Testing

Testing is a complex subject that sometimes is overlooked during the development of a software application, but is crucial to its success. In this document I want to do a quick introduction to testing of applications, so users with less experience in the subject can understand the aims and benefits as well as how we achieve those aims. Please feel free to add any correction or extra information you feel relevant. Also, my native language is not English, so I apologise if you find some parts of the text hard to understand. Feel free to advise me a correction.

You can also find a more detailed reference in Wikipedia [http://en.wikipedia.org/wiki/Software_testing]

Please note in this document when referring to DSpace I'm talking about the vanilla DSpace code. Customisations in the code may have an impact on what's described here.

What's Testing

The primary purpose for testing is to detect software failures so that we can fix them. Simple as it sounds, it's a non-trivial task as any amount of testing can't guarantee a software doesn't fail, only that it fails under a set of conditions or (when we fix the issue) that it works under that set of conditions.

As you can deduce from the previous definition, testing is not only a matter of code, but also of the environment in which this code is run. That means that testing includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do.

Types of Testing

Testing has many approaches that try to cover different aspects of the application, to reduce the possibility of a user finding an unknown bug as much as possible. Some of these approaches are:

- **Functional Tests:** these are tests which come from user-based use cases. Such as a user wants to search DSpace to find material and download a pdf. Or something more complex like a user wants to submit their thesis to DSpace and follow it through the approval process. These are stories about how users would preform tasks and cover a wide array of components within Dspace.
- **Integration Tests:** these tests work at the API level and test the interaction of components within the system. Some examples, placing an item into a collection, or creating a new metadata schema and adding some fields. Primarily these tests operate at the API level ignoring the interface components above it.
- **Unit Tests:** these are tests which test just how one object works. Typically test each method on an object for expected output in several situations. They are executed exclusively at the API level.
- **Regression Testing:** these are tests which focus on finding defects after a major code change. The aim is to detect if old bugs have resurfaced due to the changes, or if those changes have broken another component of the system. It's usually covered with the tests created for Unit Testing, Integration Testing and Functional Testing.

- **Performance Tests:** these are tests which test how the application behaves under heavy load, and how many users it can manage at a time. While the other types of tests mentioned are platform-independent, these tests depend on the hardware they are run and on the settings of components external to the application itself, like the database. They usually reproduce functional tests to simulate "real load" in the application.
- **Manual Testing:** these are tests run by humans on the application. Although slower and probably not so extensive, they can detect issues that other test may fail to recognise, like problems with the layout, graphic glitches or similar.

How to Work with Tests

Ideally all tests, when possible, have to be run automatically and after every code change, so no code that doesn't pass the existing set of tests is committed to the repository. Also ideally you should do Test Driven Development (explained later). To this aim, tests should be run in the compile cycle, that way the developer doesn't need to remember to run them. The downside of this approach is longer build cycles as the test are run, but the benefits of running the tests are worth this delay.

To compensate for this a good alternative is to build a Continuous Integration system (like Hudson) where the tests are run constantly. That way the developers only need to run the tests before submitting the changes, which saves time in the compilation process, and Hudson will do an extra validation to ensure the commits don't break the build.

All tests require extensive maintenance. Every time a new bug is discovered, a test has to be created to reproduce it before solving it. This serves two purposes: on one hand, we ensure we understand the bug and why it happens. On the other hand, we add this test to the existing tests, which enables regression testing and (in the long term) increases the stability and robustness of the application.

How Hard is to Create Tests

Creating tests is not too complex in general, although for certain types of test some technical knowledge is required.

The simplest tests to create are probably (and depending on our approach to them) Functional Tests. Most of those tests are based on UI interaction and tools like Selenium allows us to record the actions we are doing on the page to reproduce them later. This lets a non-technical user, with a minimal knowledge of the tools, to create a set of tests for his or her repository.

Integration Tests and Unit Tests require someone with a deeper understanding of DSpace API as they work directly with it. That means only developers will be able to create these kind of tests. One issue we may find when creating Unit Tests is that the code is not testable. In this case, a refactoring of the code may be advised, as lack of testability may point to design or coding issues.

Performance Tests are probably the hardest to create, as we have to account for several variables (hardware, settings) besides running the testing itself, and after that we have to understand and interpret the data the correct way.

Testability

We understand Testability as a set of practices that makes the creation of tests (mainly Unit and Integration Tests) easier, and at the same time they are generally good practices when coding.

A Testable code, in general, uses injection of dependencies in the constructor, avoids global state (static variables) and follows the Law of Demeter. Following this simple principles our code quality will improve and we will reduce the effort required to create tests.

The Law of Demeter is a guideline for software development that promotes loose coupling. It can be stated as "use only one dot", referring to the fact that if we need access to an object we should get the object in the constructor, not through a method call to another object. That is:

> *b.getValue() is better than a.getB().getValue()*

The main reason is that the second form hides some dependencies of the current object, which makes the code less maintainable and complicates the development of tests as we may not be aware of that dependency.

Google has published a tool, Testability Explorer, to determine how testable is an application. You can find it at http://code.google.com/p/testability-explorer/

Test Driven Development

Test-driven development (TDD) is a development technique that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to fit the application.

The advantage of this procedure is that we develop the tests (mainly Unit Tests and Integration Tests, but also Functional Tests when required) as we develop the code. Having to create the test first makes us think about the problem deeper, ensuring we are aware of all the requirements. In short, this leads to simpler code which will result easier to maintain and that already has tests to validate it.

You can find more details at Wikipedia [http://en.wikipedia.org/wiki/Test-driven_development]

Thanks

This page has been created with help from Stuart Lewis, Scott Phillips and Gareth Waller. I want to thank them all for their comments. Some information has been taken from Wikipedia to make the text more complete. I'm to blame for errors in the text.

Feel free to contribute to this page!