Duck Testing Enhancements for Automated Validation of Student Programmes

How to Automatically Test the Quality of Implementation of Students' Programmes

Pavel Herout and Premysl Brada
Department of Computer Science and Engineering, University of West Bohemia, Plzen, Czech Republic

Keywords: OOP, Testing, Duck Typing, Evaluation, Java, JUnit.

Abstract: This article deals with the issue how to test the quality of novice programmers' software assignments. This problem is becoming serious due to the hundreds of students in the introductory courses of programming. The article discusses the motivation for using quality of implementation tests of students' programmes, their principles and a practical solution. So called "duck tests" are used for this type of validation. A combination of a framework Duckapter, JUnit library and own programmes constitutes the practical solution. It is represented by a self-contained tool which is freely at disposal. The described tool has been used for three years in the elementary course of object oriented programming based on the Java programming language, alongside three other tools used for automatic validation of students projects. The article discusses the experience gained from its use and the effects on student's programming skills.

1 INTRODUCTION

Basic courses of programming have several goals. Certainly the main one is to lay the foundations of programming. The next aim is considered to be teaching students the good habits/practices, recommended techniques and other so-called "extra-functional properties". Those will be used by students later on in all programming subjects during their studies, and also in their professional life. For the real learning of skills mentioned above it is necessary to use them actively in students' individual projects.

To make sure that they really have learned them and moreover that they have done it properly, the best way is to inspect the code thoroughly. That means situation when a teacher looks into student's source code and discusses the mistakes the student made in both the algorithmic aspects and in language constructs.

Unfortunately, this ideal status "one-to-one" (Bloom, 1984, Lane, 2003) is mostly hard to achieve due to the high number of students (hundreds) in the programming courses. In reality, the number of individual student projects is even larger while it is necessary to evaluate all of these projects to a reasonable degree.

One widely adopted solution of this problem is to automatically check the projects on some validation server (eg. Web-CAT, 2014). Although not addressing all needs in teaching programming skills, the appeal of automated validation lies in the possibility to provide basic feedback to large numbers of students in a time-effective manner.

1.1 Testing Techniques and Possibilities

Two main needs related to this validation are testing the student programmes for correct functionality and checking the qualitative aspects of their internal implementation.

There are several ways how to check the functionality, but the most appropriate one seems to be the usage of Unit tests. Unit tests have several advantages. The most important one nowadays is that they are becoming integral parts of the programmer's work. ("Immediately check the functionality of your piece of code.") That means, employing Unit tests as part of course evaluation prepares students for this style of programming, e.g. in test-driven-development. The second advantage of the Unit test is their fine granularity. Rather than exercising the functionality of a program as a whole, it is possible and convenient to explore the functionality of standalone methods. This approach

Herout P. and Brada P..
Duck Testing Enhancements for Automated Validation of Student Programmes - How to Automatically Test the Quality of Implementation of Students' Programmes.
DOI: 10.5220/0005412902280234
In *Proceedings of the 7th International Conference on Computer Supported Education* (CSEDU-2015), pages 228-234
ISBN: 978-989-758-107-6

significantly helps properly structure the source code.

A disadvantage of the Unit tests is that they can validate the functionality, moreover only the functionality of accessible methods. When we are concerned about the proper usage of attributes or private methods (common elements of well designed code), Unit tests do not help.

To assess the internal quality of implementation, a commonly used means are tools for static checking of the source code. The most known ones are PMD (PMD, 2014), CheckStyle or FindBugs. In the case of checking of beginners' programs we usually use only very limited set of their possibilities (rules), because there we do not expect sophisticated bugs to appear in these simple programmes and the detailed checks would overwhelm the students. Exaggerating slightly, we use these tools mainly for checking how "nicely written" the explored code is, i.e. its comprehensibility for humans—eg. an inspection of block parenthesis, an appropriate number of lines of methods, suitable number of formal parameters, not too big cyclomatic complexity etc.

A good programmer's documentation is a necessary part of a well written source code. Not many solutions are available in this respect; one such tool we developed (JavadocCheck, 2014) is able to check an occurrence (but not meaningfulness) of all the elements of the source code which can be documented.

1.2 Tests of Quality of Implementation and Meeting Assignment

Unfortunately, it is quite uncommon to automatically test how student met the assignment and what is the quality of implementation. Both these needs are very important in the introductory programming courses.

Simple tasks in these courses are very often based on learning basic skills which students sometimes struggle with. A typical negative example is not using any formal parameters of method or local variables, when students misuse "global" attributes instead. "Magic numbers" in the source code instead of using symbolic constants are another example. Very confusing are insignificant names of variables or methods (namely private ones).

Functional (Unit) tests are inherently not capable to reveal all these flaws, and we cannot use them for tests of documentation either. The tools for static checking of the source code are successful in this case, but only partly.

1.3 Goal of This Paper

The main goal of this article is to explore a solution which would reconcile these two needs—to test that functionality with respect to a given assignment is correct and to check that the implementation is well written, in an automated way. In the following sections we first discuss a motivation example and then describe a technique which has proven to be useful in our work. It is based on a lesser known approach of duck testing.

2 MOTIVATION EXAMPLE

Let's have a typical assignment in beginner's courses of programming: Prepare the class `Person` which calculates a person's Body Mass Index (BMI). The "non-functional requirements" are:

- The class is able to create immutable object only.
- Attributes of this class are *name*, real *weight* (in kg) and integer *height* (in cm).
- The class has two constructors. A constructor without parameters creates the person with name Person, weight 65 (kg) and height 175 (cm). Use symbolic constants for setting these values.
- The constructor with three formal parameters sets all the three attributes.
- Both constructors calculate an integer value of BMI and store it into an attribute.
- BMI will be calculated by private method `calculateBMI()`.
- All attributes will have getters only, no setter (because of the immutable objects).
- The class has an overriden method `toString()`, which returns eg. string `"Person [w:65.0, h:175, BMI:21]"`.

2.1 Teacher's Solution

The teacher's idea of source code meeting the assignment is as follows (the name of the class is changed to `PersonByTeacher` for the sake of definiteness):

```
/** Class Person with atributes
 *       name, weight and height
 * BMI is calculated
 * immutable object - getters only
 */
public class PersonByTeacher {
  private static final String
                   DEF_NAME = "Person";
  private static final double
                   DEF_WEIGHT = 65.0;
  private static final int
                DEF_HEIGHT = 175;

  private String name;
  private double weight;
  private int height;

  private final int BMI;

  public PersonByTeacher() {
    this(DEF_NAME, DEF_WEIGHT,
         DEF_HEIGHT);
  }

  public PersonByTeacher(String name,
           double weight, int height) {
    this.name = name;
    this.weight = weight;
    this.height = height;
    BMI = calculateBMI();
  }

  public String getName()
    { return name; }
  public double getWeight()
    { return weight; }
  public int getHeight()
    { return height; }

  public int getBMI()
    { return BMI; }

  private int calculateBMI() {
    double heightInMe = height / 100.0;
    double bmi = weight /
             (heightInMe * heightInMe);
    return (int) Math.round(bmi);
  }

  @Override
  public String toString() {
    return name + " [" + "w:" + weight
       + ", h:" + height
       + ", BMI:" + getBMI() + "]";
  }
}
```

2.2 Student's Solution

Unfortunately, a student's solution handed in would typically be like the following one (the name of the class is changed to `PersonByStudent`):

```
public class PersonByStudent {
  static String sss = "Person";

  String s;
  double d;
  int i;
  double d1, d2;

  PersonByStudent() {
    s = sss;
    d = 65.0;
    i = 175;
  }

  PersonByStudent(String parS,
           double parD, int parI) {
    s = parS;
    d = parD;
    i = parI;
  }

  String getName() { return s; }
  double getWeight() { return d; }
  int getHeight() { return i; }

  byte getBMI() {
    d1 = i / 100.0;
    d2 = d / (d1 * d1);
    return (byte) Math.round(d2);
  }

  @Override
  public String toString() {
    return s + " [" + "w:" + d + ", h:"
       + i + ", BMI:" + getBMI() + "]";
  }
}
```

2.3 A Confrontation of Both Solutions

Both solutions would pass all functional tests since they give functionally the same results. However, the student's solution does not meet assignment fully (and the quality of its implementation is poor) because:

- it does not use the symbolic constants,
- it does not use the access modifiers (`public`, `private`),
- the naming of attributes is very unclear for a reader (but probably clear for the student, who chose `d` for `double` instead of `weight`, etc.),
- it uses "global" variables instead of local ones (`d1`, `d2`),
- the method `getBMI()` returns inappropriate (not expected) integer type (`byte`),
- the private method for calculating of BMI is not used at all (private methods cannot be tested externally).

Static code checkers would identify some of these problems but they cannot compare the solution to the one expected by the teacher.

The following section shows how this problem can be overcome by using duck tests to pinpoint all of these flaws.

3 DUCK TESTS AND THEIR USE

In dynamically typed languages, an object's suitability for some purpose is determined by the presence of certain methods and properties (with appropriate meaning), rather than the actual type of the object. This concept has been named *duck typing*, after a test attributed to James Whitcomb Riley, which may be phrased as follows:

> "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

In duck typing, a programmer is only concerned about ensuring that objects behave as demanded in a given context, rather than ensuring that they are of a specific type. Instead of specifying types formally, duck typing practices rely on documentation, clear code, and testing to ensure correct use (Duck typing, 2014).

The duck tests for Java programming language as described below are based on the Duckapter framework (Duckapter, 2012). It was created as a master's project and its detailed description can be found at (Orany, 2010). Basic principles of its functionality are duck typing, Java annotations and the adaptor design pattern.

3.1 Duckapter Test Specifications

In Duckapter, the checked properties of a class are defined by interfaces—every such attribute, method and constructor must be described in a standalone interface. Because it is not possible to describe requested attributes, constructors or static methods in Java interfaces, annotations describe these aspects. The framework is able to process these annotations when running the tests.

Attributes are denoted by the `@Field` annotation which determines the particular attribute. In the next step, following annotations are used:

`@Declared`—attribute is declared in the checked class, not inherited

`@Private` `@Public` `@Protected`—access modifiers

`@StaticField` `@NonStatic`—static or instance attribute

`@Final`—constant

The name of the attribute must start with the prefix `get` and finishes with the parenthesis, like a method (`getDEF_NAME()`).

Description of checked methods must be again described in the interface and it is very similar to a definition of method. Following annotations are used:

`@Declared`—method is declared in the checked class, not inherited

`@Private` `@Public` `@Protected`—access modifiers

`@Static` `@NonStatic`—static or instance method

The annotation `@Constructor` and the name `newInstance()` with appropriate numbers and types of formal parameters is used for description of constructors.

Some examples from the `Person` class above in their standalone interfaces are:

```
interface ITestDuckPerson_DEF_NAME {
  @Field @Declared @Private @Final
     @StaticField String getDEF_NAME();
}
interface ITestDuckPerson_name {
  @Field @Declared @Private
           @NonStatic String getname();
}

interface ITestDuckPerson_toString {
  @Declared @Public @NonStatic
                   String toString();
}
interface ITestDuckPerson_calculateBMI() {
  @Declared @Private @NonStatic
                  int calculateBMI();
}

interface ITestDuckPerson_Person2 {
  @Constructor Person
      newInstance(String name,
            double weight, int height);
}
```

3.2 Actual Testing

During the test the framework Duckapter verifies if the checked class includes all elements defined in the interfaces. A method `wrap()` is used for it. Its real parameters are the checked class and a specific interface. If all checked parts of class's element correspond with their description in the interface, the actually checked element is valid. If not, an exception `WrappingException` is thrown.

A short example of testing source code for the attribute `DEF_NAME` described in the interface `ITestDuckPerson_DEF_NAME` (see above):

```
public class DuckTestPerson {
...
  try {
    Duck.wrap(new Person(),
       ITestDuckPerson_DEF_NAME.class);
  }
  catch (WrappingException e) {
    System.out.println(e.getAdapted()
                     .getClassWrapper()
       .getUnimplementedForInstance());
    System.out.println("Wrong declared"
          + " attribute DEF_NAME - " +
      "must be private static final ");
  }
...
}
```

If we return back to the motivation example above, in the teacher's solution we can see the parts with a grey background—they provide the information what needs to be checked in the solution. If the corresponding duck tests are a part of an automatic validation tests, a student's source code like the one shown will be rejected as incorrect. Moreover, detailed error messages will advise the student about the problems found.

3.3 Limitation of Duck Tests

The basic idea of duck test is well suitable for our purpose; also the implementation in the Duckapter framework is fit to use.

However, a major weakness for regular use is that it is a necessity to have a template solution (usually the teacher's one) in advance. Moreover, the students have to keep this solution. These requirements imply several drawbacks:

- Duck test are suitable only for rather simple (trivial) tasks.
- Student's own creativity is almost impossible.
- Teacher's solution must not include any errors or imperfections.

Despite these limitations it is possible to find situations (see Section 5) where these are not a drawback but an advantage. This comprises the evaluation of programs in the introductory programming courses where students learn the basic skills and the level of their creativity should be minimized.

During practical use of duck tests in our first year courses, we also revealed several practical issues:

- Handmade (eg. using text editor) preparation of all of interfaces with the descriptions of attributes, methods and constructors is long term monotonous work requiring a high level of perfection. Several errors or omissions occur every time and it is very difficult to detect them. The complexity (eg. number of testing interfaces) obviously increases with the number of checked classes. The situation becomes very difficult to manage if we have to use approximately 100 interfaces (Section 5).
- Error output to the console is not suitable.
- The same set of tests is used in two contexts—for student self-evaluation and when grading the projects they hand in. Therefore, students need to have the same local testing programme as are the ones located on the validation server, used for the grading evaluation.
- Running duck tests from the command line is an almost insuperable obstacle for many beginners.

4 CREATED MODIFICATION OF DUCK TESTS AND RESULTING SYSTEM

On the basis of the experiences mentioned above we decided to keep using the framework Duckapter but extend its possibilities in the following ways:

- An evaluation of tests (pass/fail) will be provided by JUnit library, which is well known and widely used.
- A new duck test generator will be prepared to simplify duck tests preparation. It should have a GUI and work automatically, eg. generate the source code of all interfaces and all tests.
- A new duck test launcher will be created. It must have a GUI to let duck test run as easily as possible. This way the students can run tests repeatedly with clear results and error messages. The "directive feedback" (Shute, 2008) is to be used.

Implementation of these ideas resulted in a new system consisting of a test generator and test launcher (`www.kiv.zcu.cz/~herout/data/duck-test-system.zip`) which has been used for three consecutive years now. The test generator (`duckTestGenerator.jar`) automatically creates two source codes for each of the checked classes. The `ITestDuckXYZ.java` contains all interfaces (eg. `ITestDuckPerson.java`) and the file `TestDuckXYZ.java` the source codes of all tests (eg. `TestDuckPerson.java`). The generator subsequently compiles both files and packs them, together with the Duckapter framework and JUnit library, into one JAR file (`duckTestPer-son.jar`). This file is at disposal to the students.

The generator works fully automatically, the only activity which the teacher should do is to select a list of checked classes. The test launcher allows choosing a given JAR file (eg. `duckTestPerson.jar`) and a directory with student's source codes. The launcher runs all the prepared duck tests and generates a detailed and transparent error message in case any test fails. The student is able to correct flaws in his/her source code immediately and validate this correction by another run of the tests.

5 PRACTICAL EXPERIENCE

The system for generating and running duck tests described above has been a routine operation for three years of teaching the course Object Oriented Programming, run in the first year of a bachelor curriculum.

Note: In this course we use Level 1 of learner engagement, where "an advancement employs simple scenarios or interactive examples that demonstrate or require the learner to work through a problem that is tied to a learning objective." (Greitzer, 2007)

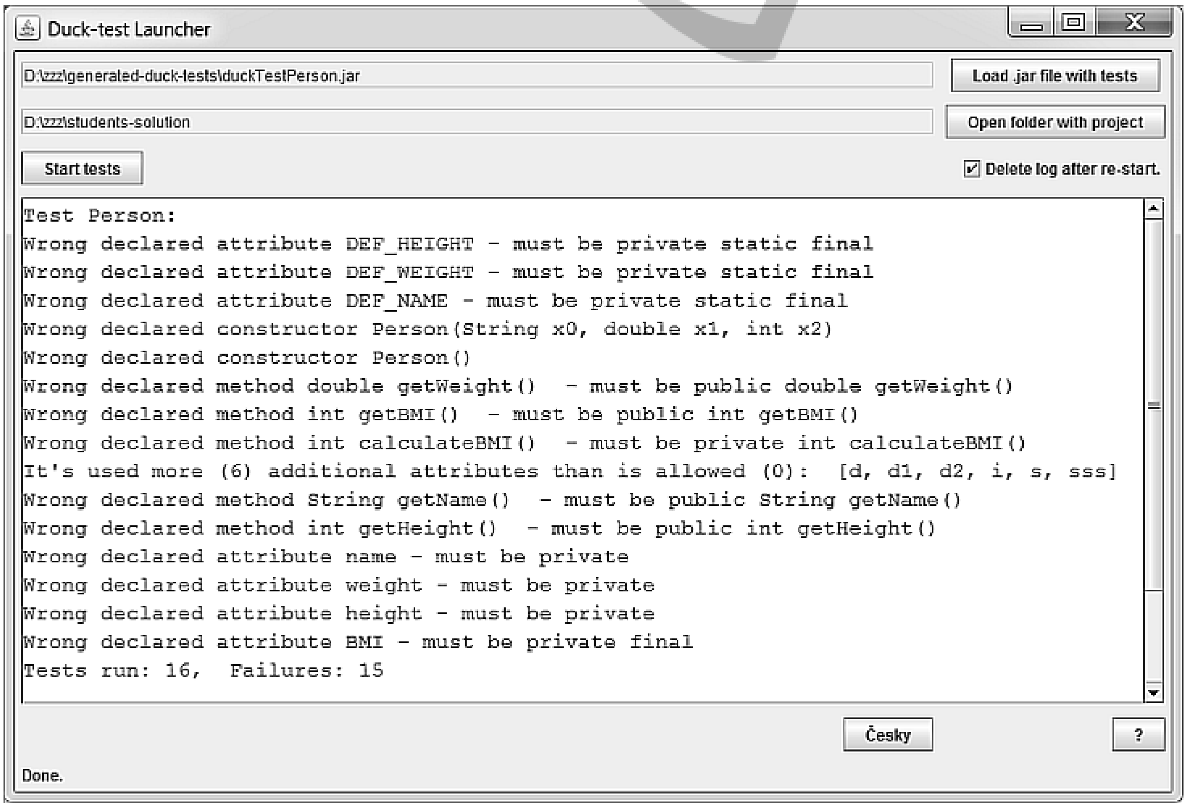
Approximately 100 students enrol for this course each year. Each of them has to prepare a complex student's project consisting of eight parts which gradually follow each other.

The final version (task) of the project consists of 8 classes, one enum and two interfaces. Together they have 22 constructors and 58 methods.

All classes are checked by duck test (and of course parallel for their functionality and completeness of Javadoc documentation). There are 103 duck tests totally in the final version. All classes of the whole project are checked by 326 duck test, but some of them repeat, of course.

Students have all mentioned tests at their disposal and run them by the launcher mentioned above. That means that the launcher can be considered as verified. On the other hand the test generator is used by the author of this article, so it should be considered as a prototype.

At the beginning of the project students consider Duck-test as useless and annoying (similarly to static checking of the source code and checking of completeness of Javadoc comments). But during the course they find out their usefulness. They concede this fact in personal meetings during seminars and in course final quality assessment too.



D:\zzz\generated-duck-tests\duckTestPerson.jar

D:\zzz\students-solution

☑ Delete log after re-start.

```
Test Person:
Wrong declared attribute DEF_HEIGHT - must be private static final
Wrong declared attribute DEF_WEIGHT - must be private static final
Wrong declared attribute DEF_NAME - must be private static final
Wrong declared constructor Person(String x0, double x1, int x2)
Wrong declared constructor Person()
Wrong declared method double getWeight()  - must be public double getWeight()
Wrong declared method int getBMI()  - must be public int getBMI()
Wrong declared method int calculateBMI()  - must be private int calculateBMI()
It's used more (6) additional attributes than is allowed (0):  [d, d1, d2, i, s, sss]
Wrong declared method String getName()  - must be public String getName()
Wrong declared method int getHeight()  - must be public int getHeight()
Wrong declared attribute name - must be private
Wrong declared attribute weight - must be private
Wrong declared attribute height - must be private
Wrong declared attribute BMI - must be private final
Tests run: 16,  Failures: 15
```

Done.

Figure 1: Duck test launcher.

They realize that without properly fulfilling non-functional requirements they cannot understand a progressive complexity of the project (the first task has only one class, but the final task 11 classes/enums/interfaces).

Understanding that their piece of source code is not isolated and the only one used, but forms a part of bigger system, is very important for them.

On the other hand as teachers we know that fully automatic validation does not solve all problems. We therefore check the source code of the final task manually too.

Because the students have used the Duck launcher locally up to now, we do not have data about their mistakes and improvement at disposal. This is planned to change in the future.

6 CONCLUSIONS

This article describes motivation for using tests of implementation quality for students' programmes, their principles and a practical solution based on the duck testing approach. The practical solution is represented by a system of tools which allow highly automated use.

The examples of practical experience gained during three years of usage of the system allow us to state that implementation quality tests prove their usefulness as a complement to other validation test.

The proposed tool need not be used directly by students or for automatic validation only. It can be used by the teachers only to quickly pinpoint flaws in the student programme, so the teacher can concentrate e.g. on students with higher number of flaws.

For future enhancements of the approach, we are preparing data collection from all running Duck-tests. The goal is to make it possible to analyse:

- continuous improvement of students during the course,
- typical / most common mistakes,
- correlation between total error rate and final mark,
- percentage of students who make mistakes repeatedly, etc.

Further, the used "directive feedback" can be exchanged to the "error flagging" one (Shute, 2008) for some advanced tasks.

ACKNOWLEDGEMENTS

This work was supported by Ministry of Education, Youth, and Sport of the Czech Republic—University specific research—1311.

REFERENCES

Bloom, B. S., 1984. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher 13(6), pp. 4-16.*

Lane, H. C., 2003. Preventive Tutoring in Programming: A Tutoring System for Novice Program Design., *University of Pittsburgh.*

Web-CAT, 2014. Resources for automated grading and testing, http://web-cat.org.

PMD, 2014. The scanner of Java source code, http://pmd.sourceforge.net.

JavadocCheck, 2014. The parser and the checker of Java documentation, https://github.com/scais/jdparser.

Duckapter, 2012. Java reflection library introducing duck typing into the Java programming language http://github.com/musketyr/duckapter.

Orany, V., 2010. Automatic validation of students' projects. *Master Thesis, University of Economics, Prague.*

Duck typing, 2014. http://en.wikipedia.org/wiki/Duck_typing.

Shute, V. J., 2008. Focus on formative feedback. *Review of educational research 78(1), pp. 153-189.*

Greitzer, F. L., Kuchar, O. A. and Huston, K,, 2007. Cognitive science implications for enhancing training effectiveness in a serious gaming context. *Educ. Resour. Comput., 7(3):2.*