HOMEWORK 3
Informatics 2-Software Engineering 2014/15

Ajitha Rajan

23rd October 2014

1 Overview

Implement the CCS design you handed in for homework 2. The customer insisted on Java, so Java it is. Use the Eclipse IDE for your implementation and test. Code for the interfaces is available on the web page (make sure you've downloaded the most recent version and keep your eye out for possible updates). Make sure you pay attention to these and the rest of the document. If your code does not compile and run, you will be severely penalized in the grading. You are required to follow the coding standard provided by us in an appendix to this document. You will create automated system-level tests for your CCS implementation and measure the code coverage achieved by your tests. By system-level, we mean tests will test CCS as a whole, not the subsystems defined in your design document. Thus, all tests will provide input to CCS through the interface we provided, and will verify that the output matches your expected output. We have provided sample tests and expected output that your implementation is expected to pass at the minimum. Note: Naturally, this does not preclude you from performing unit testing on your classes and subsystems to ensure that they do what you think they are doing. You are just not required to submit your unit tests and unit-test infrastructure as part of this assignment. Detailed expectations follow below.

2 CCS Implementation Notes

1. First download the existing project from: `http://www.inf.ed.ac.uk/teaching/courses/inf2c-se/Coursework/coursework-2014/CruiseControl.zip` This can be placed anywhere, a temporary directory for example.
2. Open up Eclipse and select (or create a new) workspace
3. Select File→Import→General→Existing projects into workspace
4. Browse to wherever you unzipped the project to and select the folder 'CruiseControl'
5. Make sure the "Copy projects into workspace" option is selected.

You will find several Java source files. You should **only modify** `CruiseControlSystem.java` and `BasicTests.java`, the rest are **support code that you should not modify**.

Open the BasicTests.java file and you should find that you can `Run` this as a `JUnit` test. When you do so, you will find that all three basic tests fail. Question: Can I run this from the command-line? Answer: Yes, try this:

```
$ cd to CruiseControl/src/ directory
$ javac -cp /usr/share/java/junit4.jar *.java
$ java -cp .:/usr/share/java/junit4.jar org.junit.runner.JUnitCore BasicTests
```

2.1 Input

Input can be given via a file. Each separate line is treated as an input state. You can see in 'CommandLine.java' that the input states are read in from a file given on the command-line. Alternatively input state lines can be given as a List or Array of Strings which each String representing an input state. In 'BasicTests.java' you will see how this is used to perform automatic testing of two very basic properties.

Each input state, gives the values of the sensors on the car, these are in the order defined in 'InputState.java' and are as follows:

1. Engine status as a boolean
2. Speed as a double
3. Brake pedal position as a double
4. Accelerator pedal position as a double
5. Start ccs button as a boolean
6. Stop ccs button as a boolean
7. Start accelerating button as a boolean
8. Stop accelerating button as a boolean
9. Resume button as a boolean

Each value is separated by a single space character. So an example input state line is:

```
true 50.0 0.0 1.0 false false false false false
```

In addition, you can specify that a given state has not been changed since the previous pulse. This is specified with the '-'.

Output states have the same format, but with an additional value at the end. The additional value is a double which represents the position of the throttle. Output states cannot contain '-' characters.

2.2 Additional Hints

You will need to make sure you set the status of the buttons correctly. The input states may cause the buttons to be pressed, but they may also use the '-' character to leave them in whatever position they were last set to either by an input state or by your CCS implementation.

Just because you set the accelerator to a particular position does not mean that the speed of the car will reflect this. It may be going over terrain which responds differently than expected. However, *your* task is simply to follow the requirements and set the throttle to the correct position.

2.3 Version Control

It is recommended, but not required, that you keep your implementation and tests under version control. Dice machines have the following version control tools installed: RCS, CVS, SVN, GIT. You can use any of these. Following are simple steps to set up a repository using SVN in your user directory on a DICE machine,

Create Repository: Log in to a dice machine and create your repository in your user directory. To create the repository, issue the following command: $svnadmin\ create \sim /myrepository$

Create your SVN user: Now that your repository is successfully set up, you'll need to create an svn user. Simply open the svnserve.conf file in the editor of your choice:
$pico\ \sim /myrepository/conf/svnserve.conf$
and add the following:

```
anon-access = none
auth-access = write
password-db = passwd
Now you'll need to create a password file:
pico ~/myrepository/conf/passwd
Add a line in that file for your user in the format =
exampleuser = examplepassword
```

Run the svn server as daemon:
$svnserve\ -d$

Check out your repository onto your local machine. On your local machine, go to where you keep your course stuff. Let's say it is in /workspace. Then use the svn co command to check out a copy of your project.

```
cd ~/workspace
svn co svn+ssh://username@dice-hostname/path to repository/myrepository
```

Common svn commands: svn add is used to create a new file or directory on the SVN repository. Note that the file won't appear in the repository until you do an **svn commit**.

svn update or svn up. This command syncs your machine with the server. If you have made local changes, it will try and merge any changes on the server with your changes on your machine. Always run **svn update** before **svn commit**.

svn commit or svn ci. This command recursively sends your changes to the SVN server. It will commit changed files, added files, and deleted files. Note that you can commit a change to an individual file or changes to files in a specific directory path by adding the name of the file/directory to the end of the command. The -m option should always be used to pass a log message to the command.

svn delete. This does what it says! When you do an svn commit the file will be deleted from your local sand box immediately as well as from the repository after committing.

svn diff. This command can be used to find out what has changed between two revisions using:
$svn\ diff\ \ -r\ revision1 : revision2\ FILENAME.$
For example: $svn\ diff\ \ -r$ 168 : 169 *index.xml* will output a diff showing the changes between revisions 168 and 169 of index.xml.

3 Testing Notes

You must implement requirements-based tests to test your requirements from homework1. Remember, you will need specific, concrete inputs as well as concrete expected outputs . Moreover, even if the test checks only, for instance, the CCS accelerating by pressing the start acceleration button, you must still specify all the system inputs and outputs to ensure that the application under test will return the result you wish to check. Test case implementations should use descriptive names, both for the class and the methods you implement. We do not want to see test1(), runTest4(), etc. Implementation of test cases must follow provided coding standards as well. You must also measure the statement coverage achieved by these tests and include a report on the coverage level and which tests passed and failed (include this report with what you hand in). If you do not achieve at least 80% statement coverage, supplement your requirements-based tests with additional system-level tests until you reach at least this level of coverage.

4 Testing Tools

Below is a list of tools in Eclipse that will help with testing and coverage. We will not provide tech support for these. Read the documentation and ask questions of your peers on the forum.
Test Framework: JUnit
Code Coverage Tools: EclEmma for Eclipse

Running Junit. You need to add the JUnit library to the project to start using it.
- Double-click on the CCS project within Package Explorer. This will both expand and select the project.
- Select Project > Properties from the main menu. This will bring up a properties box for your project.

- Select Java Build Path and click on the Libraries tab.
- Click Add Library..., select JUnit and click Next.
- Choose JUnit Library Version as JUnit 4 and click Finish.
- Click Ok in the properties box.

If you look under Package Explorer, there should now be a JUnit 4 entry. There are a number of introductory video tutorials online you may want to look at:
`https://www.youtube.com/watch?v=v2F49zLLj-8`
`https://www.youtube.com/watch?v=QEyxgtCEWMw`
`https://www.youtube.com/watch?v=oCNMinACgAk`

For EclEmma installation, see `http://www.eclemma.org/installation.html` and follow instructions in **Option2: Installation from update site**. For user guide on use of EclEmma to measure coverage refer to `http://www.eclemma.org/userdoc/index.html` and `http://agile.csc.ncsu.edu/SEMaterials/tutorials/eclemma/`.

5 Deliverables

You are responsible for delivering the following as a single zip via the submit command:

1. Implementation of CCS, incorporating all feedback provided on the design assignment (the implementation must be located in the src/ directory)

2. Any required libraries for your implementation (in the lib/ directory)

3. A description of how to compile your code we will be using our own script, but provide any additional instructions that are necessary for code compilation.

4. Requirements-based tests in JUnit.

5. Report on these tests, including a list of which tests passed and which failed (a single-PDF JUnit report is acceptable here).

6. Statement coverage report from these tests (EclEmma provides a nice report format). Additional tests (in both formats) to achieve, at minimum, 80% statement coverage.

7. A traceability matrix to show which test cases verify which requirements from your first homework (requirements may have been updated since based on feedback and design decisions, in which case show the updated requirements).

	Test ID1	Test ID2	Test ID3	Test ID4
Req.1	X			
Req.2		X	X	
Req.3	X			X

8. Explain how you have **addressed the feedback** (for improvement if any) provided by markers on your design. Updated versions of previous documents, if changes are made (requirements and design).

9. Include **a title page with names and UUNs of team members**.
10. You should also submit **a text file named team.txt** with only the UUNs of the team members (one UUN on each line) as shown,
    s1234567
    s7891234

How to Submit. On the School of Informatics DiCE computer system, if your project is in a folder called *Application* (with the source code, tests and team.txt files) then you should submit it for Inf2C-SE **homework3** using the command:

```
submit inf2c-se 3 Application
```

6 Due Date

Homework 3 is due

Thursday, November 18th at Noon.

This homework is worth 50% of the total coursework.

Appendix: Coding Standards

Below is a list of coding conventions that are adopted from Apache Commons Net (`http://commons.apache.org/net/code-standards.html`). Everything else not specifically mentioned here should follow the official Sun Java Coding Conventions (`http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`).

1. Variables and Class/Interface/Enum names should use CamelCase with variable names starting with a lower case letter and Class/Interface/Enum names starting with an upper case letter. Moreover names should be easily readable, using long names over abbreviations.
2. Brackets should begin on the same line as the opening code, and end on a new line. They should exist even for one line statements. Examples:

```
if ( foo ){
  // code here
}

try{
  // code here
}catch (Exception bar){
  // code here
}finally{
  // code here
}
```

```
  while ( true ){
    // code here
  }
```

3. Though it's considered okay to include spaces inside parens, the preference is to not include them. Both of the following are okay:

```
  if (foo)

or

  if ( foo )
```

4. 4 space indent.**NO tabs**. Period. We understand that many developers like to use tabs, but the fact of the matter is that in a distributed development environment where diffs are sent to the mailing lists by both developers and the version control system (which sends commit log messages), the use of tabs makes it impossible to preserve legibility.

5. Descriptive JavaDoc comments **MUST** exist for all methods and classes. JavaDocs on data members is preferred and encouraged, but a standard comment (called an implementation comment) describing the data member on the line before it is acceptable here. For more information on how to write JavaDocs, please see: `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`

6. Blocks of code should have inline implementation comments to help with finding code quickly during maintenance. These should be there in addition to the JavaDoc comments above the method/class/member. For example:

```
/**
 * JavaDoc description here.
 * @param param1  describe param1 here.
 */
 public void myMethod(String param1)
 {
   //Process parameter
   ...

   //Do something else non-trivial
   ...

   //Do yet another non-trivial thing
   ...
 }
```

   It is generally bad practice to include these comments as trailing comments as it makes the code harder to read.

7. Import statements must be fully qualified for clarity.

```
   import java.util.ArrayList;
   import java.util.Hashtable;

   import org.apache.foo.Bar;
   import org.apache.bar.Foo;
```

And **not**

```
   import java.util.*;
   import org.apache.foo.*;
   import org.apache.bar.*;
```