CPS122 Lecture: Unit Testing / Test-First Development

Last revised February 16, 2022

Objectives:

1. To introduce test-first development
2. To show how to use JUnit

Materials :

1. Projectable diagrams for UMLImplementation labs and class `EnrolledIn`:
   a. Overall class structure
   b. CRC Card for `EnrolledIn`
   c. Detailed Design for class `EnrolledIn`
2. Code for demos
   a. Netbeans project with skeleton of class `EnrolledIn` (methods not implemented) and no JUnit tests.
   b. Netbeans project with completed tests.
   c. Netbeans project with completed tests and code.
3. Choosing test cases activity and Netbeans demo project to go with it

I. **Test-First Development**

A. As you may recall, we saw earlier that coding comprises about 1/6 of the total effort on a project. How much effort do you think testing comprises (or at least should comprise)?

ASK

About 50%!

(The fact that there's so much buggy software out there suggests that what should be done and what actually is done are not the same thing!)

B. We will talk more about testing later in the course, but for now note that there are actually several different kinds of testing that need to be done.

1. Unit testing tests the individual pieces of the system - e.g,, in an OO system, the individual methods. Any errors discovered by this process are fixed before development proceeds. We will look at an approach to doing this shortly.

   This kind of testing should be done during implementation.

2. Integration testing puts several (already tested) units together to test how they work together. Any errors discovered at this time are likely the result of a misunderstanding about the interface of one of the methods.

   EXAMPLE:

   a) Suppose a certain method is required to compare two objects and return true or false based on their relative order in a sorted list. Assume this method is used by another method that actually sorts the objects.

   b) Suppose the author of the method understands the expectation to be that the method returns true if the *first* object belongs *before the second*, but the author of a sorting method that uses it assumes that it returns true if the *second* object belongs *before the first*.

   c) Both methods would test successful during unit testing (based on their author's understanding of the interface between them.) However, the error would show up in integration testing with the output being backwards!

   d) This kind of testing, too, should be done during implementation.

3. System testing tests the overall functioning of the system relative to the specifications (the use cases). By its very nature, this can't be done until implementation is at least nearly complete.

4. Regression testing is the repetition of tests that have already been passed after a fix/change has been made to be sure that the change has not broken another part of the system.

a) This sort of testing is done as needed during implementation

b) It is also done during maintenance

c) To facilitate this, tests are automated where possible. (We will shortly see an example of this with unit tests).

5. The "50% of effort" rule of thumb includes all kinds of testing that occur during implementation and testing of implemented code, of course. (It does not include regression testing during software maintenance).

C. Today, we are going to focus on a type of testing done as a class is being implemented: unit testing.

1. Many software development organizations actually use an approach to this known as <u>test-first development.</u>

a) The idea is this. <u>Before</u> writing the code for a method, one first writes the specification (perhaps in the form of comments) and write code to test the method <u>before</u> the method itself is actually written.

b) This may seem counter-intuitive - but the idea is that writing a specification and a test helps clarify what is to be done before actually doing it.

Experience has shown that this actually significantly improves code quality and effort.

c) Last semester those of you who were in CPS121 saw an approach like this using pydoc and pytest. We have already seen that there is a similar facility for documenting Java code known as javadoc, and we will shortly see that there is a similar facility for unit testing Java code known as JUnit.

d) You will use this approach during labs 8-10, and will also be expected to use it on your team project. In fact, I will not give you any help with code unless I first see your specification in the form of prologue comments and - in the case of model classes - your JUnit tests.

II.JUnit

A. For Java, there is a unit-testing framework called JUnit which performs similar functions to those provided by pytest - though, of course, the details of using it are quite different.

B. We will introduce this framework via an example, using a class that will be part of the the code you are working with for Labs 8-10. First a brief introduction to the student registration system being used in these labs might be in order.

1. The labs are built around the following class structure. (Actually, what we are looking at here represents just a portion of the overall structure. The lab requirements contain a more detailed structure.)

   PROJECT Overall structure

2. The heart of the registration system model is a collection[1] of courses and a collection of students related by a relationship.

   a) Courses and students are related by a many-to-many relationship called `EnrolledIn`, using an association class because the `EnrolledIn` relationship has an attribute called `grade`.

   b) The `EnrolledIn` relationship is bidirectional - `Course` objects "know" about the `Student` objects in them, and `Student` objects "know" about the `Course` objects they are in.

   c) To model this relationship, both Student and Course objects hold a collection[1] of EnrolledIn objects, and each EnrolledIn object holds a reference to a particular Student and Course, as well as a grade.

[1] When we use the term “collection” with a small “c” we mean any kind of collection (including a Map) - not just a class that implements the Collection (capital “C”) interface.

3. For this example, we will focus on the class `EnrolledIn`. Its role can be described by the following CRC card:

    a) PROJECT `EnrolledIn` CRC Card

    b) PROJECT Detailed design for `EnrolledIn`

C. Demonstration of developing unit tests for `EnrolledIn`

1. PROJECT skeleton for class (just prologue comments and method prototypes- no code)

2. Before we actually begin writing code for the methods of `EnrolledIn` we can develop mechanisms for testing them.

3. Demonstrate creating JUnit tests using NetBeans

    a) Point out no test folder in project

    b) Create tests

    c) Show test folder and contents of created file.

4. Open project with completed tests and show tests.

5. Demo implementation of a couple of methods and show results in testing

6. Open project with completed tests and code and demo testing.

7. Demo some errors and show how JUnit catches:

    a) Omit assignment of grade in `setGrade()`

    b) Omit initialization of `student` in Constructor

    c) Omit initialization of grade in constructor to "WIP".

D. Do choosing test cases activity

1. Distribute and have students do

2. Discuss cases with class

3. Demo using my test cases with Netbeans - progressively correct errors until all green

4. Note that these test cases don't actually catch all errors - e.g. since the erroneous line in last case is

    ```
    int [] roots = { -b/2*a };
    ```

    Then the test we have used will pass, since a = 1 and multiplying by it has no effect!

5. How would we improve the tests cases to catch this errors (and other similar ones in other cases)

    - use test data in which a is not 0 - e.g. multiply a, b, and c by 2

    ```
    e.g. 2, -10, 12;
         2 -2, 2
         2 -4 2
    ```

    - demo how this now catches just the last error; works OK when code is fixed