Vrije Universiteit Brussel
Pleinlaan 2
Software Languages Lab - Dept. Of Computer Science Master Studies

Capita Selecta: Software Engineering

Project Software Engineering

Deliverable 1

Maarten Vandercammen **Christophe De Troyer**

DATE: *2nd March 2015*

DUE : *1st March 2015*

CONTENTS

1 Project Description

The project we (i.e., Maarten Vandercammen and Christophe De Troyer) are using for this course is named “CodeBox”. CodeBox is a project built by Christophe De Troyer during his studies for a Bsc at Ghent College. It is a web application built using the Microsoft .NET web stack. It uses MVC 3 as architectural pattern, Entity Framework for the persistence layer and C# as the language of choice. The views are constructed using the then-new markup language Razor.

1.1 Functionality

Snippets Codebox is basicly a clone of pastebin.net (or any other text-sharing website). The main idea of the website is to allow users to upload snippets of code using the webinterface and share them with other users. If wanted, the snippets can be kept private as well. Snippets can be given a title and description for future reference.

Groups To differentiate a bit from pastebin.net the notion of groups was introduced. This allows a user to create a group and add other members of CodeBox to that group. This allows one to share snippets not only with everyone (i.e., public) or nobody (i.e., private) but with groups as well. This allows every member of a group to view snippets shared in that specific group. Users are however not allowed to edit snippets created by a different user in that group.
Finally, groups have some sort of profile. Administrators of a group can modify the name, description and profile picure of a group. They can also invite new members to a group by adding their e-mail address to the list. This e-mail has to be a valid and existing user for the system to work.

Users The last small feature of CodeBox is that users can edit their profile. It is possible to provide a full name (first and surname). The user can change his/her password and upload a profile picture.

2 Testability

The testability of this project is rather high. The main reason for this is that the project is built strictly according to the MVC pattern. This has a few key advantages when it comes to unit testing the application. First of all, a proper MVC application is loosly coupled. By definition, the MVC pattern decouples the application logic, display logic and persistence logic which is in sharp contrast to regular page rendering systems (E.g., ASP.Net Web Forms).
We will discuss each important component and technique to show that there is indeed high testability in this particular application.

2.1 Ninject

Ninject is a dependency injection framework for .Net. Using dependency injection it is possible to decouple the controllers from the actual data layer implementation. In Listing 1 a small snippet shows how the `SnippetController` class is decoupled from actual implementation details of the repositories. In the context of testing this enables us to mock up a test repository with, for example, hardcoded values. Unit tests will not be aware of this and thus totally oblivious of the fact that it is working with fake data.

```
public class SnippetController : Controller
{
    private ISnippetRepository snipRepo;
    private ILanguageRepository langRepo;
    private IUserRepository userRepo;
    private IGroupRepository groupRepo;

    ...
}
```

Listing 1: Example of decoupling controller from data layer

2.2 MOQ

As mentioned before, we use “mocking” implementations of data repositories to unit test controllers. This is all well enough for small tests. However, this approach does not scale properly. If you were to stest the number of invocations of a certain controller method one would simply add a field in the `FakeRepository.cs` class and increment it each time the fake method is called. This approach is problematic if the fake repositories grow too large to maintain. One might even say that it would need unit testing. To mitigate this, there is a thing called `Moq`. Moq is a library that allows one to easily create fake repositories. An example of creating a fake repository is given in Listing 2.
In this example we ask Moq to give us a class that implements the `ISnippetRepository` interface. Moq then generates this class on the fly. To specify behaviour for each method we can use some LINQ magic. The last line in the example specifies that the method `m.GetSnippets()` should simply return the array of `Snippets`.
This way, we can easily generate test repositories. It is fairly obvious that this approach is much less error prone that manually writing all the test repositories.

```
Mock<ISnippetRepository> mock = new
    Mock<ISnippetRepository>();

Snippet[] snippets = new Snippet[]
        {
        new Snippet() { Name = ''SSL Decryption'',
            ''User''=''cdetroye'',..},
        new Snippet() { Name = ''Windows Backdoor'',
            ''User''=''mvdcamme'',..},
        new Snippet() { Name = ''GPG Bruteforce algo'',
            ''User''=''cdetroye'',..},
        new Snippet() { Name = ''iCloud Hack'',
            ''User''=''mvdcamme'',..}
        };

mock.Setup(m => m.GetSnippets()).Returns(snippets);
```

Listing 2: Creating a mocking repository using Moq.

2.3 CONTROLLERS

The key component of an MVC application is the controller. In this case testing it is a breeze. Testing a controller is as simple as mocking up the above mentioned repository for data. The unit test can then test if the returned model is correct by means of testing the data that it contains. This does mean that we can not test the

actual representation. But this is rather an up- than downside. An example of how to test a controller is given in Listing 3. Note that it assumes the code of Listing 1 and 2.

```
SnippetController c = new SnippetController(mock.Object);
IEnumberable<Snippet> result =
    (IEnumberable<Snippet>)c.List(2).Model;

// Assertions
Snippet[] ss = result.ToArray();
Assert.IsTrue(ss.Length == 4);
Assert.AreEqual(ss[0].User, ''cdetroye'');
Assert.AreEqual(ss[1].Name, ''Windows Backdoor'');
```

Listing 3: Testing the model of a controller using Moq and Ninject.

2.4 Models

Unit testing a model is rather trivial. In a strict MVC application these classes do not contain any logic. They are merely carries of data. Thus, unit testing them would be as simple as writing some tests for a getter and a setter.

2.5 Repositories

Testing the repositories is also fairly straightforward. To do this we should test the logic of the actual repositories. This can only be verified by actually running data insertions, deletions and retrievals. Again, these tests should prove to fairly straightforward. However, that might depend on the complexity of the actual persistency layer. In our case this is entity framework that is hooked up to a plain old SQL Server instance. This means that we keep raw SQL queries at a distance and just program against the Entity Framework model. Thus, relying on Entity Framework we can assume that testing beyond the interface of Entity Framework is not required.

2.6 HTML

The final component of our application is the actual view. To test this we have te resort to technology such as Selenium. Selenium is a framework that is built for testing HTML user interfaces. Again, we can test each page seperatly.

2.7 Conclusion

We can conclude that by using all the frameworks and tools at hand it should be possible to have the highest possible test coverage. We have the MVC pattern to thank for this, as well as the top notch libraries like Moq and Ninject.

3 Test Strategy

3.1 Overall Test Strategy

We plan on employing a wide array of different testing strategies, to maximize the quality of our delivered software, in accordance with the principles of agile programming. Testing will not happen ad-hoc, but will be structured and integrated directly into the core of our development process. In order to maximize the number of tests

that can take place, we will mainly focus on creating tests that can be run and verified automatically, i.e., unit-tests. However, we will also dedicate ourselves in part to manually explore our application, to validate the overall quality of our software. This extends not only to finding and solving any bugs that may be present, but also to making sure that the runtime performance of our application is sufficiently high to remain responsive and that our graphical user interface is intuitive for new users.
Since it is in practice impossible to deliver an application completely free of any errors, we will have to divide our time and choose which aspects of our software will receive the most focus. We have therefore decided that the time spent on a feature directly corresponds with the priority level of that feature: high-priority features will receive the most attention, low-priority features the least.
Concretely, we plan on using the following testing strategies.

UNIT TESTING

Unit-tests will form the backbone of our testing strategy. Each developper will create unit-tests simultaneously with implementing the various features. If the developper has finished writing the unit-test, the test will be run. If an error is reported, the error should be fixed immediately. Additionally, all unit-tests that have been created will be run overnight and any errors should be solved the following day.

FEATURE TESTING

In each iteration of our development process, we will release a number of new features in our application. Once a feature has been completed, we will perform manual testing on this feature to make sure it lives up to the expected standards with regards to errors, runtime performance and a qualitive GUI. Since we are developping a web-application, we will rely on the Selenium framework[1] to automate our feature tests as much as possible.

SYSTEM TESTING

At the end of each iteration, we will execute manual, system-wide tests. These tests will mostly be focused on verifying and validating the newly implemented features, but they will also cover the existing features, in order to make sure that the whole application lives up to the standards we have set for it. While testing these existing features, we will especially focus on the high-priority features.

INTEGRATION TESTING

After completing or updating a module in our software, we will create automatic integration tests to verify whether this module correctly interacts with the other, already existing modules. Specific attention will be payed for mission-critical modules, or modules specifically created to implement high-priority features.

3.2 TEST COVERAGE

Below, we describe how we will maximize the test coverage for each of our user stories. We give an overview of the various tests we aim to create for these stories. Note that this set is only the minimum set of tests we will complete. It is likely that, as we progress, we will uncover additional opportunities for testing.
We start by examining the highest-priority features and progressively move on to the lower-priority features.

[1] http://docs.seleniumhq.org/

LOG IN

The most important step to be tested in the 'Log in' user story is the validation of the log-in details, i.e., the e-mail address and the password. It should be possible to fully automate these kinds of tests, by storing dummy user-data in the database and checking whether the validation process succeeds when entering the correct details, and fails when entering incorrect data. The other steps in this user story can be verified with the Selenium framework.

REGISTER

The most important task here is again the validation process. This time however, validation is more complex, since the application needs to make sure that no user with the given e-mail address has already been registered, that the password is strong enough and that the string that is provided as the given e-mail address can indeed be parsed as an e-mail address, for example by making use of a regular expressions matcher. All of these tasks can be fully automated, again by providing dummy data to the database. For the last task, we should analyze the regexp that will be used and determine where the corner cases of the expression lie.

LOG OUT

For this user-story, we will rely almost exclusively on the Selenium framework.

CREATE NEW SNIPPET

For this story, we will focus mainly on the step where the user-input, i.e., the actual snippet along with all meta-data such as its name and id, are saved to the database. To this end, we will automatically generate snippets, store them in the database and retrieve them immediately afterwards. If all data can be retrieved, the test has succeeded.

SHOW SNIPPET

As with the other user stories, we will heavily rely on the Selenium framework to test the input/output actions. However, when viewing a snippet, we should make sure that syntax highlighting is correctly applied on the code that is shown. Because this syntax highlighting depends on the input language that was chosen, as well as the code that was entered, testing the correctness of this step will be an arduous task. At the moment, we do not know of any technique which would allow us to automatically verify whether syntax highlighting has been correctly applied given a specific input program and programming language. We will therefore have to rely on manual testing for this feature.
Testing this user story would go well hand-in-hand with the 'Create new snippet' story: we could test the creation of one snippet and immediately afterwards check whether this snipper is correctly displayed. However, we do not wish to completely rely on this technique, since we would then have to wait until both features are fully implemented before we can start testing them. Furthermore, if one feature were to fail, we would have delay testing the other feature until the issue was fixed.

EDIT USER PROFILE

Testing for this user story will be similar to the testing of the 'Register' story. We will again automatically generate mock data, use it to edit user profiles that also have

been automatically generated, and check whether the correct changes have been made in the database.

DELETE SNIPPET

It should be fairly straighforward to test this user story. We will again automatically generate snippets and store them in the database. Afterwards, we delete them and verify whether they have indeed been correctly removed.

ADD PROFILE PICTURE

CREATE NEW GROUP

For this story map, the most important part to test is the saving of the details of the new group. This can be done in a similar matter as the other features where data is automatically generated, stored into and then retrieved from the database. The other aspects of this feature can be tested by using the Selenium framework.

SHOW GROUPS FOR USER

Since this feature should consist of no more than a simple database fetch, followed by showing the retrieved data, which can be tested through the Selenium framework, it should be trivial to create automatic tests for this feature.

INVITE USER

As with the previous story map, it should be fairly straightforward to write automatic tests for this feature, since it mainly consists of reading input and showing output, for which we can use the Selenium framework.