Hasso Plattner Institut
IT Systems Engineering
Universität Potsdam

Bachelor's Thesis

Test-Driven Web Development
— A Case Study With Django —

by
Philipp Giese

Potsdam, June 2010

Supervisors

Prof. Dr. Christoph Meinel

Martin Wolf, M.Sc.

Internet-Technologies and Systems Group

Disclaimer

I certify that the material contained in this dissertation is my own work and does not contain significant portions of unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation.

Hiermit versichere ich, dass diese Arbeit selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, June 25, 2010

_____

(Philipp Giese)

Kurzfassung

In dieser Arbeit werde ich als erstes in das Thema der Testgetriebenen Entwicklung einführen und dabei die Vorteile dieses Ansatzes Software zu schreiben hervorheben. Im Anschluss stelle ich die Testumgebung von Django vor und bewerte diese, indem ich darlege, wie sie uns in der Entwicklung des Bachelorprojekts "Sendinel" unterstützt hat. Zum Schluss führe ich in die Testumgebung Selenium ein und zeige, wie diese genutzt werden kann, um die Teile von Web-Applikationen zu testen, die von der Django Testumgebung nicht abgedeckt werden.

Abstract

This thesis will first give a brief introduction to Test-Driven Development highlighting the advantages of that approach in writing software. Afterwards I will present the Django test suite and evaluate it by pointing out how we used it during the development of our bachelor project called "Sendinel". At the end I will introduce the Selenium test framework showing how it can be used to test the parts of a web-application that are not covered by the Django test suite.

Contents

1 Introduction

Modern software development more and more moves away from standard desktop applications as they are hard to maintain and the developers have to struggle with being compatible to a variety of operating systems. The internet has opened up new possibilities for application developers. With powerful server computers and highly interactive front-end technologies as JavaScript, CSS and HTML5 it is now possible to build web-applications with the look and feel of a desktop application. It also enables developers to update their software more often, as the end-user does not need to struggle with the installation of updates. Once installed on the server system they are immediately available to all users.

But because it is now possible to have large systems in form of a web-application, it becomes even more important to have these systems available all the time. Automated testing and Test-Driven Development helps developers to ensure that their software works under (almost) all conditions.

1.1 What does Test-Driven Development mean

Test-Driven Development is an approach in writing software, where the developer does not write any line of code until a test for this particular piece of code exists. To most developers, that are new to this methodology, this appears paradox at first glance. How do you write tests for code that is not written yet? But it is this paradox that opens up new ways. If you write tests for your code before you implement it, you have to start to think about what you want to achieve before you struggle with the implementation. You also start "working" with your code, meaning that when you write tests, you imagine the perfect interface for the operations you want to implement and start telling yourself a story about how the operation will work from the outside. Maybe not all of the stories will become true in the end, but here you have the chance to start with the best-possible application program interface (API), than to make things complicated right from the start. [1]

1.2 Contributions

There are a lot of different testing frameworks available. Many of them are collectively known as xUnit. Below you find three representatives for other testing frameworks. All these frameworks resemble in design and code structure. You can define rules for your code using assertions and in some way group and organise your test cases through your project and run either all or particular ones.

- SUnit for Smalltalk[1]
- JUnit for Java[2]
- RSpec for Ruby on Rails[3]

[1] `http://sunit.sourceforge.net/`
[2] `http://www.junit.org/`
[3] `http://rspec.info/`

In the following sections I will first give a theoretical overview of Test-Driven Development, pointing out what Test-Driven Development means and how you can build applications using this methodology. I will also give some aid for those who are new to Test-Driven Development showing some development patterns that help you to write tests and deriving the correct application code of them.

Subsequently I will shortly introduce Sendinel – the system we developed – and afterwards describe the test framework provided by Django, pointing out how we used it during the development.

In the last part you will find a section about the Selenium test framework which can be used as an add-on to the Django test suite to better test the front-end of web-applications.

2 Test-Driven Development

2.1 The word "Test"

test [verb] "Take measures to check the quality, performance, or reliability of (something), esp. before putting it into widespread use or practice."

If we map this definition to the work of a software developer we can see some analogies. Software developers also need to evaluate the code they have written before they can release a piece of software. This is not how Test-Driven Development works. Testing the changes you have made to your code is not the same, as having automated tests. This way you might not have thought about what goals you want to achieve before you implement new features.

Test [noun] "A procedure leading to acceptance or rejection."

This definition more likely describes what the essence of Test-Driven Development is. It is a procedure, where you first think about what you want to do, then figure out the tests that assure, that the code works the way it is intended to work and finally write the real implementation.

So why does the word feel different when we are using it as a verb or as a noun. The psychological part is different. If you test your software after you have implemented something errors produce stress. Now you have to have a look on everything you did in the past minutes or hours and search for the error. If this happens more often, developers tend to test less to reduce their personal stress level. Automated tests help to lower this level. Defining tests first gives developers a safety net. Whenever a developer reaches a situation where he asks himself whether he has broken something, he can run the tests. If all tests pass he can be sure that everything works fine. This reduces stress, thus the developer feels better and that encourages him to test more often.

2.2 The Development Cycle

When you develop test-driven you will go through five stages as shown in Figure 1. The first step is to write a small and above all atomic test. After implementing the test you run the test runner and see the newly added test fail. Why do that? If, for example, you are working in a team, it may happen that certain functionality has already been implemented by someone else or even by yourself. So if you run the test runner on a newly added test and see the test pass than you have to step into the code and search for a reason why. The most common reasons are the following two:

- The functionality has already been implemented
- The test case is incorrect and needs to be refactored

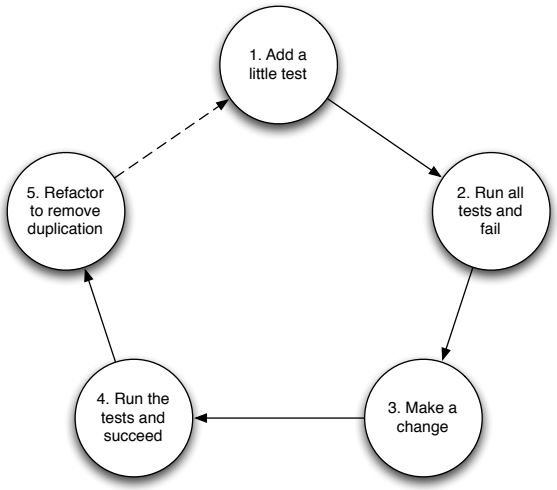

Figure 1: The development cycle of a Test-Driven Development process [1]

After you have seen your test fail you implement the code as simple as possible to get the test pass. In Section 2.4 you find some patterns that help to figure out how to implement the code in the right way. For example, if you expect a function to return the number five, than implement the function to return the constant 5 (also see Section 2.4.1).

At this point one could wonder how we will derive working code from that development cycle, if we only fake implementations so that our test cases do not fail. This is where the last stage of the cycle comes into play. We now have to *refactor to remove duplication*. In our example we have a duplication of the number five in our test case and the actual code. We can remove the duplication by adding a parameter to the function and returning that parameter. The correct way, of course, would have been to first add the parameter in the test case and afterwards in the implementation.

If you have removed all duplicate code and the test still passes, you have reached a point where you can tell yourself that you are done. This is another advantage of Test-Driven Development. You know when you are done and when you are not. If you have tests that

fail you need to work to get them pass and if you have duplication between the test cases or within your implementation you have to refactor to remove the duplication. Having gone through all five stages you can now start over from the beginning by adding a new test for the next thing that is on your todo-list.

2.3 How to write tests

When you write tests in most cases the hardest thing is the start. So having the whole system in mind we ask ourselves the following questions [1]:

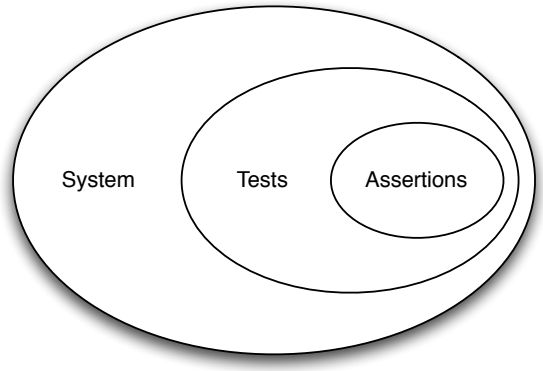

Figure 2: Asserts are the smallest component when writing tests for a software system

- Where should you start building a system? With stories you want to be able to tell about the finished system.
- Where should you start writing a bit of functionality? With the tests you want to pass with the finished system.
- Where should you start writing a test? With the asserts that will pass when it is done.

Figure 2 shows that assertions are the smallest, but yet most important component when writing tests. Without them a test would not actually test anything and thus would not make any sense at all. And without tests you would have a system on which you could not rely.

With the assertions you write you define the way you want your system to behave and react to certain inputs. They also help you to write the tests. Kent Beck uses a very good example on how to write the tests "backward".

Given we want to communicate with another system over a socket. When we are done, the socket should be closed and we should have read the string hello.

```
1 public class exampleTest extends TestCase {
2     public void testCompleteTransaction() {
3         assertTrue(reader.isClosed());
4         assertEquals("hello", reply.contents());
5     }
6 }
```

Having these assertions that clearly define what we expect, we can now write the rest of the code for the test. So where does the reply come from? Clearly from the socket. And the socket? A socket is created by connecting to a server. To connect to a server we have to open one. So what we could write now is:

```
1  public class exampleTest extends TestCase {
2      public void testCompleteTransaction() {
3          Server writer = Server(defaultPort(), "hello");
4          Socket reader = Socket("localhost", defaultPort());
5          Buffer reply  = reader.contents();
6
7          assertTrue(reader.isClosed());
8          assertEquals("hello", reply.contents());
9      }
10 }
```

Knowing what we want to achieve we could perform tiny logic steps backward that helped us figuring out the right code for the test case.

2.4 Development Patterns

In Section 2.2 the development cycle clearly states that after you wrote a new test and have seen it fail, you have to implement your code as simple as possible to get it running. But how do you do this the best way? In this section I will present three methods that can be used to write code to get a test pass.

2.4.1 Fake it! ('til you make it)

The easiest way to get a test running is to fake all the objects and the code structure you need. But why do so? Why would we write code that we know will be replaced later on? There are some reasons. First of all the fake implementation tells us, if we made any mistakes writing the test case itself. If the test fails we have to look at it again. Are the assertions really the ones we wanted to write? If not, we can fix them and do not have to be annoyed because we spent a lot of time with the real implementation, just to see that the test fails. Because then we would more likely search for the error in our implementation than in the test case itself.

Given this example we have two main effects why *Fake It* is powerful. The first one is the *psychological* effect. Seeing a green bar feels different than seeing a red bar. Given the green bar we know were we stand. We have the test running and can from there refactor with confidence. The second one is *scope control*. The fake implementation helps to focus on one topic at a time. If you would start with the real implementation right from the start you may get distracted by what you code. Having the fake implementation you can write the real code step-by-step while still knowing that you do not break any of the other tests.

2.4.2 Obvious Implementation

Sometimes you write a test case for a piece of code and already have the implementation details in mind. If you know what you type and the implementation is simple then just implement it. But you have to be aware that when using *Obvious Implementation* you are "demanding perfection of yourself" [1].

If a test now fails this is a sign to shift down and use the common red/ green/ refactor development cycle.

2.4.3 Triangulation

"Triangulation is the process of determining the location of a point by measuring angles to it from known points at either end of a fixed baseline, rather than measuring distances to the point directly." [2]

When writing test cases triangulation means that it is not enough just to check if one specific input produces one specific output. For example, if we implement a function *times* and we have a test that checks whether `5.times(2) == 10` a fake implementation of times that returns the constant 10 would be enough to get the test pass. But when we add an assertion that checks whether `5.times(3) == 15` returning the constant 10 would no longer suffice.

But there is a little hitch with triangulation. Figure 3 shows how triangulation can lead to an infinite loop. Once we have abstracted the correct implementation for times we can delete one of the two assertions, because they are completely redundant. Now it would be enough to return a constant number to pass the test and that requires us to add an assertion and so on.

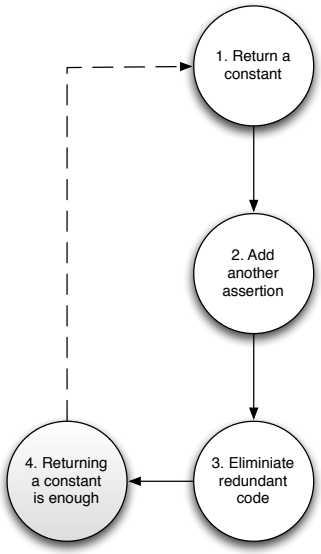

Figure 3: The rules of triangulation create an infinite loop

3 A Case Study

3.1 The Sendinel-Project

During the bachelor project, which is part of the graduation process of our bachelor studies we developed a software called "Sendinel". Sendinel aims at improving the communication between clinics and patients in rural africa. It accomplishes that via sending text messages and automated telephone calls to patients. This way the clinic staff is able to inform the patients about for example vaccinations or when their lab results have arrived.

We developed Sendinel as a group of seven students at the chair of Internet Technologies and Systems of the Hasso-Plattner-Institut in Potsdam[4], Germany. The project partners were SES

[4] `http://www.hpi-web.de`

Astra[5], SAP Research South Africa[6] and the University of Capetown[7]. During the time of the project we were able to deploy Sendinel in a clinic in rural South Africa [3].

The system itself is implemented as a web-application using the Python-based web-framework Django[8].

We used the approach of Test-Driven Development to build Sendinel but could only manage to have the backend completely tested (see Section 4.5 for more information). The following sections will give an overview about what is possible with the test suite of Django and how you can develop applications tests-first. At the end of each category I will point out how we applied the shown techniques while developing Sendinel and which challenges we had to face.

3.2 Choosing the right Framework

During the planning phase of the project we had to choose the technology we want to use to develop our system. In order to do that we came up with four main ideas.

Java As we knew our partner SAP Research in South Africa mainly uses Java as a programming language for their applications, it might be useful to use this language to make it easy for SAP to adopt the software afterwards. But as we wanted to learn something new and everyone of the team had already written something in Java we decided against it.

.NET We also considered .NET but as we did not want to focus only on machines running Microsoft Windows it became clear that we had to choose another language.

Ruby on Rails During the semester in that we started the bachelor project, we had a course where we had to develop a web-application using Ruby on Rails. Therefor the idea emerged that we could simply use Ruby on Rails for Sendinel. But the experiences we had made were not that positive, so we also dropped this idea.

Django Some of the team members had already developed applications using Django and the others were keen on getting to know it as it becomes more and more popular in the area of web-development. Django also provides a test suite suitable for Test-Driven Development. We also thought that developers might get interested in the project and are more willing to continue our work if we utilise a young framework.

[5] http://www.ses-astra.com
[6] http://www.sap.com/about/company/research/centers/pretoria.epx
[7] http://www.uct.ac.za/
[8] http://www.djangoproject.com/

4 Test-Driven Development with Django

4.1 The Django Web-Framework

The Django web-framework has its origin at The World Company of Lawrence, Kansas. In 2005 the developers of the company created a Python-based web-framework to help them rapidly build web-applications for their news websites. This framework than was released under a BSD license and is now better known under the name Django [4].

4.1.1 The MVC-Pattern

The Model-View-Controller (MVC) design pattern separates the code into three distinct parts as shown in Figure 4.

Model Stores all the data and the application logic

View Renders the interface

Controller Translates user inputs into updates that are sent to the model

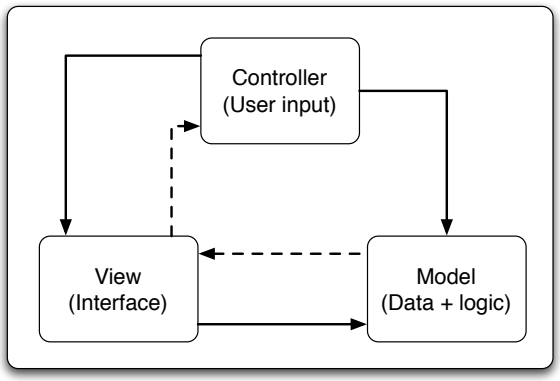

Figure 4: The basic structure of the Model-View-Controller design pattern

The basic principle of the MVC-pattern is to separate responsibilities. A model class only concerns itself with the application's logic and data. It does not display any of the data to the user. By contrast the view class only concerns itself with how the data is displayed in the user interface. Finally, the controller class is only concerned with translating user input that is received through a view into updates that it passes to the model. [5]

Separating the code into these three parts yields some benefits:

- Developers can focus on one topic at a time while writing the code
- Parts of the code can be reused more easily (e.g. one view might be used with different models)

- User interfaces (views) can be added, removed or changed at runtime and compile time
- Multiple representations (views) of the same information (model)

Responsibilities of the model The model stores the application data in properties and provides methods to set and retrieve that data. The methods for data-manipulation are **not** generic, they are specific for each application and must be known to the controller and the view. Thus controllers in MVC are custom written to manipulate a specific model.

Responsibilities of the view Views must provide the user interface and keep it up-to-date. A view *never* changes a model but it can retrieve data from it. The view listens for changes in the model and displays them when they occur. A view itself does not evaluate inputs from the user but forwards them to a corresponding controller which then decides what to do with the data.

Responsibilities of the controller The controller reacts to notifications from the view. It translates the user input it gets from the view and translates it into updates for the model. Sometimes the controller also makes logical decisions about the input before making a change to the model.

4.1.2 How Django implements MVC

When you create a new application with Django there are some files that have been automatically created for you. For this section primarily the files `models.py` and the `views.py` are of interest to us. Although the name `views.py` is a bit distracting because it is more of a file for the controllers than for the views.

Following the principle of *fat model – skinny controller* most of the business logic and the data handling goes into the `models.py`. The object-relational mapper of Django translates the fields of the model classes into fields of the underlying database. Each controller can be accessed via a url that has to be specified in the `urls.py`. But only having these files we are not getting any user interface for a application. This is down by creating so called *templates* that correspond to the views of the MVC-pattern. But as the templates are delivered by a web server to client's computers these views cannot listen for changes of the models themselves. They are snapshots of the state of the model at the time when the view was rendered by the controller. All local variables declared in the controller are handed to the view and are not linked to the fields of the model. To update a view a page needs to be reloaded or other actions like an AJAX request to the server have to take place in order to fetch updated data.

Although the template files are meant to be plain HTML, Django provides a technique to implement a certain logic into them. This is done via so-called template tags [6]. These tags provide logic structures such as if-else-clauses and convenience wrappers for functions that are often used when displaying data (e.g. `cycle`[9] to dye rows of a table in alternating colors).

[9] `http://docs.djangoproject.com/en/1.1/ref/templates/builtins/#cycle`

When developing test-driven the separation into model, view and controller brings a great advantage. You can also separate your test cases. You do not have to struggle with testing a mash-up of model-, view- and controller-code, you can write different test cases focussing on one topic at a time. This also eases the debugging process if a test fails. You exactly know in which part of your application you have to look for the error.

4.2 Docstring Testing – using *doctest*

When you create a new Django application a file named `tests.py` is automatically created for you. In this file you will find two examples of how to write tests. As in many other frameworks you have the possibility to write unit tests (see Section 4.3) but Django also gives you the ability to write your test cases as so called *doctests*. In the following sections these doctests will be described and discussed in detail.

4.2.1 How testing with *doctest* works

When writing Python programs you can add a so called docstring to every method you implement. These can then be used to help other developers. In a shell session a developer could enter `help(method_name)` and the docstring would then be displayed to help him use this method.

Doctests can be placed either directly into the docstring of a method or in the `tests.py` file. Often doctests are a direct result of a Python shell session. You give a set of commands that shall be executed and note down the expected results.

In a doctest each line starting with >>> will be executed by the Python interpreter (excluding the >>>). This is the example that is automatically provided by Django.

```
1 def exampleMethd(self):
2     """
3     Tests that 1 + 1 always equals 2
4
5     >>> 1 + 1 == 2
6     True
7     """
```

When parsing the docstring the Python interpreter will find the >>> and evaluates the following statement. After that it compares the produced output to the expected one from the test case. If the produced output does not match the expected one the test case will fail.

One advantage of doctests clearly stands out. You have your tests right where your code is. This way you are able to identify untested methods by looking at their docstrings. But there are also some people who would see a disadvantage in that. Because now you have all your tests widely spread through the whole project. If you rather like to have all tests in one place you can also put them into the `tests.py` file.

In the file `tests.py` you will find a variable called `__test__`. This is a dictionary where you can put the doctests as follows:

```
__test__ = {"doctest": """
Another way to test that 1 + 1 is equal to 2.

>>> 1 + 1 == 2
True
"""}
```

Note that there can be more values inside the dictionary. You can use meaningful names for your keys to group and organise your tests inside the `__test__` variable.

4.2.2 Problems with dependence

If you are using doctests you have to be aware of some disadvantages that come along with them. Most of them can be overcome by using unit tests instead (see Section 4.3). In this section I will point out some of the issues and how they can be avoided [7].

Environmental Dependence As you compare raw console output to expected values it can easily happen that your test cases get dependent on environmental conditions. Lets assume that we have a class called `Lunch`. We want to override the standard `save` method of Django so that a lunch that is being created has it's date automatically set to 12 o'clock. A doctest for this could look like this:

```
def save(self, **kwargs):
    """
    Lunch automatically serves at 12 o'clock

    >>> l = Lunch.objects.create(name = "Spaghetti")
    >>> l
    <Lunch: Spaghetti serves at: 12>
    """
```

To get the custom class description `<Lunch: Spaghetti serves at 12>` we have to define another special method that is invoked in order to create this description. The method is called `__unicode__`.

```
def __unicode__(self):
    return u"%s serves at: %s" % (self.name, self.date.hour)
```

Now our test case depends on the implementation of the `__unicode__` method. If we, for example, change the method to return "... 12 *o'clock*" we did not change the way the `save` method acts but the test will fail because the expected outcome does no longer match the one produced by the Python interpreter.

To overcome the dependence of our test to the implementation of the `__unicode__` method we need to change the test case itself. In fact, we do not have to rely on any vague implementation of a method to print out objects. The better way is to test the attributes directly. We know that we want the hour of our date to be 12 and the name of the lunch should be "Spaghetti". We can now rewrite the test case.

```
def save(self, **kwargs):
    """
    Lunch automatically serves at 12 o'clock

    >>> l = Lunch.objects.create(name = "Spaghetti")
    >>> l.date.hour
    12
    >>> l.name
    "Spaghetti"
    """
```

With that implementation we are no longer dependent on another method than `save` itself.

Besides environmental dependence in the meaning of relying on code of other methods, there are also other traps. For example if you are developing on a UNIX-based operating system paths are separated by a forward slash. On Windows-based systems, a test case relying on this fact would break, because here paths are separated using a backslash.

Database Dependence Django's object-relational manager goes through much trouble in order to abstract from the differences of the underlying database and make them look as similar as possible for the application above. But it is not feasible for Django to make every supported databases look *exactly* the same.

So if you are writing doctests to assure that the rules you have specified for the fields of a class are correct, you can easily run into database dependence. To adhere to the `Lunch` example: if `DATABASE_ENGINE` in your `settings.py` is set to `sqlite3` and you want to check that no `Lunch` will be created with the name left blank, you would write a test case like:

```
def save(self, **kwargs):
    """
    Name may not be NULL

    >>> l = Lunch()
    >>> l.name = None
    >>> l.save()
    Traceback (most recent call last):
    ...
    IntegrityError: lunch.name may not be NULL
    """
```

Here we define that we expect an `IntegrityError` to be raised when we try to save a `Lunch` object with an empty `name` field. Maybe you are wondering why we did not had to paste the whole stack trace. We used the "`...`" operator to tell the Python interpreter that we are not interested in the whole error message. The important parts are that there actually is a *traceback* and that we have found an error message stating that the `name` field may not be `NULL`.

But it is exactly this message that binds us to the SQLite database engine. If we decide to move our application to a mysql database, for example, this test case would fail. Having the same conditions MySQL reports us the error a little different.

```
IntegrityError: (1048, "Column 'name' cannot be null")
```

How can we overcome this problem? We shortly take a rest and think what we want to achieve with the test case. Is it really important what error is reported by the database? No. For us it is enough to know that an `IntegrityError` was raised. We are writing atomic tests that always do only test one thing at a time and because of that we can be sure that all other conditions are correct. The previous test also assures that the `date` field is set to a valid value automatically.

The first solution is to use a doctest directive. These directives are placed as comments on the line that contains the test. The comment has to start with `doctest:` and is followed by one or more directive names. These directive names are proceeded by either a `+` or a `-` to switch them on or off. In our case we use the `IGNORE_EXCEPTION_DETAIL` directive.

```
>>> l.save() # doctest: +IGNORE_EXCEPTION_DETAIL
```

An alternative is to tell the doctest runner to ignore the specifics of the error message using the `ELLIPSIS` marker. It relies on the `ELLIPSIS` option that is by default disabled in Python. But it is enabled by the doctest runner that Django uses. You have already seen this marker before. Remember when we wanted to ignore all the details of the stack trace? There we used the `...` operator. We can do the same to ignore the details of the `IntegrityError`.

```
Traceback (most recent call last):
...
IntegrityError: ...
```

Test Interdependencies The last problem with dependence I want to point out is the interdependence between test cases. When writing tests it happens, that objects that were created for one test case are being reused in another one in order to save cpu time and memory. But reusing objects can result in coupling tests together. A more concrete example is the following: If we are using a PostgreSQL database engine and we write a test like shown in the previous paragraph, we can run into trouble when we are trying to reuse the `Lunch` object we created in the other test case. As a side-effect the PostgreSQL database enters a broken state when the `IntegrityError` is raised. Any next test case will fail as soon as it attempts database access. This example shows us that there is no database isolation between doctest test cases.

A way to overcome this problem is to use the transaction model. By adding the lines

```
>>> from django.db import transaction
>>> transaction.rollback()
```

Django performs a rollback of the last transaction before the `IntegrityError` and the PostgreSQL database is back to normal. This solves this specific issue of the PostgreSQL database and is harmless to other database engines.

In general it is good to look out for two points when writing doctests:

- You should create all objects you need in a test case in the test case itself. Don't rely on objects that could be manipulated by other test cases. Test cases should be atomic. If you delete one test case all others must still pass.
- If you create objects you have to guard them against collision with objects created by other test cases. Let's assume the `name` field of our `Lunch` object is marked as unique.

It may not be good to call the meal "Pizza" because it is likely (assuming you have a "Pizza" addicted development team) that others also give their `Lunch` objects the name "Pizza". When the doctests are run this produces an error as the first object was already saved to the test database. To overcome this you should use names for unique fields, that are unlikely to be used in other tests.

4.2.3 Pros and Cons of Docstring-Testing

Doctests clearly have some outstanding advantages. As you type them directly into the docstring of a method they are available to every Python tool that uses the docstrings to autogenerate help files or documentation out of them. Having your tests in the docstring provides completely unambiguous documentation. The drawback is that you often have a long docstring for a rather short method. And because good and complete testing includes having tests for border cases and all possible errors, you also have to put all these tests into the docstring of a method and blow it up with information that is not at all costs important for documentation purposes.

Another advantage is that you can easily re-use work you have done for example in a shell session. You can simply copy and paste the code you typed into the docstring and have your test ready. But here you are breaking with the development cycle of Test-Driven Development. If you are already able to work in a shell session, it means that you already have implemented something. Even if you type in the doctest from scratch into the docstring of a method you break with the development cycle in the strict sense. It is just the method signature you wrote but you wrote program code before you wrote a test.

One big disadvantage of doctests is that every docstring will count as exactly one doctest. So if you define four tests in one docstring and one of the tests fails you won't get an accurate answer from the test runner which test exactly failed.

4.2.4 Evaluation of *doctest*

During the development of Sendinel we did not use doctest. We decided to use unit testing and to place the test cases for each application into separate files. That way we had all test cases that belong together in one file and did not had to look through different files if a test case failed.

In retrospect it might have been good to have at least some doctests for documentation purposes. Some methods that require explanation, because it is not clear what they exactly do would benefit from doctests. This way we could have the explanation in form of a doctest that on the one hand helps future developers to understand the code and that is another test case to ensure that the method behaves correctly.

But having in mind that we want other developers to continue our work and drive the development of Sendinel or implement it into their solutions, the problems of dependence (see section 4.2.2) would clearly have emerged. If they for example change the database it could have happened that test cases fail and the developer that is new to the code might have no clue why. This is more likely an obstacle that we would put in their way than it actually would

help them. And we ourselves would have had to look out very carefully for these traps. This in turn is very time consuming and we were very pinched for time during the development. So as a downside of our lack in time and knowledge about the advantages of doctests when it comes to documentation, we did not use them.

4.3 Unit Tests with Django

In Section 4.2 we discussed the possibility to have doctests in the docstring of every method. This way of testing has the big advantage, that it does not only act as a test for the method but also serves as an example if you extract a help or documentation file out of the docstring. We also highlighted the problems of dependence. Especially these problems can be overcome by using unit tests instead of doctests.

The xUnit Patterns and the Django Test-Suite Test frameworks exist for several programming languages. Unit testing means testing small parts (units) of a application one at a time. This large number of frameworks is also known collectively as **xUnit** [8]. These frameworks are based on a design by Kent Beck who first implemented it in SUnit for Smalltalk. In this section I will show how the xUnit patterns Assertion, Fixture, Exception Test and All Test are implemented in the Django unit testing framework.

4.3.1 Assertions

As stated in Section 2 assertions are the smallest part when testing applications. They define a set of rules for the behavior of your program code.

I have chosen two representatives for all assertions Django provides to explain the standard functionality and the additional assertions in Django that are specific for a web-framework [9].

assertEquals The `assertEquals` assertion is a wrapper to test equality. It goes along with assertions like `assertTrue` or `failUnlessEqual`. These are assertions that are based on the fact that you have an object or method and you want to compare it to a certain output that you know. You could substitute each of these example assertions with a combination of the others.

Sticking to `assertEquals` the common usage is to provide the assert function with two parameters. It is good style to give the expected value as the first parameter and the object that is being tested as the second parameter Additionally a third parameter can be given. It is usually a string that will be shown if the test fails and that gives an explanation of what is being tested. This message can help the developer to understand the error and to fix it.

assertRedirects As Django is a web-framework, it provides us with some special assertions to test web specific circumstances. If you for example want to test that a certain controller redirects to another page, you could test if the status code is equal to `302`. But this requires

other developers to know the HTTP status codes. With assertions like `assertRedirects` the test case is much easier to read and be understood by other developers.

Django also provides other assertions like `assertTemplateUsed` to ensure that a certain template is used to render a page or `assertFormError` to check if forms are validated correctly.

4.3.2 Fixtures

While writing your tests you will somewhen reach a point where you create the same object for the n-th time at the beginning of your test case. This is *a*) time consuming and *b*) code duplication . If something changes in the way these objects have to be created, you have to change this in every test case. Django provides two mechanisms to establish certain preconditions for your tests automatically. The first one are fixtures that are sourced out into files that are loaded into the test database before a test is run. The second one is the `setUp` method. Here you can write down code that you want to be executed before each test.

Creating Fixtures using the Django Admin-Interface The easiest way to create fixtures is to use the admin interface that Django provides. Here you have a simple graphical user interface that helps you to insert your test data. To use this interface you first have to activate the Django admin module and prepare the model code as described in [10].

To export the data you created into a file use the `dumpdata` command. For example

```
1 $ python manage.py dumpdata --indent 4 > sample_data.json
```

will write the whole database into a JSON [11] file. The hitch is that this will also write out all the data used for the admin interface itself. This is data that is, in most cases, completely unnecessary. To export only the data of a certain Django application you can add the application name after the `dumpdata` command.

```
1 $ python manage.py dumpdata application_name --indent 4 > sample_data.json
```

To use the fixture in a test case you have to move it to `application_name/fixtures/`. Django can now find the fixture file and you can load it by adding one line of code to your test class.

```
1 class ExampleTest:
2
3     fixtures = ['sample_data']
4
5     def testMethod(self):
6         ...
```

Now the sample data we want to use for our test cases is being loaded before all tests are run. That assures that each test has a safe base it can rely on. Every changes we make during a test will be reverted after the test has finished. Even if a test case fails this will not affect the other tests. This way even if two tests use the same objects, they do not affect each other. We have overcome test interdependence (see section 4.2.2) using fixtures along with unit tests.

Naming Conventions and Mysterious Errors If you create a fixture file maybe the filename `initial_data` comes to your mind because you want the data to be loaded when the test database is initiated. But this filename is a special one. Django will load a fixture file called `initial_data` automatically. This file is meant to include constant application data.

Another problem that can occur while using fixtures is that you accidentally misspell your fixture file in the test case. Unfortunately Django does not provide a lot of feedback about loading fixtures. The only fact indicating that something went wrong is that one or more tests will fail. Hopefully the Django team will improve the error reporting in the future but in the meantime you can only look out for mysterious errors like

```
1 DoesNotExist: [...] matching query does not exist.
```

This error is more likely due to an error loading the fixture or in the fixture file itself rather than an error in the test case.

The setUp-Method Sometimes you will find yourself in a situation where you think it is better to create the objects you need close to the tests where you want to use them. Maybe because you think the tests can be better understood this way or because you think a fixture file would be too much for the one or two objects you need. In these cases you can override the `setUp` method. `setUp` is call before every test case. Here you are also able to create objects or execute certain commands to get the system to a state that you want to test.

The tearDown-Method The Django framework will automatically undo all database changes you have made during a test but it can not undo, for example, temporary file creation. To clean up after a test case and to leave the system in a proper state for the other tests you can override the `tearDown` method. This method is called after every test and gives you the opportunity to remove temporary files and to undo changes that may affect other tests.

4.3.3 Exception Test

The pattern states that exception tests work in a way that you catch the exception you expect and pass, or fail if no exception is being raised. Python comes with a convenience wrapper for that.

assertRaises This assertion works in a way that it takes the expected exception as the first parameter, a callable as the second and any further parameters as parameters for the callable. The callable is the function that shall be called by `assertRaises`. This only means that you give the method name without the `()`.

One thing you have to be aware of when testing for exceptions is the `DEBUG`-mode of Django. For example MySQL warnings are only turned into exception if `DEBUG` is set to `True` in the `settings.py`. Though these warnings are still important. If the `strict_mode` of MySQL is not turned on, an attempts to save an object with a blank field that has set `blank = False` will only result in a warning instead of an error. These warnings would simply be ignored by the framework and test cases that expect an exception to be raised would fail.

4.3.4 All Test

By default Django will look up every application for a `models.py` file and if finds one look for the `tests.py` file and execute all the tests in it. You do not have to add tests to a suite manually. All the work is done by Django. To run the complete test suite you have to execute

```
1 $ python manage.py test
```

That way all tests will be run and the result will be shown to you. But the runner now also executes all the tests of Django itself. If you want to run only the tests for your application you can simply attach the application name to the command.

```
1 $ python manage.py test application_name
```

It is even possible to run exactly one test at a time using this command structure. Assuming we have class in the `tests.py` called `ExampleTest` and a test method called `testMethod` than we could write

```
1 $ python manage.py test application_name.ExampleTest.testMethod
```

The tests that shall be run are selected by calling a method called `suite()`. By default this method simply executes all tests in the `tests.py` file or in a `test` module. If you want to have a closer look on how tests can be organized and how to manipulate the `suite()` method go ahead to Section 4.4

4.3.5 Evaluation of Django Unit Tests

During the development of Sendinel we completely focused on unit testing. We forced ourselves to develop tests-first. At the beginning of the project this was hard for everyone because it is a completely different approach to writing software. But as time passed by and we began to rethink parts of the architecture and to refactor the code itself, these tests were extremely helpful for us. I claim that some of the refactoring we did would have been impossible without the tests. We could change major parts of the design relying on the tests that would tell us if the changes we made have broken the system. In section 2.1 I mentioned how automated tests reduce the stress level of application developers. When we started developing no one could imagine how our tests would ever reduce the stress (we had to write the whole system in three weeks) because writing them was time-consuming and thus they even put more stress onto our shoulders. But in the end the strategy worked out. The safety net saved us more than one time from braking the system, plus at each moment we knew if there was still work to be done or not by just looking at the bar. If it was green everything was working and we could go home. Otherwise some more long hours were due.

When we wrote our test cases, we stumbled over one lack in the test suite that was hard to identify. In some way the `Session` object does not work the way it does outside the testing framework. For some test cases we had to create a `Session` because some controller actions did rely on it. We then tried to manipulate the object to establish a state of the system we wanted to test but only errors were produced. We figured out that some of the actions you can normally perform on the `Session` object do not work during testing. So what you have to do is to create controller actions that will do this for you. We do not know why it did not

work the other way around but we spend a lot of time to figure out a solution to run the test nonetheless. So if you are facing problems when working with `Session` in tests you might also try to put the manipulating code into controller actions.

4.4 Organizing the Tests

In Section 4.2 I mentioned that you can either put your tests directly into the docstring of a method if you are using doctest or you put them into the `test.py` file that resides in the folder of every Django application you create. But is that all? If you are building big application that require a lot of tests it can quickly get ugly if you put all of these tests into one file. For Example if you have different classes like `Lunch` and `Dinner` for example you may want to create different files for the tests like `lunch_tests.py` and `dinner_tests.py`. This is no problem at all. Let's assume you want to put all files that contain tests for a certain application together in one folder named `tests`. You simply have to create a Python module named `tests` in your application folder. That means you have to create a subdirectory and place a file called `__init__.py` into it so that Python is able to recognise it as a module. But sadly this is not everything. There is another thing we have to do before Django can find our test cases. As Django utilises `unittest.TestLoader.LoadTestCasesFromModule` to find all the tests in the modules we have to make the test cases we, for example, put into the submodule `lunch_tests.py` visible to the parent `tests` module. We can do so by importing them in the `__init__.py` file.

```
1 from lunch_test import *
```

If you now run your tests you may recognize that something has changed. Some tests are no longer run. This happens if you are using the `__test__` variable to store doctests. As the variable name starts with an underscore that signals that it is a private variable it is not imported when we use a wildcard import. At this point we have to explicitly import the variable.

```
1 from lunch_test import __test__
2 from lunch_test import *
```

But as you may have already recognized we are getting ourselves in big trouble if we use doctest and have chosen to place all our doctests into the `__test__` variable in the corresponding `*_test.py` files. If we for example also import all tests for the `Dinner` class we see that we have a problem.

```
1 from lunch_test import __test__
2 from lunch_test import *
3 from dinner_test import __test__
4 from dinner_test import *
```

The `__test__` variable we imported from `lunch_test` is now being overwritten with the `__test__` variable of `dinner_test` meaning that we loose all doctests we defined for `Lunch`. A quick solution for this particular problem is to move all the doctests from the different files directly into the `__init__.py` file. Mention that you do not need to restrict the dictionary tree to one level. To have a clean structure you may arrange the doctests in the `__test__` dictionary to fit the structure of your files.

The last thing I want to mention in the context of how to organize your tests is overwriting the `suite()` method [12]. You can take full control over what tests are run by Django by defining a function called `suite()` in the `modely.py` and/ or `tests` module. This function is called automatically in order to create a test suite. It simply must return an object that is applicable as an argument for `unittest.TestSuite.addTest`. This can for example be a `unittest.TestSuite`.

```
1 def suite():
2     suite = unittest.TestSuite()
3     suite.addTest(LunchTest('test_initial_time'))
4
5     return suite
```

4.5 Evaluation of the Django Test-Suite

The testing framework provided by Django is a very powerful one. You are able to organize your tests in a way that future developers will not have to struggle with finding the right place to put their tests. It also implements all the main xUnit patterns and thus makes it very easy for the developer to, for example, have example test data loaded automatically before the tests are run. It also automatically creates a test database so that you never have to worry that any of your production data is being touched by a test.

If you are using tools like `pydoc`[10] to generate a documentation right out of the docstrings of your methods, doctests (see Section 4.2) are a really good way to add another help by adding unambiguous examples right to the documentation. As a side effect these examples are also test cases for your code. But if you are using doctests you have to be aware of several problems that come along with them (Section 4.2.2). If these problems are getting to big and it would be a huge effort to overcome them you might consider using unit tests (Section 4.3) instead. Although it is still utile to at least keep doctests that are beneficial for documentation purposes.

Unit tests do not suffer from most of the doctest problems as for example database dependence. With them and special assertions for web-applications that come along with Django, you can build reliable software that works.

When we worked on Sendinel we only utilised unit tests. Without the tests some refactoring we did, especially during the last days of the project, would not have been possible. For example we changed an attribute called `way_of_communication` (for further details see the developer documentation[11]) to become an object. After we had written the tests that described how we wanted the object to behave, we were able to start. First of all we implemented the new object and got all tests for it running. After that we then refactored all the old tests to fit the new implementation. Needless to say, now a lot of tests failed but these tests did tell us exactly what parts of the code had to be refactored in order to integrate the new `way_of_communication` object into Sendinel. After all tests were back to green Sendinel was up and running again and we were satisfied and motivated to write even more tests because we had seen how they lowered the stress level during the integration (Section 2.1). At

[10] `http://docs.python.org/library/pydoc.html`
[11] `http://sendinel.github.com/Sendinel/`

every time we knew if we were done or if there was something left to do.

The downside of Django is the fact that you are not able to create highly interactive user front ends with the framework itself. To do so you have to use technologies such as JavaScript, CSS and HTML5. The front-ends created with these technologies cannot be completely tested with doctests or unit tests provided by Django. It is also very hard to test if all paths through the application work as described in the user stories for example or if all links work correctly.

For example if you have a form inside one of your pages, you can easily write a unit test that assures that the controller action that takes the values works correctly. You just have to simulate a `POST` request and evaluate if the method output is the one you expected it to be. But if you for example only want to allow certain characters to be typed in into the `input` fields of the form, Django cannot provide you with a function to assure this. In general one would accomplish such functionality using a simple JavaScript. But as Django is only able to view the plain output it creates, it is not able to actually *run* the page in a way a browser would do this.

In order to have your complete application tested you have to use other additional frameworks. If you for example have used Ruby on Rails[12] before, you might have seen the cucumber framework[13] for Behavior Driven Development. Here you can describe scenarios in plain text, having a syntax that can almost be read like a description one would normally write down.

In section 5 the Selenium test framework is being described as an add-on to the Django test suite. Selenium gives you the possibility to test your web-applications automatically on different browsers and in different environments. It has a very simple syntax and the test cases can also be easily understood by non-programmers.

5 Front-End Testing With Selenium

When we had a look at the test suite of Django we had to ascertain that neither unit tests nor doctests are very useful when it comes to testing flows through the application. Here different pages and multiple controllers are involved. Writing tests for these use cases was circuitous and hard to maintain for future developers. Another problem is that if you are using JavaScript to create highly interactive user interfaces you cannot test this code with the Django test suite. Therefore we had to look out for another test framework that overcomes these gaps and in this case we found the Selenium framework.

At this point I already want to point out that Selenium is not useful to test whether all links are working correctly. This use case is better covered by crawlers that one after another follow every link of the web-application and check whether it works as expected or not.

[12] `http://rubyonrails.org/`
[13] `http://cukes.info/`

5.1 Why use Selenium

Selenium is especially good for functional tests where certain inputs are given to the system and the resulting output is checked against a specification sheet for example. This so called black box testing abstracts from the underlying implementation and concentrates fully on how the user sees the system. You can write unit tests for every component in the backend and verify that these components work together correctly by taking advantage of Selenium.

You are now also able to test the JavaScript code you have written in the front end by specifying certain workflows and asserting the output you expect. For example if you expect a script to open a new window and create a `div` element in that window, you can use the Selenium IDE to create that workflow and to add the assertions. This test case will fail as long as the JavaScript code does not work properly and will pass if everything is implemented correctly.

If you are using Scrum as a development process, you define the acceptance criteria to your software in user stories. These stories specify exactly what outcome is expected when a user acts a certain way [13]. With Selenium you can map these stories directly to your tests. This helps you to assure that your software does not only behave the way you expected it to behave, but it also works as you defined it in the user stories you created together with your stakeholder.

5.2 The design of Selenium

Selenium is a test framework for web-applications. The Selenium tests run directly in the web browser, thus where the user would interact with the application. It runs on most modern browsers like Mozilla Firefox, Safari or Internet Explorer. There are even plans to bring it to the Safari browser of the iPhone. Supporting these browsers Selenium runs under Windows, Linux or MacOS X and can therefore be called platform independent.

Supporting all these browsers makes it possible for Selenium to test web-applications concerning their compatibility to a certain browser.

Selenium mainly consists of four modules as shown in Figure 5. The heart of Selenium is the Selenium Core. It provides a JavaScript library to simulate user interaction on webpages and abstracts from browser specific implementation details of JavaScript. Thus all other components are build on top of it.

The Selenium IDE [14] is an extension for the Firefox browser that can be used to record tests directly in the browser and to replay them afterwards. As it is an extension only available for the Firefox browser it is coloured grey in Figure 5. In addition tests can be exported so that they can be used with Selenium Remote Control or the Selenium Grid. It also provides the ability to set breakpoints to debug your application in your browser.

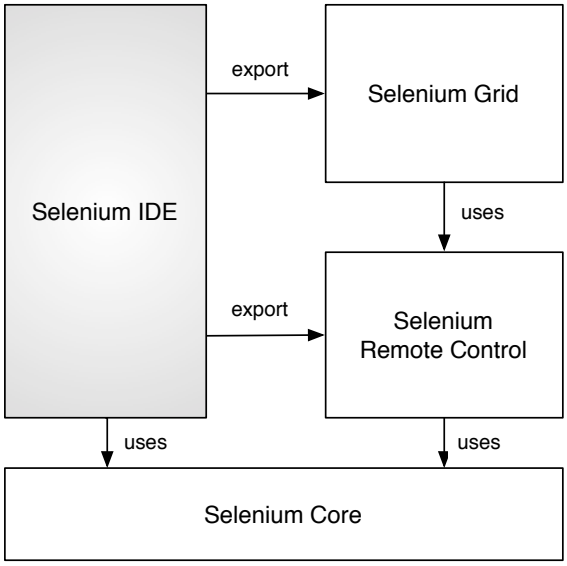

Figure 5: The four main modules of Selenium [15]

The Selenium Remote Control module provides developers to outsource their Selenium tests to other machines. For example if a test suite is run, the unit tests connect via HTTP to a Selenium Remote Control Server and transfer the commands needed to run the tests. For each session a browser is then launched by the Remote Control Server and the tests are run.

Selenium Grid was build to reduce the time needed to execute all Selenium tests and to add more transparency to the underlying infrastructure. Here the Selenium tests are distributed to multiple Selenium Remote Controls to run in parallel. But in advance to Selenium Remote Control itself, the test suite that runs the tests does not need to care about how the tests are distributed. Here the Selenium Grid Server also called Selenium Hub comes into play. The tests are now only send to the Selenium Grid Server which then distributes them using Selenium Remote Control.

As Selenium tests can be easily exported to different programming languages like Python or Java for example and due to the ability to run these tests remotely they can be easily integrated into an ANT-Build or a continuous integration environment [16].

5.3 Selenese

What is Selenese? Put simply: Selenese is the language of Selenium. But it does not require a web developer to learn a new programming language. Selenium tests are displayed as a simple three column table in HTML. Every action is put in a row. Each row follows the structure of:

| Command | Target | Value |

meaning that the first column holds the command that shall be executed by Selenium followed by a reference to the target on which the command will be executed. Targets are HTML elements that are referenced by their `name` tag. The last column contains the value that is handed to the command as a parameter. If you for example have a login form and you want to enter the name `philipp.giese` into the `input` for the username a Selenese script for that could look like the following.

```
<tr>
    <td>type</td>
    <td>username</td>
    <td>philipp.giese</td>
</tr>
```

Having this structure brings two main advantages. You do not have to be a programmer to understand the test cases and it takes developers not long to understand how to write tests. Every written test produces a graphical output if you open the file in a browser.

5.4 Evaluation of Selenium

As we had little time developing Sendinel we could only become acquainted with one testing framework. This was the Django test suite. The front-end was only tested by ourselves manually assuring that Sendinel behaves like defined in the user stories.

In retrospect Selenium would have been a big help for us, as we had to assure that the look and feel of Sendinel is equal in all browsers. The Selenium tests would have been very time saving. We could have set up a Selenium Remote Control Server that would have run all the tests automatically whenever a new build was run by the continuous integration server. That way we could have assured that all user stories are implemented into Sendinel.

The Selenium IDE provides a great interface to record the tests but at this point there already has to be some kind of user interface on which you can perform certain actions. If you want to develop your application test-driven and thus as the first step construct Selenium test cases from the user stories, even before starting to write the tests and the code of the backend you have to write them in Selenese. But as Selenese is easy to learn even for non-developers this should be no big hurdle.

So we see that Selenium perfectly fits into Test-Driven Development. It even might be easier to write the Selenium tests because they can be directly derived from user stories. In contrast this is not given for backend testing as the user might never interact directly with the backend but only through the front end.

6 Conclusion

Test-Driven Development may seem inappropriate to a lot of developers at first glance. I think most of them are scared of the fact that they are not allowed to start coding immediately if an idea comes to their mind. Writing the tests in the first place forces them to rethink everything and to define guidelines for their code before they write it. In fact this is annoying if you are not used to it. I can tell that I was not pleased when the decision was made to develop test-driven. I only thought of the little time we had and how time-consuming it would be to always first write the tests, even if you already had the implementation in mind. But after only three months of development I am completely retuned. The advantages of having automated tests cannot be dismissed. During the last months I did a lot of refactoring that would not have been feasible without the tests that told me if what I was doing was working or not.

The Django test suite was a great help for us. You can easily organise your tests and the variety of assertions ease writing tests a lot. Also Django is very good documented and the community is rapidly growing. This way we were always able to quickly solve problems that occurred. As Django is Python-based you can also benefit from doctests to either have test cases directly where your code is or to have unambiguous examples if you extract a documentation out of the docstrings of your methods.

Unfortunately the lack in time rendered it impossible for us to become acquainted in two testing environments. So we could not add Selenium tests for our front-end. With these tests the complete System would have been developed in a test-driven way. This in turn would have lowered the stress level inside the team because during the development most of the errors were produced by untested front-end code.

Concluding I can say that after the end of the project I have become a fan of Test-Driven Development and Django. Yes, you have to spend time in writing tests and yes, their are times when you have to force yourself to first write a test and then the actual code but it is worth it! The bigger an application gets the more errors can be produced and the automated tests help you to avoid these errors right from the start.

References

[1] Beck, Kent: *Test-Driven Development – By Example*.
Addison Wesley, 2003.

[2] *Triangulation*, June 2010.
http://en.wikipedia.org/wiki/Triangulation, visited on 15.06.2010.

[3] Frister, Michael, Philipp Giese, Patrick Hennig, Thomas Klingbeil, Daniel Moritz, Johan Uhle, and Lea Voget: *The sendinel-project*, June 2010.
http://www.sendinel.org, visited on 16.06.2010.

[4] *Django*, June 2010.
http://en.wikipedia.org/wiki/Django_%28web_framework%29, visited on 23.06.2010.

[5] Moock, Colin: *Essential ActionScript 2.0 – Object-Oriented Development with ActionScript 2.0*.
O'Reilly Media, 2004.

[6] *Django template tags*, June 2010.
http://docs.djangoproject.com/en/1.1/ref/templates/builtins/mpty , visited on 15.06.2010.

[7] Tracey, Karen M.: *Django 1.1 Testing and Debugging*.
Pack Publishing Ltd., 2010.

[8] *xunit*, June 2010.
http://en.wikipedia.org/wiki/XUnit, visited on 16.06.2010.

[9] *Django assertions*, June 2010.
http://docs.djangoproject.com/en/1.2/topics/testing/#assertions, visited on 16.06.2010.

[10] *Django admin module*, June 2010.
http://docs.djangoproject.com/en/dev/ref/contrib/admin/, visited on 16.06.2010.

[11] *Javascript object notation*, June 2010.
http://en.wikipedia.org/wiki/JavaScript_Object_Notation, visited on 16.06.2010.

[12] *Python unittest*, June 2010.
http://docs.python.org/library/unittest.html, visited on 22.06.2010.

[13] Hennig, Patrick: *Software development with scrum in the context of a small project*.
Bachelor's thesis, Hasso-Plattner-Institut, 2010.

[14] *Selenium ide*, June 2010.
http://release.seleniumhq.org/selenium-ide/, visited on 21.06.2010.

[15] Kain, Michael: *Selenium – Web-Applikationen automatisiert testen*.
Open Source Press, 2008.

[16] Frister, Michael: *Continuous integration – practices and tools used in the sendinel project*.
Bachelor's thesis, Hasso-Plattner-Institut, 2010.