JExercise – a specification-based and test-driven exercise support plugin for Eclipse

Hallvard Trætteberg
Associate Professor
Dept. of Computer and information sciences
Norwegian University of Science and Technology
7491 Trondheim, Norway
+47 918 97263
hal@idi.ntnu.no

Trond Aalberg
Associate Professor
Dept. of Computer and information sciences
Norwegian University of Science and Technology
7491 Trondheim, Norway
+47 976 31088
trond.aalberg@idi.ntnu.no

ABSTRACT

Programming exercises are an important part of an introductory course in programming. To improve the focus on encapsulation, requirements-based testing and give better feedback given to the students during their work, we have created an Eclipse-based plugin called JExercise. Based on a model of an exercise, it presents the structure of requirements to the student and allows her to test the code by running accompanying JUnit tests.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.1.5 [**Programming techniques**] Object-oriented Programming; D.2.5. [**Software Engineering**]: Testing and Debugging—testing tools; D.3.2 [**Programming Languages**]: Language Classifications—Java.

General Terms

Languages, Verification.

Keywords

Test-driven development, JUnit framework, Eclipse IDE.

1. INTRODUCTION

Programming is a practical skill that requires both conceptual and practical training. It's not just a matter of learning the programming language semantics, but also learning to use it in a sensible way, by utilizing sound object-oriented methods and modern development tools. Hence, a programming course's exercises should be based on sound object-oriented methods and drive the student through the relevant practical experiences.

For our introductory Java course we have developed a set of specification-based and test-driven exercises and an Eclipse plugin named JExercise. In the following sections we explain our approach to exercise design, the role and design of the JExercise tool, JExercise's implementation and our experiences after our first semester using our approach and tool.

2. OUR INTRODUCTORY COURSE

Our introductory Java course is taken by 500+ students from many engineering faculties, including our own ICT students, and has three main learning objectives: 1) Java language semantics, 2) object-oriented principles and methods (including JUnit testing) and 3) practical programming skills using modern development tools. A requirement for taking the course is a basic understanding of procedural programming (scripting). In practice, the interest, understanding and skills vary considerably; some don't know what a variable is, while some have programmed for many years already.

Several observations led us to developing JExercise-based exercises: 1) Students follow the 20/80 rule: 20% of the effort gives 80% of the solution. However, 80% correct code have little (20%?) value, so it's important to force them to write 100% correct code. 2) Grading exercises requires a lot of effort, which should rather be used for supporting the learning process. 3) Weaker students need continuous feedback that they're on the right track, while stronger ones prefer freedom.

The "solution" is a set of exercises based on precise and testable requirements and a support system that lets the student continuously test their code's correctness and their overall progress. In addition to being a remedy for the problems noted above, we hope this implicit way of teaching test methods will be a pedagogical gain, as testing methods have increased focus in several of our courses.

3. SPECIFICATION-BASED AND TEST-DRIVEN EXERCISES

A specification for an exercise may be given at many levels and in many ways. Although we ideally would like students to practice all relevant language constructs and ways of using them, a too detailed specification will give little room for important exploration and experimentation. A too vague and high-level specification is however difficult to test and leaves little guarantee that sound and relevant skills and practices are acquired.

Based on the learning objectives, we use object encapsulation, i.e. an object's externally visible behavior, as the guiding principle for our specification. This is sound both from a methodological and pedagogical viewpoint: Many design practices, patterns and modern testing methods are based on object encapsulation, and the freedom given to the student may be varied by defining an encapsulation at the appropriate level of detail.

3.1 Specifying the behavior of an object

The externally visible (and testable) behavior of an object includes many elements:

- the value of public fields and the return value of public methods over time
- side-effects, i.e. changes to any accessible data, the console and file system or other elements of the class' environment
- exceptions, i.e. how certain conditions aborts the "normal" flow of control
- method trace, i.e. that calling a method on one object should result in a method call on an other object

When specifying the behavior of a class or method, more or less of these elements may be included, depending on the topics covered so far in the course. The canonical first exercise may simply require that there is a static void main(String[]) method that prints "Hello world" to Standard.out. An exercise focusing on validation of arguments would introduce exceptions and specify when they are thrown, while an exercise on the observer-observed pattern would specify the required sequences of method calls, that Observed.addObserver(Observer) followed by Observed.change() should result in Observer.notify(Observed)).

To be able to test such behavior, the structure of named elements (packages, classes and methods) and their visibility must be well-defined and specified. This may seem like a drawback, as it makes the specification fairly detailed. However, many standard practices prescribe both the specific set of classes and methods (i.e. their names and members and their parameters) that are part of the encapsulating interfaces. E.g. getter/setter pairs are standard for 1-1 associations, while several variant sets of methods are used for 1-n associations. Similarly, although design patterns are generic and must by definition be adapted to the specific context of the application, they are fairly well-defined in the context of a specific exercise, so the syntactic elements and behavior may often be specified in detail.

3.2 Testing the specified behavior

Our approach is based on specifying the behavior of objects, or rather, classes and their methods. However, we also require that the specification is testable by means of unit tests using the JUnit framework. This style of testing, where individual methods, groups of methods, classes and small groups of classes are tested for functional behavior, suits our needs well, since we want to give the students feedback during the work on the exercise, rather than when it is completed. The JUnit framework [5] supports running both whole test classes and individual test methods in such classes, each of which may test one or more exercise methods. This gives great flexibility for the exercise author.

In the simplest case, there is a one-to-one correspondence between exercise methods and test methods. I.e., once an exercise method is written and a single test method may be run to give the student feedback about the correctness. Fairly often methods are so interrelated that they cannot be tested separately, so one test method may cover several exercise methods. For complex exercise methods, it may also be possible and desirable to have several test methods, one for each specific requirement. E.g. one method may test the standard behavior, while another may test boundary cases, like null arguments and empty collections. This makes it possible to give more precise feedback and guidance, since it is easier to pinpoint the fault. Of course, for more advanced courses and students, the other extreme case may be desirable, that of testing a whole application with a single test method.

Although few specifications can be tested completely, e.g. how do you test that a method correctly sums any two integers, it's fairly easy to test students' code for correctness, since there are limited ways of incorrectly implementing typical exercises. Besides tests for the return values of method, we use tests for output to System.out (using regular expressions), tests for correct use of exceptions (whether they are thrown or not), processing of files and JavaBean-compliant event notifications (that the listeners' method is called with the correct values). In a different course, we've used a JUnit add-on called JFCUnit for testing Swing GUI's.

4. USING JEXERCISE[1]

4.1 Installing and preparing JExercise

The JExercise system is packaged as one feature which may be installed by pointing to the JExercise update site.[2] The package consists of two plugins, one for the underlying model and one for Eclipse view, and depends on the standard Java Development Tools (JDT) and EMF 2.2.0.

Using JExercise for the first time requires three steps. First, the Java project must be set up, with a standard folder structure and build path (including junit.jar), and the JExercise preferences filled in accordingly. Second, one or more exercises must be imported into the project (more may be imported later). Third, the JExercise view must be opened and a specific exercise selected.

Currently, the import step utilizes Eclipse's built in wizard for zip-files. The zip-file contains XML and HTML files describing the exercises and test files for testing the student's code and may additionally contain java source files and resources that the student may start from. By standardizing the folder structure, the files may be imported into the Java project's top level, and the files be spread across sub-folders. We have chosen to have one folder for each kind of content mentioned above, named ex, tests and src, but other structures will work as long as the Java project's configuration and zip-files agree.

4.2 The JExercise view

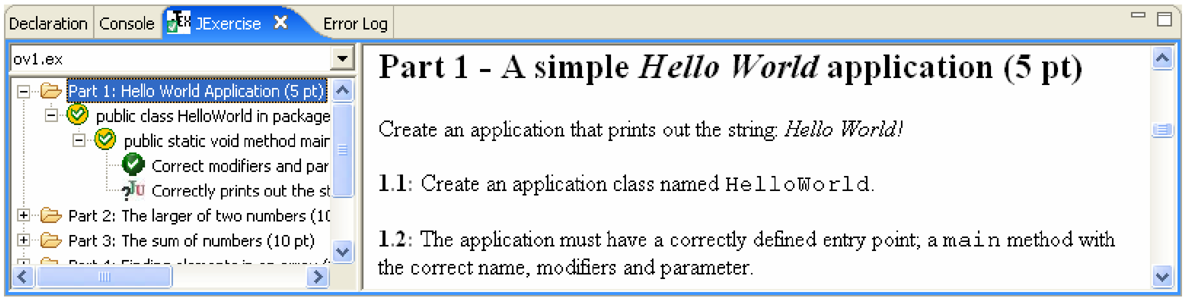
JExercise is designed to integrate into Eclipse, as a view below the main editor pane. The view contains three main elements, as shown in Figure 1. In the top left drop-down a specific exercise (.ex file) may be selected. The requirements structure of this exercise will then be shown in the tree at the left. I.e. each line in the tree corresponds to one a requirement, either for a syntactic element or a testable, functional requirement. At the right a standard web browser window shows the exercise text. The browser is linked to the tree, so when a requirement is selected in

[1] For details and demos, see JExercise's Home Page at http://www.idi.ntnu.no/~hal/development/jexercise/

[2] http://www.idi.ntnu.no/~hal/development/site/

the tree, the browser navigates to the corresponding text in the browser (if the underlying XML and HTML files are correctly linked, that is).

An exercise is typically structured as a set of parts containing requirements for specific syntactic elements, some of which may be tested with a JUnit test. The exercise shown in Figure 1, has several parts, the first of which is the canonical Hello World application. This part requires a HelloWorld class with a main method with correct signature and modifiers. A test for the main method is provided, but has not yet been run. The text for each requirement may be generated from the model or explicitly authored.



Part 1: Hello World Application (5 pt)
public class HelloWorld in package
public static void method mair
Correct modifiers and par
Correctly prints out the st
Part 2: The larger of two numbers (1(
Part 3: The sum of numbers (10 pt)

Part 1 - A simple *Hello World* application (5 pt)

Create an application that prints out the string: *Hello World!*

1.1: Create an application class named `HelloWorld`.

1.2: The application must have a correctly defined entry point; a `main` method with the correct name, modifiers and parameter.

Figure 1. The JExercise view

The requirement's icon indicates whether or not the requirement is met. Requirements for syntactic elements are continuously checked (when a Eclipse's Java model update notification arrives) and the icons updated correspondingly. Testing functional requirements using JUnit tests is more costly and must be triggered manually. When the test run is finished, the success or failure is indicted by the icon. The different icons and their meanings are shown in Table 1.

Table 1. The icons used by the JExercise view

Icon	Meaning
?	The requirement's state is currently *undecided*, presumable because a pre-condition is unsatisfied.
✓	The requirement is completely *satisfied*, as indicated by the green color.
✓	The requirement itself is *satisfied*, but there are *undecided* sub-requirements
✓	The requirement itself is *satisfied*, but come sub-requirements are *violated*, as indicated by the red color.
✗	The requirement is *violated*
?JU	The requirement is a JUnit test that has not been run.
✓JU	The requirement is a JUnit test that has *succeeded*
✗JU	The requirement is a JUnit test that has *failed*

As can be seen, the icons encode three kinds of states, undecided (yellow), success (green) and fail (red), indicate the state of the requirement (symbol) and the sub-requirements (background) and if it is testable by means of a JUnit test (JU symbol).

In Figure 1, we see that the HelloWorld class exits and the main(String[]) method is present with correct modifiers, as indicated by the check marks. However, since the test has not been run, the background is yellow, indicating undecided sub-requirements. Once the test run completes, the icons will be green or red, depending on the result.

Note that during the test run, JUnit runner pane in Eclipse will be activated as usual. Thus, by carefully designing the tests, and in particular the message argument to the assertXXX methods, the student gets additional guidance in the debugging process.

5. THE JEXERCISE MODEL

The JExercise system is based on a logical exercise model conceptually split in two. The *solution* model describes the structure of syntactic Java elements, i.e. packages, classes, method, fields and their signatures and modifiers that the complete exercise requires. The *requirements* model describes a hierarchy of requirements, both syntactic requirements, with references to the solution model, and functional requirements and their corresponding JUnit tests. It is the requirements model that is shown in the JExercise view, the solution model is hidden.

The most important elements of the models are shown in Figure 2 as a UML class diagram. The left part with the JavaElement class as its root is the solution model, while the right part with Requirement class as its root is the requirements model. The link between these is the association between a JavaRequirement and the corresponding JavaElement. As can be seen, both models are hierarchical, the solution model has three levels, the JavaPack, JavaClass and Member levels, while the Requirement hierarchy may be many-leveled. In a specific exercise model, the solution hierarchy of JavaElements and JavaRequirements will be similar.

The reason for this logical split, is to make the granularity of the requirements structure independent of the solution. E.g. it is possible to have several requirements for one java method, or a single (and large) requirement for a whole class. Hence, the exercise author may tailor the level of feedback and guidance to the course and students for the same programming problem.

6. THE JEXERCISE IMPLEMENTATION[3]

JExercise is implemented as two Eclipse plugins, the model and the client view. The model is implemented using the Eclipse

[3] The JExercise code is open source and available from http://opensource.idi.ntnu.no/projects/jexercise/

Modeling Framework (EMF), i.e. the code is generated from a Ecore model similar to the one shown in Figure 2. Each JavaElement subclass has methods for finding and comparing itself against the corresponding element in Eclipse JDT's internal model of the Java exercise project. As the latter model is continuously updated when the student edits the code, JExercise is able to have a correspondingly up-to-date model of what parts of the solution is student has implemented.

The JExercise view is an extension to the org.eclipse.ui.views extension point, and uses JFace and SWT for the GUI. The drop-down control implements a view on the exercise (.ex) files in the Java exercise project, and hence listens for changes to the resource structure to ensure the list of available exercises is up-to-date. The tree control is similarly a view on the requirements model (and indirectly the solution model), and hence listens to changes to the exercise code. Any change triggers a refresh of the icons, so they always reflect the current state of satisfaction of the *syntactic* requirements.

Since JUnit test runs are time-consuming, they are must be triggered manually. JExercise listens for completed runs and updates the model and tree label icons based on the success or failure of the individual tests (Unfortunately, it's not possible to listen to individual runs, so JExercise must listen "globally" for completed test runs and look up the requirement based on the provided (and undocumented) test run *name*).

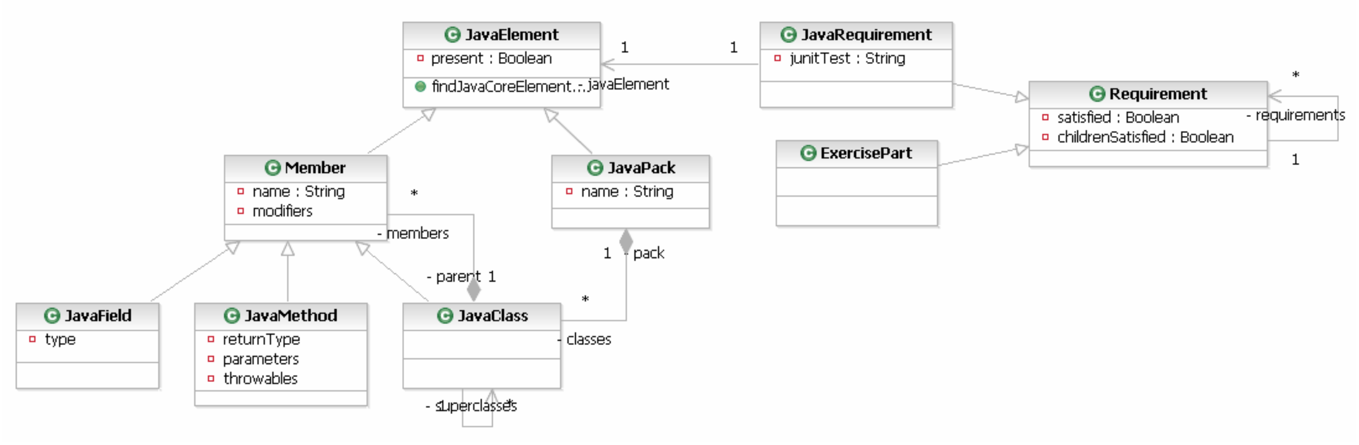


Figure 2. The JExercise model

7. EVALUATION

JExercise was implemented during the fall and early winter of 2005 and was used by 400-500 students for the exercises in TDT4100 – object-oriented programming (see http://tdt4100.idi.ntnu.no/ for details and example exercises) the following spring. JExercise was used on Windows XP (thin clients and standard PCs), Linux and MacOS X. Although many students had initial installation problems on the Unix-based platforms, due to Eclipse and Java VM issues, JExercise worked without causing much frustration.

During the semester, the students used a web-based system for providing feedback (and letting out frustration). The main complaints were that the requirements, as formulated in the exercise text, were difficult to understand, vague and ambiguous. Sometimes this was intentional, to avoid giving too much guidance, but we also admit that it was harder than we thought to formulate precise and complete requirements.

The student assistants had the task of scoring the exercises, based on JExercise test runs and code inspection. They were allowed to give more points than JExercise indicated, if the code was "good enough". The assistants reported that it was easier to be "hard" on the students with backing from JExercise, i.e. they gave fewer points with JExercise than they would have without it. They also used less time grading, as the code was inspected only when the students felt they deserved more points.

After the semester, the course as a whole was evaluated by means of a questionnaire, with particular focus on the exercises and the JExercise. The main findings were:

- Over 70% of the students felt that precise, testable requirements were a good starting point for implementing the exercises.
- Students liked getting feedback about their progress, without having to get in touch with the course's staff, whether working on campus or at home.
- Most students appreciated the guidance the system gave, while some stronger students considered the exercises too constraining.
- Some felt there was too much focus on (testing) fragments of code, instead of interesting and complete applications.
- Over 70% would have preferred a PC-based exam with Eclipse+JExercise instead of the current paper-based one!

The feedback indicates that JExercise worked well for all but the strongest students. It is therefore recommended to complement this kind of exercises with more open projects, as we do with our game project [7]. The strongest students weren't only dissatisfied, however, as they read the test code with interest, and took pride in fooling our tests and suggesting improvements to remove holes!

Midway through the semester, we gave an exam-like Eclipse+JExercise-based test, to let unfortunate students collect missing points and other students a chance to test themselves. This experience was interesting in several ways: It effectively revealed the students that relied too much on (or simply copied) fellow students. The students that did well used more time than expected to get the code completely correct. This is important to consider if JExercise is to be used for a real exam, as the evaluation indicates would be favorable received.

Finally, the test showed that although Eclipse+JExercise work well, the scalability (all students must take the exam simultaneously) and robustness of the client/server and network setup is a major concern.

A different concern is the resources used on making the exercises, both formulating the requirements and writing the corresponding exercise (.ex) files and test code. The exercises were made from scratch and this work took more time and effort than expected. We expect future work on an editor to help, but the resources needed for developing requirements of high enough quality should not be underestimated.[4]

8. RELATED WORK

The idea of combining testing and programming exercises has been explored by many others. Edwards [4] discuss how testing may be an integrated part of the programming assignments, while Wick [8] discuss integrating it into the curriculum. While the work described here focus on using tests for giving feedback to the students, we also have exercises were the students write tests themselves. Ideally this should be integrated with JExercise, but we haven't found a way of using JUnit tests for testing other JUnit tests.

Most modern Java development tools support JUnit testing, and such functionality has also been introduced into pedagogical programming tools like BlueJ [2][6]. Since BlueJ, like Eclipse, has an extension mechanism, we thought of writing JExercise for BlueJ, but quickly found that Eclipse was better suited, both for our course's learning goals and technically.

There exists several systems for supporting managing assignments. BlueJ includes a mechanism for submitting code. The Web-CAT project includes an Eclipse for submitting code and advanced tools for automatic analysis and grading [1]. The eAssignment project [3] also extends Eclipse with functionality for submitting, managing and testing code. Both of these, however, focus more on the teachers' work(flow) than on supporting the students' learning process, thus complementing, rather than competing with our work.

9. CONCLUSION AND FUTURE WORK

We have presented a specification-based and test-driven exercise support plugin for Eclipse, named JExercise. The plugin gives the student continuous feedback about her progress and lets her test the code for correctness with respect to the exercises' requirements.

The evaluation after one semester of using JExercise in a introductory course with 500+ students, indicates that the students appreciate the feedback and guidance that JExercise gives. The increased resources used for authoring the exercises, should, however, not be underestimated.

The planned work goes in two directions. First, we will develop an editor to improve the most time-consuming and error-prone aspects of the authoring process. In particular, much of the structure and cross-references may be generated: the solution model may be partly generated from the real solution, the requirements structure from the solution and the structure of the exercise HTML from the requirements structure. Second, we plan to integrate JExercise with Web-CAT's infrastructure for automatic grading, both client-side submission and server-based JUnit testing.

[4] An open source repository of exercises would be valuable.

10. REFERENCES

[1] Allowatt, A., Edwards, S.H. *IDE support for test-driven development and automated grading in both Java and C++*. In Proceedings of the Eclipse Technology Exchange (eTX) Workshop at OOPSLA 2005, October 2005 Edwards.

[2] BlueJ Home Page, http://www.bluej.org/

[3] Bruch, M., Bockisch, C., Schäfer, T., Mezini, M. *eAssignment - A Case for EMF.* In Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange, October 2005; San Diego, California, USA; ACM Press, Pages 110-114.

[4] Edwards, S.H. *Adding software testing to programming assignments.* Workshop at the 37th SIGCSE Technical Symposium on Computer Science Education, March 2006.

[5] JUnit Home Page, http://junit.sourceforge.net

[6] Patterson, A., Kölling, M., Rosenberg, J. *Introducing unit testing with BlueJ.* Annual Joint Conference Integrating Technology into Computer Science Education. In Proceedings of the 8th annual conference on Innovation and technology in computer science education, Thessaloniki, Greece, 2003. ACM Press, Pages 11-15.

[7] Sindre, G., Line, S., Valvåg, O.V. *Positive experiences with an open project assignment in an introductory programming course*. In Proc. 25th International Conference in Software Engineering (ICSE'03), Portland, OR, USA, 3-10 May 2003.

[8] Wick, M., Stevenson, D., Wagner, P. *Using testing and JUnit across the curriculum.* In Proceedings of the 36th SIGCSE technical symposium on Computer science education, St. Louis, Missouri, USA, 2005. ACM Press, Pages: 236-240.