid if analyzed statically by openJML. The main objective, however, is to ensure that given the specifications (those required by the tax office and shown in chapter 2), the class checks all of them correctly.
We have divided the specifications into 3 parts.

3.2.1 Part 1

1. Persons are either male or female.

2. Persons can only marry people of the opposite sex.

To guarantee both requirements, these invariants have been added:

```
//@ invariant this.isFemale <==> !this.isMale;
//@ invariant (this.spouse!= null) ==> (this.isMale <==> ↩
    this.spouse.isFemale);
```

These are an obvious consequence of the first ones (some are simple design choices):

```
//@ invariant (this.spouse != null) ==> (this.spouse != ↩
    this.mother && this.spouse != this.father && this.↩
    spouse != this);
//@ invariant (this.spouse != null) ==> this.spouse.↩
    spouse == this;;
//@ invariant mother != this && father != this;
```

The second block of invariants causes problems with the `marry()` method, in fact, to ensure that the method in question verifies all the added invariants, it is also necessary to carry out checks (in the form of preconditions) on the taxpayer passed in input:

```
//@ requires new_spouse != null && !new_spouse.isMarried;
//@requires new_spouse.isFemale <==> this.isMale;
//@requires !this.isMarried;
//@ requires new_spouse != this && new_spouse != this.↩
    mother && new_spouse != this.father;
//@ requires new_spouse.mother != this && new_spouse.↩
    father != this;
//@ requires new_spouse.age >= 18 && age >= 18;
```

With this combination of invariants and requirements openJML continues to consider the class valid. So the specifications of the first part are verified.

```
<package-name>
 Taxpayer
   [VALID] divorce()[0.127 z3_4_3]
   [VALID] haveBirthday()[0.239 z3_4_3]
   [VALID]   marry(Taxpayer)[0.125 z3_4_3]
   [VALID] Taxpayer(boolean,Taxpayer,Taxpayer)[0.251 z3_4_3]
   [VALID] transferAllowance(int)[0.127 z3_4_3]
```

```
Completed proving methods in Taxpayer [3.34 secs]
Summary:
    Valid:        5
    Invalid:      0
    Infeasible:   0
    Timeout:      0
    Error:        0
    Skipped:      0
   TOTAL METHODS: 5
   Classes:       1 proved of 1
   Model Classes: 0
   Model methods: 0 proved of 0
   DURATION:          3.4 secs
[3.98] Completed
```

3.2.2 Part 2 and 3

1. The tax system uses an income tax allowance (deduction): Every person has an income tax allowance on which no tax is paid.
   There is a default tax allowance of 5000 per individual and only income above this amount is taxed. Married persons can pool their tax allowance, as long as the sum of their tax allowances remains the same. For example, the wife could have a tax allowance of 9000, and the husband a tax allowance of 1000.

2. The new government introduces a measure that people aged 65 and over have a higher tax allowance, of 7000. The rules for pooling tax allowances remain the same.

The properties required with parts 2 and 3 were considered together,these are the invariants:

```
//@ invariant (!isMarried && age<65) ==> tax_allowance ==↩
    DEFAULT_ALLOWANCE;
//@ invariant (!isMarried && age >= 65) ==> tax_allowance↩
    == ALLOWANCE_OAP;
/*@ invariant isMarried ==> (
    ((age < 65 && spouse.age < 65) <==> (tax_allowance + ↩
       spouse.tax_allowance == 10000))
   ||
   (((age < 65 && spouse.age >= 65)||(age >= 65 && spouse↩
      .age < 65)) <==> (tax_allowance + spouse.↩
      tax_allowance == 12000))
   ||
   ((age >= 65 && spouse.age >= 65) <==> (tax_allowance +↩
       spouse.tax_allowance == 14000))
```

```
    );

@*/
```

Although these invariants lead to verifying the requirements of part 2 and part 3, the `divorce()` and the `haveBirthday()` methods are no longer considered valid. The `divorce()` method is invalidated because of the first and the second invariant.
In the case where two married taxpayers divorce, the values of their tax allowances must be restored to their default value and according to their age. So if the age is greater or equals then 65, at the time of divorce, the tax allowance must be restored to the value defined in the variable ALLOWANCE_OAP, otherwise to the value defined in the variable DEFAULT_ALLOWANCE.
This is the correct version of the divorce method () according to new invariants:

```
//@ requires isMarried;
 void divorce() {
  if(spouse.age>=65)spouse.tax_allowance = ALLOWANCE_OAP;
  else spouse.tax_allowance = DEFAULT_ALLOWANCE;
  spouse.isMarried=false;
  spouse.spouse = null;
  spouse = null;
  isMarried = false;
  if(age >= 65) tax_allowance = ALLOWANCE_OAP;
  else tax_allowance = DEFAULT_ALLOWANCE;
 }
```

The `haveBirthday()` method is invalidated because of the second invariant. When a taxpayer turns 65, it is necessary to update his tax allowance with the difference between the new one and the previous one. In this way, regardless of whether he is married or not we are sure that the second invariant is verified and even the third remains verified. This is the correct version of the `haveBirthday()` method according to these considerations:

```
//@ requires age+1 <= Integer.MAX_VALUE;
 void haveBirthday() {
 age++;
  if(age==65) tax_allowance += (ALLOWANCE_OAP - ↩
      DEFAULT_ALLOWANCE);
 }
```

This change obviously leads to another *ArithmeticOperationRange* warning,easily solved with a requirement by which we check that doing this update does not result in an *Integer Overflow*.
This is the final correct version of the `haveBirthday()` method according to these new considerations:

```
//@ requires age+1 <= Integer.MAX_VALUE;
//@ requires age+1 == 65 ==> tax_allowance + (↩
   ALLOWANCE_OAP - DEFAULT_ALLOWANCE) <= Integer.↩
   MAX_VALUE;
 void haveBirthday() {
 age++;
  if(age==65) tax_allowance += (ALLOWANCE_OAP - ↩
     DEFAULT_ALLOWANCE);
 }
```

With this combination of invariants, requirements and code changes, openJML continues to consider the class valid. So the specifications of the second and third parts are verified.

```
<package-name>
 Taxpayer
  [VALID] divorce()[0.127 z3_4_3]
  [VALID] haveBirthday()[0.239 z3_4_3]
  [VALID]   marry(Taxpayer)[0.125 z3_4_3]
  [VALID] Taxpayer(boolean,Taxpayer,Taxpayer)[0.251 z3_4_3]
  [VALID] transferAllowance(int)[0.127 z3_4_3]
```

```
Completed proving methods in Taxpayer [6.63 secs]
Summary:
    Valid:         5
    Invalid:       0
    Infeasible:    0
    Timeout:       0
    Error:         0
    Skipped:       0
   TOTAL METHODS: 5
   Classes:       1 proved of 1
   Model Classes: 0
   Model methods: 0 proved of 0
   DURATION:          6.7 secs
[7.23] Completed
```

4 Final version

This section shows the final version of the class. This version is the result of the addition of annotations and code modifications to obtain a correctly written class that verifies all the properties requested by the customer.

```
/*
  This assignment illustrates how specifications (↩
     invariants and
  preconditions) written in a formal language can help ↩
     in removing
  errors in code.
 */

class Taxpayer {


 /* isFemale is true iff the person is female */
 boolean isFemale;

 /* isMale is true iff the person is male */
 boolean isMale;

 int age;

 boolean isMarried;

 /* Reference to spouse if person is married, null ↩
    otherwise */
 /*@ nullable @*/ Taxpayer spouse;

 /* Constant default income tax allowance */
 static final int DEFAULT_ALLOWANCE = 5000;

 /* Constant income tax allowance for Older Taxpayers ↩
    over 65 */
 static final  int ALLOWANCE_OAP = 7000;

 /* Income tax allowance */
 int tax_allowance;
```

```
 int income;
 /*@ nullable @*/ Taxpayer mother;
 /*@ nullable @*/ Taxpayer father;


//@ invariant this.isMarried <==> this.spouse != null;
//@ invariant age >=0 && age <= Integer.MAX_VALUE;
//@ invariant this.tax_allowance >= 0 && tax_allowance <=↩
    Integer.MAX_VALUE;
//@ invariant  this.income >= 0 && this.income <= Integer↩
   .MAX_VALUE;

//part1
//@ invariant this.isFemale <==> !this.isMale;
//@ invariant (this.spouse!= null) ==> (this.isMale <==> ↩
   this.spouse.isFemale);
//@ invariant (this.spouse != null) ==> (this.spouse != ↩
   this.mother && this.spouse != this.father && this.↩
   spouse != this);
//@ invariant (this.spouse != null) ==> this.spouse.↩
   spouse == this;
//@ invariant this.isMarried ==> age >= 18 && spouse.age ↩
   >= 18;
//@ invariant mother != this && father != this;
 //part2
 //part3
//@ invariant (!isMarried && age<65) ==> tax_allowance ==↩
    DEFAULT_ALLOWANCE;
//@ invariant (!isMarried && age >= 65) ==> tax_allowance↩
    == ALLOWANCE_OAP;
/*@ invariant isMarried ==> (
   ((age < 65 && spouse.age < 65) <==> (tax_allowance + ↩
      spouse.tax_allowance == 10000))
  ||
  (((age < 65 && spouse.age >= 65)||(age >= 65 && spouse.↩
     age < 65)) <==> (tax_allowance + spouse.↩
     tax_allowance == 12000))
  ||
  ((age >= 65 && spouse.age >= 65) <==> (tax_allowance + ↩
     spouse.tax_allowance == 14000))
  );

@*/
```

```
 Taxpayer(boolean babyboy, Taxpayer ma, Taxpayer pa) {
   age = 0;
   isMarried = false;
   this.isMale = babyboy;
   this.isFemale = !babyboy;
   mother = ma;
   father = pa;
   spouse = null;
   income = 0;
   tax_allowance = DEFAULT_ALLOWANCE;

 }


 //@ requires new_spouse != null;
//@ requires new_spouse != null && !new_spouse.isMarried;
//@requires new_spouse.isFemale <==> this.isMale;
//@requires !this.isMarried;
//@ requires new_spouse != this && new_spouse != this.↩
   mother && new_spouse != this.father;
//@ requires new_spouse.mother != this && new_spouse.↩
   father != this;
//@ requires new_spouse.age >= 18 && age >= 18;
void marry(Taxpayer new_spouse) {

  spouse = new_spouse;
  spouse.spouse = this;
  spouse.isMarried = true;
  isMarried = true;
}



//@ requires isMarried;
 void divorce() {
  if(spouse.age>=65) spouse.tax_allowance = ALLOWANCE_OAP↩
     ;
  else spouse.tax_allowance = DEFAULT_ALLOWANCE;
  spouse.isMarried=false;     //+ static check
  spouse.spouse = null;
```

```
  spouse = null;
  isMarried = false;
  if(age >= 65) tax_allowance = ALLOWANCE_OAP;
  else tax_allowance = DEFAULT_ALLOWANCE;
 }

 /* Transfer part of tax allowance to own spouse */
 /*@
 requires change >= 0 && change <= tax_allowance;
 requires isMarried;
 requires change + spouse.tax_allowance <= Integer.↩
    MAX_VALUE;
@*/
 void transferAllowance(int change) {
  tax_allowance = tax_allowance - change;
  spouse.tax_allowance = spouse.tax_allowance + change;
 }


//@ requires age+1 <= Integer.MAX_VALUE;
//@ requires age+1 == 65 ==> tax_allowance + (↩
   ALLOWANCE_OAP - DEFAULT_ALLOWANCE) <= Integer.↩
   MAX_VALUE;
 void haveBirthday() {
 age++;
  if(age==65) tax_allowance += (ALLOWANCE_OAP - ↩
     DEFAULT_ALLOWANCE);
 }

}
```

5 Further possible improvements

Although the class now checks all the properties requested by the customer and passes all the tests performed by the static analysis, there would be other improvements that can be made.
Methods like `haveBirthday()`,`transferAllowance(int change)` etc. produce `side effect` on the data: an operation, function or expression is said to have a `side effect` if it modifies some state variable value(s) outside its local environment [9].
So in cases where there are methods that access and modify resources, it is

important to protect those resources from simultaneous access of multiple threads.
Usually this is done using the `synchronized` keyword, which can be used directly in the method header or as an instruction block.
Making the most of this keyword as best as possible, it is possible to regulate access to those methods that produce side effects on the data, in order to avoid *undefined results*.
Furthermore,those methods that perform transactions, such as `transferAllowance (int change)` could be modified in `final` methods, so that by extending the class, it is not possible to override the method in any way.

6 Conclusion

The objective of this report was to shown how to use openJML and the JML language to verify the correctness of a a Java class with respect to some specifications.
First the program was modified following the warnings provided by openJML result of the static analysis.
Then, in accordance with the customer's specifications, all the necessary changes have been added to ensure that this class can correctly verify them.

References

[1] Roger B. Dannemberg,George W. Ernst, Formal Program Verification Using Symbolic Execution,
`https://www.cs.cmu.edu/~rbd/papers/verification1982dannenberg.pdf` tml

[2] OpenJML installation Execution,
`https://www.openjml.org/documentation/installation.sh`

[3] JML Reference Manual, 1. Introduction
`http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_1.html`

[4] OpenJML Checks
`https://www.openjml.org/documentation/checks.shtml`

[5] JML Reference Manual 2.8 Null is Not the Default
`http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_2.html`

[6] Null Dereference
https://www.owasp.org/index.php/Null_Dereference

[7] Introduction to JML,Erik Poll, Radboud University Nijmegen
https://www.cs.ru.nl/E.Poll/talks/jml_basic.pdf

[8] Integer Overflows, The Web Application Security Consortium
http://projects.webappsec.org/w/page/13246946/IntegerOverflows

[9] Side effect (computer science), Wikipedia
https://en.wikipedia.org/wiki/Side_effect_(computer_science)