

Authoring specification- and test-based Java exercises with JExercise

Hallvard Trætteberg, Trond Aalberg
Dept. of computer and information sciences, NTNU

Abstract

Programming exercises are an important part of an introductory programming course. It is, however, hard to design exercises that force the students to learn what they should and validate the fact that they have. In this paper we describe an approach to designing Java exercises based on precisely specified requirements with corresponding JUnit tests and Eclipse plugins for supporting both the student and exercise author.

1. Introduction

Programming is a practical skill that requires both a conceptual understanding and practical training. By means of lectures and exercises a programming course must support both.

In our introductory course in object-orientation programming (with Java), TDT4100, we have defined three learning goals: basic principles of object-orientation, the Java language syntax and semantics, and development using standard techniques and modern tools (in our case Eclipse). The standard techniques range from conventions like Java beans getters and setters, encapsulation of one-to-many associations and design patterns like “Observer”. Although the course’s lectures may give a conceptual understanding, it is the exercises that are meant to give the students the necessary practical training.

The practical part of TDT4100 is organized as 10+ exercises focusing on small-scale programming, followed by a larger-scale game project. The exercises are meant to train the students in using specific programming techniques and at the same time provide means of testing that these techniques are actually used. The goal is to 1) force the students to write correct code for at least part of the exercise, rather than partly correct code for all, 2) give better and instant feedback during the students’ work on the exercise and 3) reduce the effort needed for validating and grading, by automatically computing a score based on the tests. Our solution is a set of exercises with precise, testable requirements and corresponding JUnit-tests. To support the students in working with and test the exercises, we provide an Eclipse plugin named JExercise[8].

In the following sections we explain our approach to exercise design, the role and design of the JExercise tool and our experiences after our first semester using our approach and tool.

2. Specification-based and test-driven exercises

One of the goals of our exercises is to ensure that the students write correct code, i.e. that the objects exhibit the desired behavior. This means that the exercises must have a precise and complete specification to test against. In addition, the specification must be written in a way that makes it straightforward to write tests to check the correctness.

Since the course’s main learning goal is object-oriented programming, we use an object’s externally visible (and testable) behavior, as the basis for our specification. This is sound both from a methodological and pedagogical viewpoint. In a Java program, the externally visible behavior of an object includes:

- the publicly visible state of the object, i.e. the value of public fields and the return value of public methods, as it changes over time
- external side-effects, like changes to the file system, databases and network
- invocation of callbacks, i.e. that calling a method on one object results in a method call on another object, typically given as argument to the first or previously registered as a listener
- the exceptions that are thrown, i.e. how certain conditions aborts the “normal” flow of control

For each of these behaviors, there are ways of testing correctness. Hence, if an exercise can be decomposed into these elements, the exercise as a whole may be tested. For instance, the canonical first exercise that consists of a main method that writes “Hello world!” to System.out is an example of changing the visible state of an object. To test the behavior, we set System.out to a stream that collects the output, so we afterwards can test that it indeed contains the desired string. (In practice, we use a regular expression that is less picky about character case, white space and punctuation.). An exercise focusing on validation of setter arguments would include elements of object state and exceptions. To test it, the setter is called with both legal and illegal arguments. In both cases we catch any exception and check the value that the corresponding getter returns. In case of a legal setter argument, the value should be changed and no exception should have been thrown, and in case of an illegal argument, the value should not change and an exception of a certain class should have been thrown.

More complex behavior, like the “observer” pattern may also be specified and tested. First we implement a special observer that keeps track of every call to its update method. Then we attach it to the observed object, and set the state that should result in a call to the observer’s update method. Afterwards we test that the update method indeed has been called. Note that to be able to test such behavior, the structure of named elements (packages, classes and methods) and their visibility must be specified. This may seem like a drawback, as it makes the specification fairly detailed. However, many standard practices and patterns prescribe both the specific set of classes and methods that are part of the encapsulating interfaces.

The JUnit framework supports running whole test classes and individual test methods, giving great flexibility for the exercise author and lets us give the students feedback during the work on the exercise, rather than when it is completed. In the simplest case, a single test method tests a corresponding method specified in the exercise, to give the student feedback about the correctness of that method. Fairly often methods are so interrelated that they cannot be tested separately. Instead, a single 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. the standard behavior and boundary cases like null arguments. Of course, for more advanced courses and students, the other extreme case may be desirable, that of testing a whole application with one test method.

3. The student’s view of JExercise

From the student’s point of view, an exercise is a set of hierarchically structured requirements, as shown left in Figure 1. Each line in the tree is a requirement, either for a syntactic Java element or a testable, functional requirement. At the right a browser window shows the exercise text, and when a requirement is selected in the tree the browser navigates to the corresponding text. Since running the tests and gaining points is an important part of the game, a button is provided below the tree for running all the

JUnit tests at once. The button also indicates the current number of points granted, based on the passed tests.

Error! Reference source not found. shows that part 1 of the exercise requires four *syntactic elements* (those in the figure with check marks), the `GameOfLife` class and three (static) methods, `willLive`, `countNeighbors` (leaf nodes) and `stepCells` (interior node) within the `GameOfLife` class. In addition, five *functional requirements* are shown (indicated with ?JU icon), corresponding to the five test methods `testWillLive`, `testCountNeighbors1`, `testCountNeighbors2`, `testStepCells1` and `testStepCells2`.

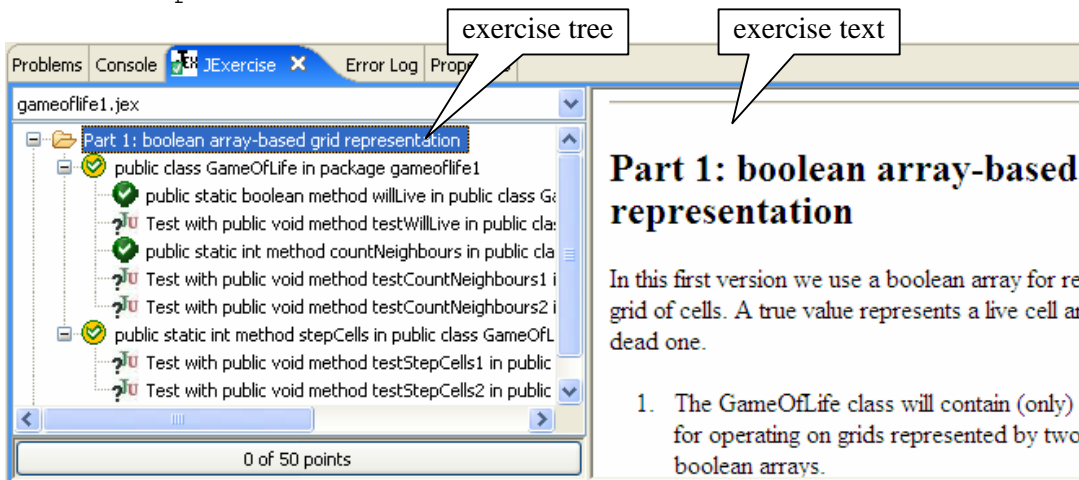


Figure 1 The JExercise view

The icon for a requirement, whether it is syntactic or functional, indicates whether or not the requirement is met. The icons for syntactic elements are continuously updated by hooking into Eclipse's model of the source code. Testing functional requirements using JUnit tests is more costly and must be triggered manually, by double-clicking the node in the tree. When the test run is finished, the success or failure is indicated by the icon. All the four syntactic requirements above are met, as indicated by the check marks. However, since the JUnit tests have not been run, these have question marks, and their parent node is left in an undecided state, as indicated by the yellow icon background. The different icons and their meanings are shown in Table 1.

Table 1. The icons used by the JExercise view

	The state is currently <i>undecided</i> (yellow background) usually because a pre-condition is unsatisfied.
	The requirement is completely <i>satisfied</i> (green background).
	The requirement itself is <i>satisfied</i> (green checkmark), but there are <i>undecided</i> sub-requirements (yellow background).
	The requirement itself is <i>satisfied</i> (green checkmark), but some sub-requirements are <i>violated</i> (red background).
	The requirement is <i>violated</i> (red background).
	The requirement is a JUnit test that has not been run.
	The requirement is a JUnit test that has <i>succeeded</i> (green icon).
	The requirement is a JUnit test that has <i>failed</i> (red icon).

As can be seen, the icons encode three kinds of states. The question mark, check mark and cross indicate the state of the requirement, the background color indicates the state of the sub-requirements (yellow for undecided, green for satisfied and red for violated), and the ?JU symbol if it is testable by means of a JUnit test.

4. Authoring an exercise

An exercise, such as the one shown in Figure 2, is based on three elements:

1. A textual *specification* of the Java elements that are required and their desired behavior. This specification should be precise enough for the students to write correct code.
2. A set of *JUnit tests* for checking whether the student's code meets the specification. These tests should be comprehensive enough to find most, if not all, coding errors. In practice, it may be difficult to avoid false negatives or positives.
3. A *model* of the required solution and the correspondence between the textual specification, required Java elements and the JUnit tests. This model is used by JExercise to guide the student and give her feedback about the progress by means of requirement icons and points.

We will illustrate our approach using Conway's *Game of Life* as an example. Game of Life is a simulation of a cell colony in a grid, based on a very simple set of rules. A cell in the grid can be either "live" or "dead" and its next state is determined by the current state and the number of live neighbors. Game of life is a good starting point for a set of exercises, since it tests basic skills in the procedural part of Java and lends itself to an object-oriented design, with classes for the grid of cells and the game as a whole. The following sections should give an impression of the level of detail and precision that our approach requires, how various aspects of Java and object orientation are addressed, how testing may be performed and how the JExercise tool is used to build the model.

4.1 Game of Life – part 1

The first part of the exercise will focus on the rules of the game and the representation of the grid as an array of booleans (true means the cell is alive), using the procedural part of Java. It will require the class `GameOfLife` and the four methods, `willLive`, `countNeighbors`, `stepCells` and `printCells`. `willLive` must take the current cell state (a boolean) and neighbor count (an int) as arguments and return a boolean according to the rules of the game. `countNeighbors` must take a two-dimensional boolean array and x,y coordinates (ints) and return the number of live cells surrounding the x,y cell. Finally, `stepCells` must take two two-dimensional boolean arrays, using the first as the current grid state and filling the second with the next state. Additionally, it must return the number of changed cells.

While the tests are used for counting points, their main role is giving the student good feedback. Hence, we would like to have separate tests for each requirement to make it is easier for the student to progress in small steps. We test `willLive` with one test method, `testWillLive`, `countNeighbors` with two test methods, `testCountNeighbors1` and `testCountNeighbors2`, `stepCell` with two methods `testStepCell1` and `testStepCell2`, and `printCells` with `testPrintCells`. Since it is difficult to anticipate the errors in the students' code, we try to test exhaustively where we can. `testWillLive` tries all the possible combinations of cell states and neighbor counts (2x8 combinations), `testCountNeighbors1` tries all cell patterns

surrounding a cell (256) and `testCountNeighbors2` tests the neighbor count of all boundary cells in a fixed 3x3 grid. Finally, `testStepCells1` performs one step for all 3x3 grids, while `testStepCell12` steps a so-called glider in a small grid until it ends up as a block against the wall.

The fact that the test code is available for the students to both read and run, makes writing the tests a bit special. We actually want our students to read the test code and learn from it, but we also want to avoid that they read and use it for coding their own solution, without learning what they should. First note that a JUnit test usually compares the result of running the students' code against the correct values. If we just include the correct values in the tests' code, the students may just copy the expected results into their own code, without making a general solution. If we instead include some of our own solution, the students may mindlessly copy that code instead of writing their own. We use two strategies to avoid helping the students: 1) we try to generate random or exhaustive test data, and 2) we use non-standard techniques that are difficult to copy. E.g. in the test code that counts the neighbors of random test grids, we represent the grids as integers and use bit operations for computing the count. A third strategy, which we will come back to, is giving the students a limited set of tests and use more comprehensive ones for computing the score, after their code is submitted.

4.2 Game of Life – part 2

The second part of the exercise focuses on hiding the array-based representation by *encapsulating* it in a `Cells` class. To teach the students how this is done in practice, and not what will finally be externally visible features of the class, we *initially* specify *both* the private fields holding the internal data *and* the method interface that other classes use for accessing the internal data. Our `Cells` class must have a private two-dimensional `boolean` array field named `cells`, a constructor taking the grid dimensions as arguments, the methods `getHeight` and `getWidth` for retrieving the grid dimensions, the methods `getCell` and `setCell` for accessing to the content, and a `copyCells` utility method for copying the content from another `Cells` object. The existing `countNeighbor` method is moved into the `Cells` class. Finally, the `GameOfLife` class must be recoded to use the new `Cells` class.

It is important to consider boundary cases, so we specify that all methods must consider cells outside the grid as dead and that setting cells outside the grid should have no effect (rather than throw an exception). In our experience, it is easy to forget to be explicit about such boundary cases, and by writing the solution and tests before formulating and publishing the text, more of these cases are uncovered before the students discover them.

As for part 1, we want to test each requirement separately. In practice, the getter methods are often tested implicitly, in the tests for the methods that change data. The test for special cases, like getting and setting cells outside the grid, are separated from the typical case, to provide better feedback to the student.

4.3 Game of Life – part 3 and 4

In part 3, the method interface of the `Cells` class is made explicit by redefining `Cells` as an interface and using the `Cells` class from part 2 as the basis for an implementation named `BooleanArrayCells`. That this is a common process is highlighted by the fact that Eclipse supports the extraction of an interface from a concrete class as a standard refactoring. A second implementation named `BitSetCells`, based on the standard `BitSet` class, is also required for training purposes. The fourth exercise part completes

the refactoring by extracting the commonalities from the two implementations into an abstract class, that the two concrete implementations must extend. Note that parts 3 and 4 do not introduce new behavior, as they mainly restructure existing code or implement behavior that is already specified. Although the value of the restructuring such a simple design is small, the process of introducing interfaces and abstract classes in general is important to learn.

Although we may test all parts of the exercise and provide feedback to the student about the progress, the students' feedback indicates that the sense of achievement is greater if the result is a complete application. Hence, we provide a pre-implemented Swing GUI written against the `Cells` interface and `GameOfLife` class that may be run once part 3 has been implemented.

4.4 Building the JExercise model

JExercise requires a model of the exercise that is conceptually split in three, a *solution* model, a *test* model and a *requirements* model. The exercise author builds all three models, while the student only sees the requirements model, visualized as a tree as shown in Figure 1.

The solution and test models are both models of Java code, the code the student must write and the code that tests the student's code, respectively. These models follow the hierarchical structure of Java code, with packages containing classes and classes containing methods and fields. For classes and interfaces, the inheritance chain is represented, i.e. the classes and interfaces that are extended and implemented. For methods and fields type information is included. If relevant, modifiers like `public` and `static` are also included.

Everything represented in the solution model may be checked against the student's code, e.g. whether or not a specific class is present, if it extends the specified super class

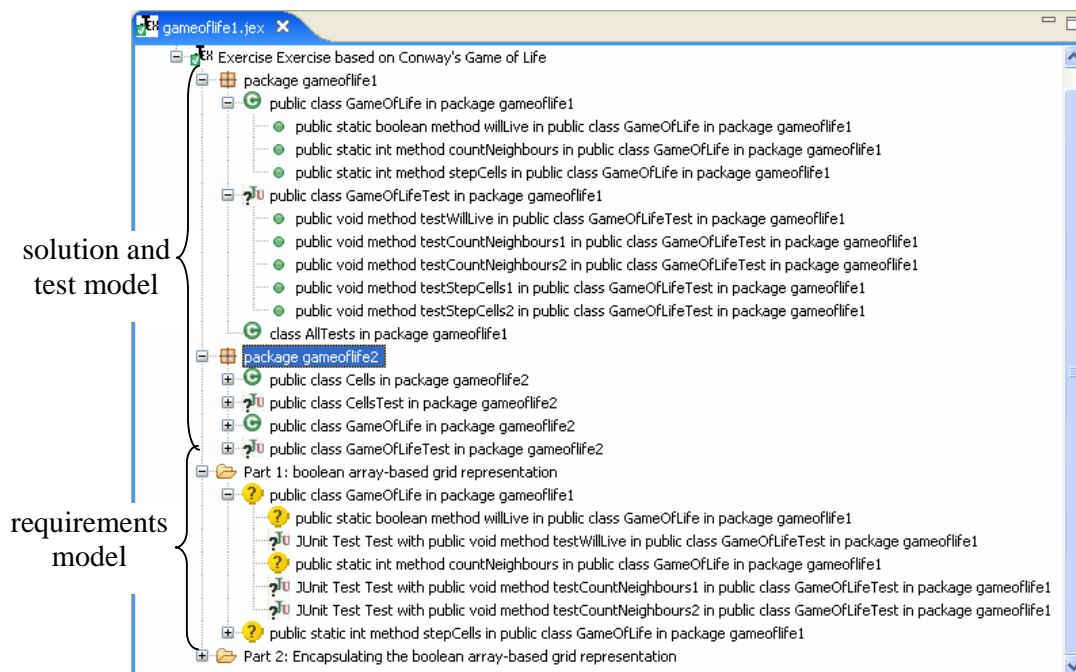


Figure 2. The JExercise editor

and inherits the specified interfaces, and if the specified methods with the correct visibility and return and argument types are present. Fortunately, Eclipse's JDT

component provides this information, so we don't have to actually analyze the student's code.

The requirements model is the student's view of the exercise, and references the solution and test model to represent both the syntactic Java elements that are required in the student's code and the JUnit tests that may be run to test the student's code. The requirements model is structured to guide the student through a natural coding sequence and to indicate when tests should be run. 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 level and students for the same programming problem.

As described above, the three-part model includes a lot of structure and detail about both Java code and the student's tasks. To support authoring the model, we have built an editor that is able to extract most of this structure and detail from the exercise's solution. Briefly, the process is as follows.

First, the learning objectives of the exercise are made explicit and the exercise solution is designed and coded. It is important to complete this step, to ensure that the required code fits with the learning objectives and does not require knowledge the students lack. The tests should be written as part of the design, since they often improve our understanding of the problem and help uncover pitfalls. From the design, solution and test code, the initial model may be created. Our Eclipse plugin includes a wizard for JExercise model files that reads the source code (or rather, Eclipse's model of it) and creates the corresponding solution and test model, with all the details filled in. This part of the model is shown in the upper part of the editor window in Figure 2, with tree nodes corresponding to packages, class and methods and fields.

Second, the structure of the requirements must be decided and represented. As a start, the parts of the solution that are explicitly required must be identified. Usually this includes all the public parts of the code, but it may also include private fields, as discussed in section 4.2. Next, the hierarchical structure and sequencing of coding tasks must be represented; in particular which main *parts* the exercise will consist of. The editor includes commands for filling a part with requirements corresponding to specific solution elements and tests. E.g. from a class `C` and method `m` and test class `CTest` with test method `testM`, it will generate a requirements that class `A` and method `m` are present and a requirements that `ATest` and `testM` must pass. This part of the model is shown in the lower part of the editor window in Figure 2, with tree nodes corresponding to requirements for classes and methods and test classes and test methods. To change the structure and sequence, requirement nodes may be moved around using drag'n drop or cut'n paste, or deleted. E.g. you may want to move a requirement for a boundary case to a later part or remove a requirement corresponding to a specific test method, while keeping the one for the test class as a whole.

The last step is authoring the exercise text. All the nodes in the requirements model may be annotated with HTML fragments describing the purpose and specific requirements for that node. From the XML for the whole model containing all these fragments and an appropriate XSLT file, a complete HTML file for the exercise may be composed. In a previous version, the author had to make the HTML separately, but it quickly became difficult to keep it consistent with the requirements structure, as the exercise changed. By adding HTML fragments to the model and utilizing XSLT, the authoring process is made more robust, and reuse of exercise parts becomes a lot easier.

5. 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 spring 2006 and spring 2007. 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 steam). 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, writing test code and building the exercise model. The exercises were made from scratch and this work took more time and effort than expected. Although the editor, which was implemented after the first set of exercises were developed, has helped a lot, the resources needed for developing requirements of high enough quality should not be underestimated.

6. 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 [9] 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 where 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][5]. 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.

[10] presents a system for automatic grading, combining both assignment management and evaluation (grading). Each submission is run on a large set of test cases and the output compared to the correct answer. Although the output is reported to help the students in diagnosing problems, the goal is not to give the student feedback but to save man-hours used for grading.

7. 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. JExercise is open source and may be downloaded from <http://www.opensource.idi.ntnu.no/homepages/jexercise/>.

The evaluation after one semester of using JExercise in an 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. To improve the learning process, we want to undertake a more qualitative evaluation, to better understand how the students use JExercise, and identify possible improvements, like linking to the text books, code examples etc. To improve the quality and lower the cost of evaluating exercises and exams, we will look at ways of utilizing JExercise in a server-side evaluation system, among others by integrating JExercise with Web-CAT's infrastructure for automatic grading.

References

- [1] Allowatt, S.H and Edwards, “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] M. Bruch, C. Bockisch, T. Schäfer, and M. Mezini, “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] S.H. Edwards. “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., M. Kölling and J. Rosenberg,,”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] G. Sindre, S. Line and O.V. Valvåg, “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] H. Trøttestad, T. Aalberg, “JExercise - a specification-based and test-driven exercise support plugin for Eclipse. In Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange. October 22 - 23, 2006, Portland, Oregon, USA.
- [9] M. Wick, D. Stevenson and P. Wagner, “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.
- [10] B. Cheang, A. Kurnia, A. Limb, Wee-Chong Oonc. “On automated grading of programming assignments in an academic institution”. Computers & Education, vol. 41, pp. 121-131. Elsevier 2003.