Lunds tekniska högskola
Datavetenskap, Dec 7, 2015
Görel Hedin/Ulf Asklund
EDA260 Programvaruutveckling i grupp—projekt

Lab Exercise
Test First using JUnit

Goal

This lab is intended to demonstrate basic Test First practice. You will use Eclipse with its built-in support for CVS and the JUnit testing framework.

Note!

- During the lab, you will write down things to do on a todo list. Be prepared to show and discuss this list with your supervisor.
- Use good names in your code, both in test code and in production code. Be prepared to show your tests and code to your supervisor.

1 Practices

In this lab exercise, you will focus on the following work practices:

Todo list For each story or task, think about how you can break down the work into small items that you can test. Jot down a name for each test on a list (it is easiest to use pen and paper for this). Use this test list as a todo list. While you are working, maybe you will identify more things that need to be done. For example, you might want to refactor your code. Or you might think of new things to test that you didn't think of earlier. Just add these items to the todo list.

Baby steps Work in small steps. Pick one item from the todo list. Complete it before you start on another item. Cross out the items as you complete them.

Test first When you are implementing a test case, write the test first. Add some stub production code to make it compile. Check that the test fails. Then write the production code that implements the test. Check that the test works. Is any refactoring needed now? After possible refactoring, you are ready to commit to the repository. You have added a new piece of working functionality.

Pair programming You will work in pairs with the role of *driver* (has the keyboard) and *partner* (has the todo list). Switch roles often. For example, switch every time you start on a new test case, or even more often. Make sure you try out both roles about the same amount of time. As driver, explain what you are doing as you code. As partner, ask questions if you don't understand, suggest improvements and point out possible errors. As partner, keep track of the todo list and make sure to note down new items. You are working together. When you identify problems, speak in terms of "we" instead of "you". For example: "Oops! we forgot to declare x..."

Commit often As soon as you have completed a new little piece of functionality, you can commit your code. Try to commit as often as possible. But remember that the code you commit must compile without errors, all the existing test cases must work, and your code should be of high quality (easy to read, good names, etc.). Feel free to also update often. However, in this lab exercise, no one else will commit to your module, so there will be no changes when you update.

2 Getting started

The scenario for this exercise is to implement the queue system for a bridge similar to the Øresund bridge. The system will be a little simpler though, there will be no BroBizzes, and there will only be a single lane before paying the toll fee and entering the bridge. The bridge is not yet built, and the idea is to implement this system in order to simulate various traffic flows.

The customer has written a few stories and a senior developer has divided some of them into tasks. The stories have been ordered by importance and can be found in Appendix A. The senior developer has already implemented a skeleton of the system, including two classes and a test case, constituting task 1 for Story 1.

2.1 Setting up the workspace

Open Eclipse, create a new Eclipse workspace if you don't have one already. You are now to check out a module `teamXX/BridgeSimulator`. Your supervisor will assign you a number to replace `XX` in this name. Make sure you check out the subdirectory `BridgeSimulator`, and not the whole `teamXX` directory!

The module is in the repository `/local/cvs/eda260/testlab` on the CVS host `cvs.cs.lth.se`, and you should use the pserver CVS account and password that you have received earlier. Here is how you check out the module:

- Enter the CVS perspective: `Window -> Open Perspective -> CVS Repository Exploring`.

  If the CVS perspective does not exist you need to install the CVS plugin.

  - Open a web browser and go to: `https://marketplace.eclipse.org/content/cvs-integration`
  - Drag the "Intall button" to your running Eclipse workspace. Follow the instructions, including a restart of Eclipse.

- Add the appropriate repository location: `context menu -> New -> Repository Location`
- Select the appropriate module: Open the repository folder, then `HEAD`, then `teamXX`, and then select `BridgeSimulator`.
- Check out the `BridgeSimulator` module: `context menu -> Check Out`
- Go back to the Java perspective. A new project `BridgeSimulator` has been created in your Eclipse workspace.

2.2 Take a look at Eclipse CVS support

Take a look at the project code. Note the CVS information that Eclipse shows for different files:

- How do the icons for files and directories differ from projects not under version control?
- What CVS information does Eclipse associate with the directories?
- What CVS information is associated with the files?
- How would you do commit and update? (*Hint!* Look at `Team` in the context menu.)

2.3 Are you in a clean state?

Make sure the code you have checked out is in a clean state: there should be no compilation errors, and all tests should work. Here is how you can check if tests work:

- Select the project directory.

- Run JUnit: `context menu -> Run as -> JUnit Test`.

A new JUnit tab opens beside the Package Explorer. The JUnit tab runs all JUnit test cases found in the project directory. Does the bar turn green?

2.4 Experiment with the tests

Now, take a look at the production code and the tests. Just as an experiment, change a test or the production code so that a test fails. Go back to the JUnit tab and rerun the tests (find the button `Rerun Test` in the upper part of the JUnit pane). What happens? Look at the `Failure Trace` and try to understand the message.

Now, change the code back again so that all tests should pass. Rerun JUnit again. What happens?

3 A little planning and design

Now the code should be in a clean state, and you are ready to start developing new functionality. But where should you start? First, you need to do a little planning and a little design:

- Discuss the current design briefly. What classes are there already?
- Take a look at the stories and tasks in Appendix A. Do you think the current design is sufficient, or will more classes be needed? Just sketch a little on paper so you get an idea of what might be needed. Don't implement anything of this yet. You can implement parts of the design as needed when you implement the tests. And feel free to change your mind about the design while you implement.
- Pick a story or task and try to break it down into small pieces of work, each corresponding to a test case. Write down these items on a todo list.

Lets take a look at the first story to get an idea of how this can be done. The story is *Vehicles arriving at the bridge should line up in a queue*. The senior developer has explained that the queue will be simulated by a `VehicleQueue` object, and she has already implemented a test case `newQueueShouldBeEmpty`. What more things could we consider testing to make sure the story is implemented? What about:

- **One vehicle**: After a vehicle has arrived, it should be part of the queue.
- **Vehicle order**: If a vehicle arrives after another vehicle, it should be placed after that vehicle in the queue.

Are there more things that would be appropriate to test for this story? Don't try to include issues not related to this story. Perhaps the above is sufficient?

4 A baby step

Now we will do a baby step forward. We want to add a piece of working functionality.

4.1 Pick a test case

First, decide which test to start with. Pick the simplest one that will take you forward in the implementation. Now, write a JUnit test method that will perform the test. Don't worry about that the production code is not already there. Instead, take advantage of this so that when you write the test code, use method names that feel natural, making the test code easy and clear.

Probably, you got some compilation errors since the production code does not yet include the new methods. Add stubs for these methods so that the code compiles.

4.2 See it fail

Now, run the JUnit tests. Your new test case will probably fail. This is expected and should feel good: you know now that you managed to write a new test case that in fact tests something new that isn't yet implemented.

4.3 Make it pass

Now, implement production code to make the test pass. (*Hint!* Make use of existing Java class libraries if you find something that fits in.) If you think of more tests to add at this point, just write them down on the todo list. Don't implement them yet. Run JUnit to make sure all tests pass. When you get the green bar, this should feel really good. You have made progress!

4.4 Reflect on the design

You have implemented a new piece of functionality. Is the design good? Are the names you chose good? Is your code easy to read? If you added a new public method in the production code, maybe you should add a JavaDoc comment? Write down the improvements you come to think of on your todo-list. Do these improvements now. Rerun the JUnit tests to make sure all tests still pass.

4.5 Prepare for commit

Now you seem to be in the position to commit your code. You have added a piece of new functionality that works, and the code is looking good. But before you actually commit, you need to check that nobody else has done commits since your latest update, because in that case you need to update first and merge their commits:[1]

- Select the project directory. This is important — you want to check the status of *all* your files.
- To check the status of your files, do `context menu -> Team -> Synchronize with Repository`. A window pane appears where you can see if there is anything new in the repository, and if any of your files has explicit conflicts that need merging.
- If there is anything new, do `context menu -> Team -> Update`. Now, merge any changes, compile, and test. If everything seems fine (green bar), go back to 4.4 to reflect on if any refactoring is needed after the integrated changes. If any test failed (red bar), go back to 4.3.
- If there are no new changes in the repository, you can continue to the next step, the actual commit.

4.6 Commit

Now you are *actually* in the position to commit your code. You have added a piece of new functionality that works, the code looks good, *and*, a few seconds ago, you checked that your code is up to date with the repository.

- Select the project directory. This is important — you want to commit *all* files in the module to make sure the repository is kept clean.
- Do the commit: `context menu -> Team -> commit`
- Fill in a suitable commit comment. For example, story number, task number, brief description of what you have added or changed.

[1] Of course, in this lab, no one else should be committing to your module, but you might as well get into the habit...

5 Continue development

5.1 Another baby step

The goal in this lab is not to finish as many tests and stories as possible. The goal is to get used to the work practices and to produce working code of high quality.

Now, do another baby step:

- Write a test.
- See it fail.
- Make it work.
- Refactor.
- Update.
- Commit.

And don't forget pair programming:

- Switch pair programming roles often!
- Work together!
- Celebrate all your green bars!

5.2 Discuss with your lab leader

Reflect on your way of working. Call on your lab supervisor and discuss how you are progressing. Be prepared to show your todo lists, your code, and answers to the questions in the text above.

5.3 A few more baby steps

Now continue with some more baby steps. When you are done with one story, pick the next one, do a little planning and design, breaking down the story into tasks and test cases. Then pick a test and continue with baby steps.

5.4 Practice different parts of the JUnit API

Continue taking baby steps. As you go forward, make sure to practice the following kinds of test support:

- Explore the different assert methods. Use them in different test cases.
- Make use of test fixtures.
- Test a method that should raise an exception.

5.5 Bigger steps?

After you have gone through several baby steps, you may consider what will happen if you take slightly larger steps.

For example, try skipping the "See it fail" step. This might save you a little bit of time. But on the other hand, are you sure now that your new test case really tests something new? The uneasiness that follows from not "seeing it fail" could actually slow you down.

Another way of taking bigger steps could be to write several test cases at once. But if you write many test cases at once, it will take a long while before you are back to the green bar again. You will have to find a balance. The important thing is that the time between green bars is kept short!

Another very important thing is that you do write the test *before* its implementation. Why? Try to think of at least three reasons.

Of course, it may happen that we find out after the fact, that we have missed writing some tests. Naturally, it is better to add those tests at that point, rather than to not add them at all.

5.6 How much should I test?

You can't test all possible cases. How many tests should you write? The usual XP advice is to "test everything that could possibly break". You will need experience to find the right balance. One rule of thumb is to test both *typical cases* (for example, remove an element from a list with three elements) and *limit cases* (for example, remove an element from a list with just one element, or no elements). Another rule of thumb is to strive for *code coverage*: If you run all your tests, every line of your production code should have been executed at least once. There are tools that can check this.

In XP, writing test cases is primarily a way to drive the design and implementation forward. This way of working, writing a test first, and then the corresponding implementation, is often called *test-driven development* (TDD). A nice side effect is that you get a regression test suite that gives you confidence in that your code works after changing it. And your test cases document what your code can do.

6 Final Reflection

When you feel that you master the work practices and using JUnit, call for the lab supervisor and discuss with her/him what you have learned. Be prepared to show your todo lists, your code, and answers to the questions in the text above. Also, fill in the table below.

Which stories are done?	
Approximate number of tests	
Approximate number of commits	
Approximate number of pair role switches	
Different kinds of asserts used	
Example of fixture	
Example of testing an exception	

Appendix A — Stories and Tasks

Story 1	**Vehicles arriving at the bridge should line up in a queue**	#Tests	
Task 1	Skeleton program with test case	1	DONE
Task 2	Finish implementation of queue		

Story 2	**A vehicle with a particular registration number can be removed from the queue**	#Tests	
Task 1	Vehicles should have registration numbers		
Task 2	Remove a vehicle, based on its reg number		
Task 3	What happens if the vehicle is not in the queue?		

Story 3	**It should be possible to print the queue in order**	#Tests	

Story 4	**Motorcycles can be removed from the queue in case of high wind speed**	#Tests	

Story 5	**A manual payment station charges the first vehicle in line and allows it to enter the bridge**	#Tests	

Story 6	**The payment station keeps track of the total sales amount**	#Tests	

Story 7	**Prices differ for different vehicle types**	#Tests	

Story 8	**A vehicle can buy a round-trip ticket**	#Tests	

Story 9	**Round-trip tickets receive a 5 percent discount**	#Tests	