An Introduction to Test-Driven Code Generation

Raphael.Marvie@lifl.fr
LIFL / IRCICA
University of Lille 1 (France)

Abstract

Agile Software Development promotes the use of techniques such as *Test-Driven Development* (TDD) and *Automation* in order to improve software quality and to reduce development time. Code generation represents a way to achieve automation, reducing repetitive and error-prone tasks.

Code generation is well accepted, writing a code generator is not necessary that hard, however it is not trivial to decide when and how to embrace code generation. Moreover, it is even harder to embrace at the same time code generation and TDD, wondering for example *How to build a generator following a test driven approach?* or *How to test drive generated code?*

This paper aims at providing hints to answer those questions. It presents an iterative approach named *Test-Driven Code Generation*. The main principle is to gain knowledge about the application during the first iterations of its development process and then to identify how to implement code generation. As code generation should not drive you out of TDD, we provide hints to marry both approaches in order to empower your developments.

Using a simple 3-tiers web application, this paper is illustrated with *Python* standard `unittest` module, *CherryPy* web application server and *Cheetah* templating system.

1 Introduction

The complexity of applications increases while the productivity of developers is still expected to improve. In the meantime, software quality is not expected to vanish. *Test-Driven Development* (TDD) [1] is one of the solutions that helps you write better software. Moreover, with proper training it also improve your productivity (for a given quality level). The leitmotiv of TDD is to *write a test, write the code, and refactor* (TDD loop). Thus, developing a piece of software is seen as a continuous flow of small steps. Going from one step to the next one is bounded by the passing of all the tests: The new test has to pass in order to prove the function has been properly implemented, and the existing tests have to pass in order to guarantee the existing function have not been broken during the new implementation (non regression tests).

The complexity of a piece of software is partly related to the number of its functions. Looking at most pieces of software, many of these functions share a common structure or behavior inside a single application or between applications in the same context. In the context of web applications for example, the implementation of most functions may lay in four categories: input form processing (like inserting a student in the system), output form processing (like obtaining the list of all the students), input / output form processing (like updating the informations related to a student) and search processing (like obtaining the list of non affected projects). Identifying these similarities and understanding their differences permits the development of code generators that (a) reduces the amount of repetitive work (thus the underlying potential errors), (b) improves the software quality (enforcing coding rules, not leaving unfinished copy / paste, and so on), and (c) reduces the development time. In all that, code generation is a way to achieve *Automation* [13], allowing you to focus on the true value of your applications: the specificities of the functions expected by your users (their business rules) instead of some details of their implementation (being sure that all the screens respect the same rules).

Like for any piece of software, developing code generators is not that easy. The main goal of this paper is to present a solution to help you support code generation. We have a look at how to develop a code generator using TDD and how to benefit from code generation while test-driven developing. As an example, we build parts of a 3-tiers web application. The goal is not to compete with frameworks such as *Django* [7] or *TurboGear* [18]. The use of a web application as an illustration example has been motivated by the fact that many readers may know what a web application is.

This introduction to *Test-Driven Code Generation* (TDCG) relies on the use of *Python* standard `unittest` module [19], *CheeryPy* [4] web application framework, and *Cheetah* [3] template engine. *Python* `unittest` is not the only choice for TDD, however it fits well as a standard module and as a support for writing unit tests. Other testing frameworks such as `doctest` [8] and `py.test` [16] can also be considered for this purpose (but have not been experimented yet in the scope of TDCG). *CherryPy* has been chosen for two reasons: Writing *CheeryPy* classes is very similar to writing *Python* classes, and *CheeryPy* classes are well suited for TDD as they are easily test-able without setting up "HTTP testing". Finally, *Cheetah* is both well suited for code generation and quite easy to start working with.

This paper first presents an overview of the proposed approach for developing code generators using TDD as well as a short presentation of the application we are going to build (section 2). As we are about to see in the coming section, writing a code generator begins with writing examples of the code to be generated. Thus, we are going to test-driven develop two use cases (or user stories) of our application (sections 3 and 4). On the basis of these code samples, we are going to build a first version of our code generator (section 5). Finally, as we are producing only part of the application code, we still have to fill the holes. So to be more confident on the code we hand write we will follow the TDD approach. The last part of the article tackles the generation of the test cases (section 6): Here again we do not generate the complete test cases, but only their skeletons[1].

[1]Writing complete test case automatically is possible and already performed in some tools, however it is beyond the scope of this paper.

2 Overview

Extreme Programming encourages as a practice the writing of tests prior to the writing of the code (see Test-First Programming in [2]). This practice has evolved to Test-Driven Development (TDD), and K. Beck has defined in [1] the *TDD mantra* or *TDD loop* (which is expected to be used iteratively and automated) as:

1. Red–Write a little test that doesn't work, and perhaps even compile at first.
2. Green–Make the test work quickly, committing whatever sins necessary in the process.
3. Refactor–Eliminate all of the duplication created in merely getting the test to work.

This section introduces an approach for writing code generators using TDD. It presents the *Double TDD loop* as an extension of K. Beck's *TDD loop*. Finally, it introduces the application we are going to use as an example.

2.1 Approach

Code generation cannot be achieved without knowing what code is to be generated. In other words, it is not possible to generate code that has not been hand written first. While this sentence may sound strange (why developing a generator for producing code we already have) do not forget that code generation is only about automating repetitive or sensitive tasks. Thus, writing the code one or two times first, then writing the code generator brings benefits for the other ten or hundred times you'll use the generator instead of hand-writing the code.

2.1.1 Generators, Tests, and Oracles

The first step in writing a code generator is to write examples of the code to be generated. As any piece of software, the code to be generated has to work and be as clean as possible. Cleanness may be subject to individual feelings while working is easy to give as an qualifier to a piece of software. We already mentioned that TDD is a good solution to write *clean code that works*[2]. Then, there is no reason not to use it for developing the samples of code we want to generate. There is no point in generating (even if it is free) non working or buggy[3] code.

The goal here is to develop code generators, and to use TDD to ease the task. Dealing with TDD, we have to deal with tests thus with *oracles*. An *oracle* is a function that compares an expected result with an actual one and say if it matches or not. For example, when testing a function that apply the taxes to a price, the oracle tells you if the prices with taxes returned by the function is the one expected for known input values. Assertions represent a possible implementation of oracles (a boolean expression that evaluate the returned values to the expected ones). As our motivation is to generate code,

[2]This expression is from Ron Jeffries.

[3]In the meantime, bug-free software does not seem to be reachable. So we put all we can in producing code as bug-free as possible.

the function under test is the generator. Then, we have to define oracles that compare the code produced by the generator for known input values to the expected code: This expected code is the examples of code we hand-write first. When generating source code, this comparison is syntactic. This approach is used whatever the generator produces whole or only part of an application: You can compare complete functions or only their synopsis for example.

2.1.2 Generating Tests

Considering we have automated the production of only part of the application code (for example the declaration of classes and their methods), we now have to fill the holes (like method bodies) with hand written code. Here again, TDD improves the confidence in our work. We cannot simply fill the holes and expect the application to work properly: For each hole to be filled we are expected to apply the *TDD loop*. As we know the functions of the application (in order to generate them) we have a good basis for defining their tests. In this paper we consider only the skeletons of the test cases, not their body[4] (obtaining an oracle index). Nevertheless, we are going to automate the writing of these test case skeletons and of the test suite associated to the application.

Finally, we expect the development of code generators not to follow a waterfall approach, but an iterative one. Your code generator cannot be completed in a simple cycle: Seek first a generator that works for a small part of the application, then improve it little by little. Add capabilities to your generator as you add functions to any of your applications: One by one, being sure that each one is working properly before going to the next one.

2.2 The double TDD loop

The *double TDD loop* principle summarizes our proposal for the support of *Test-Driven Code Generation*. It is a simple extension of the *TDD loop* (which is used 'as is' during step 1) and you are encouraged to use this double TDD loop iteratively, producing the generator incrementally. More often, step 1 has to be repeated several times before moving to step 2 in order to gain enough knowledge about what can be generated.

1. Have some working code.
   (a) Write the test for a small part of the application, this small part may range from a simple function to a class (going bigger than a class may need several iterations).
   (b) Implement the selected part of the application until the test passes (as well as all the existing ones).
   (c) On the basis of this implementation and the previous ones (if it applies) refactor[5] the code [9].

[4]Complete test suite, test case and assertions can be generated from detailed specification of a piece of software. However, this goes beyond the scope of this paper, and we are only going to discuss the generation of the test suite, and test case skeletons.

[5]A translation of M. Fowler's Book first chapter examples with *Python* is available at: `hiper.com.br/python/refactor/index.html`

2. Extract the part that can be generated (from complete functions to skeletons).
3. Develop the piece of the generator that generate the part of the code defined in the previous step.
   (a) Write the test for the generation of this small part of the application. This test mainly compares the code produced by the generator to the hand written code extracted in step 2 while providing the proper input data.
   (b) Implement the part of the generator that produces this small part of the application until the test passes (as well as all the existing ones for non regression).
   (c) Refactor the generator to reduce code duplication and improve its clarity, ability to evolution, and maintainability. This should have no impact on the generated code.

Remark When dealing with iterations, step 1.c may also include updating the code generator. If the structure of the application is refactored when adding a new part, the generator for existing code may have to be updated too, otherwise some tests may no longer pass. Such an update is often limited to modifying the code templates. Code generator refactoring performed during step 6 is on the other hand targeted at the generator structure / organization, not at refactoring the generated code. When dealing with code generation, the tests, the application, and the generator have to be kept synchronized.

2.3 Illustrative Example

In order to illustrate this paper, we present some excerpts of the development of a simple 3-tiers application in order to manage student technical projects (only excerpts of the application are presented). During the final year of Master Degree, technical projects are proposed by teachers (or researchers) to pairs of Master students for about 100 hours of homework.

First, the application is opened during two weeks for researchers / teachers to submit an abstract of their project. Then, the application is closed for project submission and is opened during one week for student pairs to select 3 projects that interest them. Finally, the person supervising the technical projects affects a project to each student pair. At any time, any user (teachers, students and the supervisor) can list the submitted projects. Finally, once the projects are affected every user can list the results (project name / pair of students). These application use cases are depicted in figure 1.

Having a good idea of the application we have to produce, we are going to have a look at the development of a few use cases in order to understand TDCG. The development process is iterative and rely on the complete development of one use case at a time.

3 First step: classical TDD

Code cannot be generated without being written first by hand (at least once). This section is a direct application of the *TDD loop* to the development of a

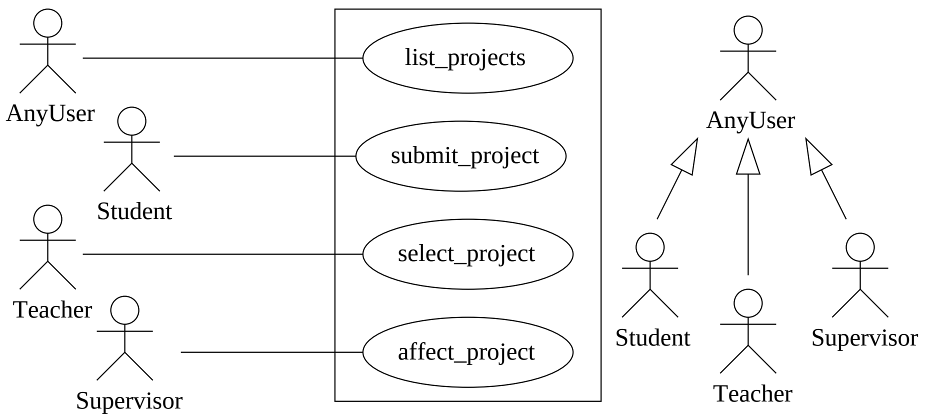

Figure 1: Use Cases for the Management of Technical Projects

3-tiers web application using *CherryPy*. The overall structure of the application follows the *Model / View / Controller* design pattern: The presentation layer (view) is coded using *CherryPy*, the processing layer (controller) is composed of pure *Python* classes, and the data layer (model) is made of simple objects in this paper (but it could be defined as a relational / object mapping like SQLObject [17]).

Starting from the use cases presented in figure 1, an actor is translated as a controller which is implemented as a class. Each action that can be performed by this actor is translated as an operation of the controller class. The actions are processing data items that can be displayed or retrieved from the presentation layer. Figure 2 presents a simplified version of the application architecture. It underlines two aspects of the application: A use case is orthogonal to the three tiers of the application, and the application can be seen as a stack of use case implementations.

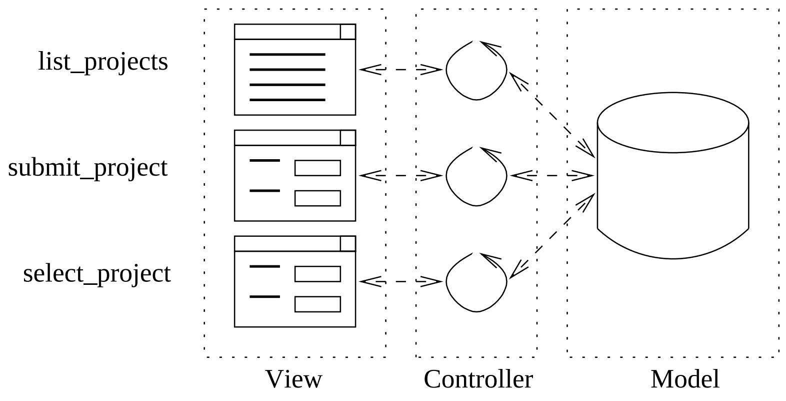

Figure 2: Architecture of the Technical Project Management System (excerpt)

3.1 Overview

For this first step, we choose to implement the `list_projects` use case. This use case is available to any user during the whole life-cycle of the application. Before going to the code, let's have a look at a more detailed presentation of the use case (sometime called textual scenario). The scenario here is really simple:

1. A user requests the list of projects.
2. The system provides the list of projects.

So first we have to define the concept of project (model), then develop the operation inside the controller class, and finally develop the presentation class (view) that provides access to this controller.

3.2 Model

At this point, the model is limited to the concept of project. In order to stay simple, and as we only use part of the real application as an illustration for this paper, we are going to consider a project to be defined as a name (and that's all). We do not enter the details of the persistence solutions that could be used, and we stick to the use of simple *Python* objects.

3.2.1 Tests

We consider a `Project` as a simple object with a single attribute containing its name. The test is quite trivial: We instantiate a `Project` and we compare the value of its attribute to the value we have provided to its constructor. The complete test is given in figure 3. After making this test fail, (first because neither the `abstraction` module nor the `Project` class have been defined yet, and second because the attribute would not have been set in the constructor –in order to watch the test failing) we are going to make it pass.

```
import unittest
import abstraction
class TestProject(unittest.TestCase):
    def test_project(self):
        p = abstraction.Project('Project1')
        self.assertEqual(p.name, 'Project1')
if __name__ == '__main__':
    unittest.main()
```

Figure 3: Test case of the `Project` class (Model)

3.2.2 Implementation

Based upon the test defined in figure 3, which defines the goal of the current implementation step, we write the `Project` class presented in figure 4: a simple class, containing the read-only property `name` that is initialized at object creation (it is difficult to make it simpler[6]).

[6]You could argue that the use of a property is not simpler than using an attribute. That is correct, however the property is defined as read-only which is interesting.

```
class Project (object):
    def __init__(self, name):
        super(Project, self).__init__()
        self._name = name
    def get_name(self):
        return self._name
    name = property(get_name)
```

Figure 4: First implementation of the `Project` class (Model)

We now have an early simple implementation of the data-layer for our first use case. We won't go further on in the development of the model layer for the moment. Its implementation will be improved in the future use case implementation. The development process is iterative.

3.3 Controller

The second part of our use case implementation is to develop the controller class. Here again, the implementation is very simple. The `list_projects` operation does not take any parameter and returns a list of projects. Baby steps is another principle of *eXtreme Programming* (see [2] p. 33).

3.3.1 Tests

The intent of a test `setUp` operation is to capitalize the initialization of tests for a given test case: It reduces code duplication and centralizes the setting up of the test environment. Here, it creates a list of two projects, and instantiate a `AnyUserController` controller providing the list of projects for its initialization. The test of the `list_projects` operation checks that the two projects provided during initialization are in the list of projects returned and only those two. Figure 5 presents this test case.

```
class TestAnyUserController(unittest.TestCase):
    def setUp(self):
        self.projects = [abstraction.Project('Project1'),
                         abstraction.Project('Project2')]
        self.ctrl = controller.AnyUserController(self.projects)
    def test_list_projects(self):
        self.assertEqual(2, len(self.ctrl.list_projects()))
        self.assertTrue(self.projects[0] in self.ctrl.list_projects())
        self.assertTrue(self.projects[1] in self.ctrl.list_projects())
```

Figure 5: Test case of the `list_projects` operation (Controller)

Remark For a class to be unitary tested, it has to be implemented with explicit dependencies: The dependencies of the class should be provided at construction time or using writing accessors. If dependencies are instantiated inside the class, it is not possible to test it independently of them (for example using fake implementations for the dependencies as presented in section 3.4.1).

This requirement is strongly related to the *favor loose coupling of software components* principle (see *Orthogonality* in [12] p. 34). Expliciting interactions between software artifacts eases their substitution and reuse.

3.3.2 Implementation

Keeping the *simple is beautiful* paradigm in mind, figure 6 presents an implementation that passes the test defined in figure 5. In order to reduce the risk of messing the list from the outside of the controller, the `list_projects` returns a copy of the original list provided at creation time[7]. Later on in the development iterations, this list will be replaced for example by the persistent layer entry point.

```
class AnyUserController (object):
    def __init__(self, projects):
        super(AnyUserController, self).__init__()
        self._projects = projects
    def list_projects(self):
        return self._projects[:]
```

Figure 6: Implementation of the `list_projects` operation (Controller)

3.4 View

We now have to define the presentation layer (view) of the `list_projects` use case. The view implementation is defined as a *CherryPy* class providing an operation named like the use case and that returns an HTML version of the list.

3.4.1 Tests

The test for the presentation class has two specificities. First, we compare the output of the view `list_projects` operation to an expected HTML output. Second, in order to develop a Unit Test (testing only the view and not the controller at the same time) the test class is also used as a *mock object*.

A mock object is used in place of a real object (having the same interface) of the application in order to test the class depending on it in a well known context. It avoids, when a bug is identified, to wonder in which class the bug really lays. It also permits to test the class in exceptional situations, like when a network or a disk error occurs. A mock object behaves as an object of the application but for specific data only[8] in [1]. Even if the controller has been tested, we test the view implementation on its own (which is the definition of unit test). So the `TestAnyUserView` class mocks the behavior of the controller providing a `list_projects` operation to the view implementation.

Figure 7 presents the test case for the view implementation. The controller provided to the view is the test case object itself (which behaves as a mock

[7]Good or bad, this is an habit when the returned list is not intended for modification. Elements still can be modified, but not removed from nor added to the original list.

[8]This technique is further discussed in [1] (Self Hunt p. 144). You can also have a look at `www.mockobjects.com` and [14] for more information about using mocks with Python.

object). The test of the `list_projects` operation checks that it has an attribute `exposed` set to `True` (*CherryPy* only provides to users operations that are marked as *exposed*, allowing the definition of internal / protected operations). Second, the output of the operation is compared to the expected one (on the basis of the test set up that provides well known information to the operation) stored in the `expected_list_projects` variable.

```
expected_list_projects = '''<ul>
<li> Project1
<li> Project2
</ul>'''
class TestAnyUserView(unittest.TestCase):
    def setUp(self):
        self.view = view.AnyUserView(self)
        self.projects = [abstraction.Project('Project1'),
                         abstraction.Project('Project2')]
    def list_projects(self):
        return self.projects[:]
    def test_list_projects(self):
        self.assertTrue(self.view.list_projects.exposed)
        self.assertEqual(expected_list_projects,
                         self.view.list_projects())
```

Figure 7: Test case of the `list_projects` operation (View)

We can see here the benefit of using *CherryPy* for building testable web applications: Operations can be called independently of the web server. A classical web application could also be tested using *Python* standard `httplib` module [11], but it would be a little bit more technical. These two solutions are in fact complimentary.

3.4.2 Implementation

On the basis of the test defined in figure 7 we can implement the `list_projects` operation of the presentation class. Figure 8 presents a simple version of this operation. An `AnyUserView` object stores the controller provided at initialization (a dependency), and uses it when the list of projects is requested. A list of HTML lines is built on the basis of the information provided by the controller, and the page is created at return time. This first version does not use *Cheetah* templating system to remain simple in a first time.

We do not consider the full HTML page for the test of the application. We only focus on the information provided by the application as this is what really matters. Moreover, when the important information is lost in a big HTML document, it is harder to test. Finally, on the basis of the current implementation of the `list_project` operation, a simple wrapper function can be defined to embed the returned HTML excerpt in a full-flavored HTML page that gives a nice interface. Using *Python* decorators or defining a meta-class for the module permits to easily apply this wrapper. Once tested, such a wrapper function could be applied to the results in order to get a fancy displaying.

```
import cherrypy
class AnyUserView(object):
    def __init__(self, controller):
        super(AnyUserView, self).__init__()
        self._controller = controller
    @cherrypy.expose
    def list_projects(self):
        result = ['<ul>']
        for p in self._controller.list_projects():
            result.append('<li> ' + p.name)
        result.append('</ul>')
        return '\n'.join(result)
```

Figure 8: Implementation of the `list_projects` operation (View)

3.5 Main

To complete this first step, figure 9 presents the main part of the application. This bootstrap creates an object and registers it to the *CherryPy* web server for providing services to end users. The root attribute of the *CherryPy* server represents the base URL of the web site. This object can in turn have children that represent the directories of the web server. Once this bootstrap written, a first (very limited) version of the application may be demonstrated, providing feedback from the end users. Feedback is defined as a value by *eXtreme Programming* (see [2] p.19) and encouraged in agile development (see [5] p.66 and Tool 3 of [15] p.22).

```
import cherrypy
import abstraction
import controller
import view
projects = [abstraction.Project('Project1'),
            abstraction.Project('Project2')]
cherrypy.root = view.AnyUserView(controller.AnyUserController(projects))
if __name__ == '__main__':
    cherrypy.server.start()
```

Figure 9: Bootstrap for the application

Starting the application with `python main.py` and pointing a web browser to the location `http://localhost:8080/list_projects` will provide you an HTML version of the list of (dummy) projects.

Remark Code generation can be seen as a generalization or abstraction of a solution. So we cannot directly move to it on the basis of a single use case implementation. Before, we shall implement a second use case. In practice, the author has implemented most of the use cases by hand before developing the generator. Code generation is to be used mainly when you see repetitive work that can be automated, like for patterns: *One is an exception, Two is a coincidence, Three may be the beginning of a pattern* [10]. Remember that we want to ease the development of 3-tiers web applications based on the MVC pattern, not the development of this specific application.

4 First step (bis): Another use case using TDD

The second use case we implement is the definition of a new project by a teacher. This second use case shows us how to process information retrieved from the user. The user enters the name of a project (keeping our simple `Project` data structure), and the system creates a new project that is added to the list of existing ones.

4.1 Teacher controller

Figure 10 presents the test case of the `add_project` operation for the teacher controller. Like described in figure 1, a teacher extends the abilities of the default actor (`AnyUser`) of our use cases. Then, any operation provided to the default actor has to be provided to a teacher. Thus, the teacher controller have to pass the same tests as the `AnyUserController` (extension of use case).

The test case of the teacher controller is defined as an extension (using inheritance) of the any user test case. The difference between both test case set up is the controller to be instantiated. In the meantime, the list of default projects have to be created in both test cases. So, the `TestAnyUserController` test case `setUp` is refactored in order to introduce a new operation (`_make_projects`) that creates this list. Finally, the operation `test_add_project` checks that projects are added to the list of existing ones (which size is increased).

```
class TeacherControllerTests(AnyUserControllerTests):
    def setUp(self):
        p = self._make_projects()
        self.ctrl = controller.TeacherController(p)
    def test_add_project(self):
        p1 = abstraction.Project('MyProject1')
        self.ctrl.add_project(p1)
        p2 = abstraction.Project('MyProject2')
        self.ctrl.add_project(p2)
        self.assertTrue(p1 in self.ctrl.list_projects())
        self.assertTrue(p2 in self.ctrl.list_projects())
        self.assertEqual(4, len(self.ctrl.list_projects()))
```

Figure 10: Test case for the `add_project` operation (Teacher controller)

Figure 11 presents the implementation of the `add_project` operation that passes the tests defined in figure 10. This operation simply add the new project to the list of existing ones. It is also trivial and you may wonder: *Why having such code in my application?* This version is a first version to get something to show, a real implementation would probably check that the name of the project is not already used in the existing ones, or send an e-mail to the submitter in order for her/him to have the identifier of the project (for later update) and an abstract of her/his submission, etc.

```
class TeacherController(AnyUserController):
    def __init__(self, projects=None):
        super(TeacherController, self).__init__(projects)
    def add_project(self, project):
        self._projects.append(project)
```

Figure 11: Implementation of the add_project operation (Teacher controller)

4.2 Teacher view

Figure 12 presents the test of the `add_project` operation for the teacher's view. Like in section 3.4.1, the test mocks the controller behavior (`add_project`). The `add_project` operation of the view has two behaviors: If the operation is called without parameters, it provides the form to be filled for defining a new project, else it displays an HTML message saying that the project has been added to the list of existing projects. We do not check for failing of the insertion as it is only append to a list (and not to a persistent layer). The test checks these two outputs together with the actual call to the `add_project` operation of the teacher controller.

```
expected_add_project_default = \
'''<form action="add_project" method="post">
Project: <input name="name" /><br />
<input type="submit" />
</form>'''
expected_add_project_filled = 'Your project <i>Project3</i> has been added.'
class TeacherViewTests(AnyUserViewTests):
    def setUp(self):
        self.view = view.TeacherView(self)
        self.projects = [abstraction.Project('Project1'),
                         abstraction.Project('Project2')]
    def add_project(self, project):
        '''Mock the controller behavior'''
        self.projects.append(project)
    def test_add_project(self):
        self.assertTrue(self.view.add_project.exposed)
        self.assertEqual(expected_add_project_default,
                         self.view.add_project())
        self.assertEqual(expected_add_project_filled,
                         self.view.add_project('Project3'))
        self.assertTrue(len(self.projects) == 3)
        self.assertTrue(self.projects[2].name == 'Project3')
```

Figure 12: Test case for the add_project operation (Teacher view)

Finally, figure 13 presents the implementation of the `add_project` operation that passes the tests defined in figure 10. The operation returns the form to add a project by default (when no name is provided), or it creates a `Project`, adds it into the list of existing ones using the controller and returns a message saying that the project has been added. (Just what we were expecting.)

```
class TeacherView(AnyUserView):
    def __init__(self, controller):
        super(TeacherView, self).__init__(controller)
    @cherrypy.expose
    def add_project(self, name=None):
        if not name:
            return '<form action="add_project" method="post">\n' + \
                   'Project: <input name="name" /><br />\n' + \
                   '<input type="submit" />\n</form>'
        else:
            project = abstraction.Project(name)
            self._controller.add_project(project)
            return 'Your project <i>%s</i> has been added.' % name
```

Figure 13: Implementation of the `add_project` operation (Teacher view)

4.3 Remarks

4.3.1 Managing the tests

The multiplication of test cases introduces the need for defining test suites. Figure 14 presents the test suite for the presentation layer. There is one operation for creating a test suite in every test module (meaning one for the abstraction, one for the controller, one for the view, and one for the main). There is also a `test_all` script that calls each of those suites in order to make the general test suite of the application. Like classes are organized in packages and modules, tests are organized in (recursive) test suites.

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(unittest.makeSuite(AnyUserViewTests))
    suite.addTest(unittest.makeSuite(TeacherViewTests))
    return suite
if __name__ == '__main__':
    unittest.TextTestRunner(verbosity=2).run(suite())
```

Figure 14: Test suite for the view module

4.3.2 Index operations for views

Each view has an `index` operation, so has the root page of the server. This operation is called by default when accessing a view without specifying the operation to be performed (*CherryPy* way of managing the `index.html` default page). In our example, it provides access to the various operations provided by the teacher view.

Figure 15 presents the new version of the application bootstrap. The `Main` class only provides the `index` operation that gives access to the various views depending whether you are a student or a teacher. Here again, the application (while still being very limited) can be run to get feedback from the end users.

```
import cherrypy
import abstraction
import controller
import view
class Main(object):
    def __init__(self):
        super(Main, self).__init__()
    @cherrypy.expose
    def index(self):
        return '<a href="default">AnyUser profile</a><br>\n' + \
               '<a href="teacher">Teacher profile</a>'
projects = list()
cherrypy.root = Main()
cherrypy.root.default = \
    view.AnyUserView(controller.AnyUserController(projects))
cherrypy.root.teacher = \
    view.TeacherView(controller.TeacherController(projects))
if __name__ == '__main__':
    cherrypy.server.start()
```

Figure 15: Bootstrap of the application

In production mode, you certainly expect the view not to be easily accessible to all the users, in order to be sure that a student does not insert a project subject for example. The production implementation can split the view into distinct base URLs without common root or between different servers, listening on different ports for example. Finally, an authentication mechanism could be important (but this goes beyond the scope of this paper).

4.3.3 What have we learned?

The structure of the application is now outlined. It contains three layers, split into four modules: abstraction, controller, view, and main. A test suite is associated to each module, and a global test suite groups all the module suites (easing the test of the whole application). We have two working operations that can be demonstrated to end users for validation. We have two examples of operation definitions for controllers and views: An operation that produces output only, and an operation that takes input and processes it. This looks like a good starting point to move on and work on a first attempt at developing a simple code generator.

5 Second step: the generator using TDD

We consider in this section a generator that produces a simple skeleton of our application modules. Based upon the knowledge we acquired during the first step, we can already automatically produce some parts of our application like its overall structure and the base definitions of controller and view classes. This section presents the definition of such a simple generator using a test-driven approach.

5.1 Overview

The development of the generator is performed on the basis of the implementations of controllers and views we have developed in sections 3 and 4. On the basis of these implementations, we can extract some skeletons of controller and view classes. These skeletons represent the common parts of the hand written code, while the completion of these skeletons represent the specific parts. They are the expected the output of the generator. Each skeleton is a template of code that is specialized for a particular class. In the meantime, we have to define the information to be used as input in order to generate the skeletons: the names of controllers and views as well as the name of their operations. On this basis, test cases for the generator are defined using expected skeletons as oracles when providing the generator known input information.

The second point we have to decide is where does the input information for the generator come from. Figure 1 has presented, using a simple use case diagram, the kind of application we want a generator for. Thus, this looks like a good start: We are going to generate the structure and some skeleton classes for the controller and view layers of a 3-tiers application on the basis of its use cases (whatever the format used for their definition).

A generator may not always take a single kind of information as input for producing code. Use cases only allow the definition of controllers and views but not the definition of the model (data structures used by the application). For the later, we could use a class diagram defining both the data structure and their relations in order to generate the database schema and the associated *Python* objects (for example). This approach is very common whatever it is hand made following the Direct Access Object Pattern or using off the shelve solutions such as *SQLObject* [17].

```
controller_test = '''
import abstraction
class AnyUserController(object):
    def __init__(self):
        super(AnyUserController, self).__init__()
    def list_projects(self):
        pass
'''
class TestControllerGenerator(unittest.TestCase):
    def setUp(self):
        self.generator = generator.ControllerGenerator()
    def test_generate(self):
        model = {'AnyUser': ['list_projects']}
        self.assertEqual(self.generator.generate(model),
                         controller_test)
```

Figure 16: Test case of the controller generator class

5.2 Generation of the controller skeletons

To begin with, we are going to look at the generation of the controller skeletons. A generator is as important as the applications it produces: It has to be properly developed, tested, documented and maintained. As such first-class piece of software we apply TDD to improve its overall quality.

5.2.1 Tests

Unit tests are defined on the basis of the application expected output and behavior. The expected behavior of our generator is to produce source code, and the expected output is skeletons of controllers. On the basis of the controller presented in figure 6 and figure 11 we extract the skeleton `controller_test` depicted in figure 16. This skeleton is the expected output of the generator if we provide the definition of the first use case (`list_projects`) as input. So the test is quite straightforward: We provide the name of the actor together with a list of the associated actions (s)he can perform and we compare the result of the generator execution with the expected skeleton.

With this first test, we can begin the implementation of the code generator. This implementation is iterative also, moving quietly one test at a time.

```
from Cheetah.Template import Template
ctrl_template = '''
import abstraction
#for $actor in $model.keys()
class ${actor}Controller(object):
    def __init__(self):
        super(${actor}Controller, self).__init__()
  #for $operation in $model[$actor]
    def ${operation}(self):
        pass
  #end for
#end for
'''
class ControllerGenerator(object):
    def __init__(self):
        super(ControllerGenerator, self).__init__()
    def generate(self, model):
        return str(Template(ctrl_template, searchList=[{'model': model}]))
```

Figure 17: Implementation of the controller generator class

5.2.2 Implementation

There are a bunch of solutions for generating code, ranging from the use of standard *Python* template strings (like `"hello %s" % name`) that allow the production of the code slice by slice to the use of a template system like *Cheetah* [3]. We use the second solution, defining a template for each file to be generated. With both solutions the approach is similar: Code with holes is defined (which defines the abstract solution), and the holes are filled using information in order to produce the solution for a specific context (providing the concrete solution).

Cheetah targets first the generation of web pages (like PHP would do) but it is also fully useable for the generation of code (whatever the target programming language[9]). We use *Cheetah* the basic way[10]: A template is defined as a multi-line string, then the template is processed on-the-fly providing the proper input (the information to fill the holes).

Figure 17 presents a first implementation of the controller generator. The template (`ctrl_template`) states that for each actor of the provided use case (which definition is contained in `model` that is a dictionary), a class is defined and named like the actor postfixed with *Controller*. Then, for each action associated to the actor, an operation is defined and named like the use case action (without parameters for the moment). The generator simply evaluates the template using the provided information (stored in the dictionary `searchList`). Finally, the *code object* created by the call to `Template()` is translated to its string representation and returned.

In order to differentiate *Cheetah* keywords from *Python* ones, the former ones start with a '`#`'. To differentiate *Cheetah* variables from *Python* ones, the former ones start with a '`$`'. When accessing the content of a *Cheetah* variable, you can use curly braces to avoid ambiguities: `${actor}Controller` concatenates the content of the `actor` variable to the string `Controller`, while `$actorController` would look for the content of the `actorController` variable.

```
class TestBase(unittest.TestCase):
    def test_generate(self):
        self.model = {'AnyUser': ['list_projects']}
        self.assertEqual(self.generator.generate(self.model),
                         self.expected_output)
view_test = '''
import cherrypy
class AnyUserView(object):
    def __init__(self, controller):
        super(AnyUserView, self).__init__()
        self._controller = controller
    @cherrypy.expose
    def index(self):
        return 'To be completed'
    @cherrypy.expose
    def list_projects(self):
        return 'To be completed'
'''
class TestViewGenerator(TestBase):
    def setUp(self):
        self.expected_output = view_test
        self.generator = generator.ViewGenerator()
```

Figure 18: Test case of the view generator class

[9]We have successfully used *Cheetah* for the generation of *Python*, *Java*, and *SQL* source code.

[10]Templates could be compiled into memory in order to improve their efficiency. They could also be defined in dedicated files. But this won't be necessary for our purpose (at least for the moment).

5.3 Generation of the view skeletons

Implementing the generation of view skeletons is very similar to Implementing the generation of controller ones: Write the test using expected generated code as oracle and write the template that produces this code.

5.3.1 Tests

Writing the test case for the view generator clearly shows that tests are similar between controller and view generators. So similar that you can imagine it (go back to section 5.2.1 otherwise) and we won't present it. When two pieces of code are similar it is time for refactoring. In order to remove duplication as much as possible, we define a base test case for skeleton generators (see figure 18) that is inherited by both the controller and the view test cases (respectively `TestControllerGenerator` and `TestViewGenerator`). The only difference lays in the `setUp` operation that fixes the proper generator to test and the proper test oracle. The test oracle for the view (`view_test`) is defined similarly to the test oracle for the controller as a multi-line of expected code.

5.3.2 Implementation

Now we got the test, we can jump to the implementation. Like when we wrote the test, we see that the code we write is very similar to the code we wrote for the controller generator (see section 5.2.2). So here again we won't have a look at the first written version, but at the refactored one.

Figure 19 presents the refactored version of the generator implementations. A base class is defined for generators, containing all the behavior: setting of the template string, and call to `Template`. Then, for each generator we only have to write the template definition and to inherit the base generator class in order to set the proper template. Figure 20 presents the template for the generation of the view.

```
class BaseGenerator(object):
    def __init__(self, template):
        super(BaseGenerator, self).__init__()
        self._template = template
    def generate(self, model):
        return str(Template(self._template, searchList=[{'model': model}]))
class ViewGenerator(BaseGenerator):
    def __init__(self):
        super(ViewGenerator, self).__init__(view_template)
```

Figure 19: Implementation of the view generator class (refactored)

You may wonder why using a subclass while the only difference lays in the template used. There are two reasons: First the existing generator is defined as a class, so we do not break existing code as it may be used at several places in the code, second this solution sounds more open for evolution (like extending processing of the generate method for example). It is important not to think of re-useness *before* useness, however code should be written in a re-usable fashion.

The balance is definitely hard to find, and your experience is your most valuable friend.

```
view_template = '''
import cherrypy
#for $actor in $model.keys()
class ${actor}View(object):
    def __init__(self, controller):
        super(${actor}View, self).__init__()
        self._controller = controller
    @cherrypy.expose
    def index(self):
        return 'To be completed'
  #for $operation in $model[$actor]
    @cherrypy.expose
    def ${operation}(self):
        return 'To be completed'
  #end for
#end for
'''
```

Figure 20: Template for the generation of views

5.4 Generation of the application

Generating partly or completely an application is not limited to the generation of source code. The code has to be organized into files, most of the time respecting a structure on the file system. This means that a basic knowledge of file system use is important for code generation.

For the moment, our application structure is defined as follows. The files of an application are stored in a folder named like the application. Each file is a *Python* module containing a set of related classes: a module for the data structures (`abstraction.py`), a module for the controllers (`controller.py`), a module for the views (`views.py`), and a module for the bootstrap of the application (`main.py`).

5.4.1 Tests

Generating the structure of the application and producing the proper code in the proper file has also to be tested. Figure 21 presents the test for the complete generator. This test checks that the various files that should be produced by the generator are properly produced and well organized. While this test is written using the `unittest` module, it is not a unit test only but an integration one. It does not simply check the class `MyGenerator` behavior using for example mock objects, it really checks how the various classes are working together. You may find the last assertion to be redundant with the tests we previously wrote, however we control that the generated code is stored in the correct file also.

```
expected_file_content = {
    'abstraction.py': abstraction_test,
    'controller.py': controller_test,
    'view.py': view_test,
    'main.py': main_test,
    }
class TestMyGenerator(unittest.TestCase):
    def setUp(self):
        self.model = {'MasterProjects': {'AnyUser': ['list_projects']}}
        self.generator = generator.MyGenerator()
    def test_files(self):
        self.generator.generate(self.model)
        self.assertTrue('MasterProjects' in os.listdir('.'))
        files = os.listdir('MasterProjects')
        for f in expected_file_content:
            self.assertTrue(f in files)
            self.assertEqual(file('MasterProjects/%s' % f).read(),
                             expected_file_content[f])
```

Figure 21: Test case of the generator main class

5.4.2 Implementation

Figure 22 presents the implementation of the main class of the generator developed on the basis of the test cases defined in figure 21. This part of the generator simply creates the folder and files associated to the web application defined by the use case and uses the various generators we developed in the previous sections to produce the content of these files. This class is a simple coordinator of the various pieces of the generator.

```
class MyGenerator(object):
    def __init__(self):
        super(MyGenerator, self).__init__()
        self.generators = {
            'asbtraction.py': AbstractionGenerator(),
            'controller.py': ControllerGenerator(),
            'view.py': ViewGenerator(),
            'main.py': MainGenerator()
        }
    def _create_file(self, path, content):
        file('%s/%s' % path, 'w').write(content)
    def generate(self, model):
        for usecase in model:
            os.mkdir(usecase)
            for name, g in self.generators.items():
                self._create_file((usecase, name),
                                  g.generate(model[usecase]))
```

Figure 22: Implementation of the generator main class

5.5 About the input model

For all the tests we wrote, we used a simple *Python* data structure composed of dictionaries and lists in order to provide the input data to generators. This data structure could be used for providing input data in production mode, however it may not be the best solution. Keeping this approach would require first to check that the information is properly structured. There is not much testing in the generators for well-formness of the input at the moment (but it can be improved).

Another solution (that sounds better) is to produce this structure of information from a more user oriented form[11]. A standard input format for code generation is XML[12]. The advantage of XML is that a bunch of parsers are available and parsing XML using *Python* is not hard. Moreover, in order to have correct input, we can validate the XML document against a DTD before producing the *Python* data structure we currently use as input for our generator. Two small steps are better than a big one for modularity reasons: Each tool may be useful in more than one context. Figure 23 presents a simple document structure for defining the input data that could be used with our generator.

```
<?xml version="1.0" ?>
<usecase name="MasterProjects">
  <actor name="AnyUser">
    <action name="list_projects" />
  </actor>
  <actor name="Teacher">
    <extends>AnyUser</extends>
    <action name="add_project" />
  </actor>
</usecase>
```

Figure 23: XML version of the input information

Finally, we used a use case diagram as input data for our generator. Use cases can easily be graphically defined using an UML tool. UML tools are supposed to export models as XMI (*XML Metadata Interchange Format*) which is an XML vocabulary for representing models. Thus, a simple function may translate the XMI into the defined XML format or directly into the *Python* data structure. And this may represent your first steps towards *Model Driven Development*.

5.6 Testing generated code

In practice, generated code is not so compact. The reason for its compactness in this paper is to avoid the filling of complete pages of code excerpts. In the meantime, blank lines and non significant white spaces may become a nightmare for testing generated code. The `assertEqual` function tests for exactly equal strings, not for strings having the same meaning. There are solutions for improving one's life with testing generated code, we are going to mention two of them.

[11]The qualifier “user friendly” has not been used as *Python* code is user friendly.

[12]Which is definitely not “user friendly” from the author point of view.

First, you can write a function that compares two code snippets or write a class that represents code (on the basis of a string) and redefine the `__eq__` operator. Second, another solution is to compile the code strings using the `codeop` module [6] that permits the dynamic compilation of *Python* source code in memory and the comparison of the resulting code objects (see Figure 24). This last solution has the advantage to also check the code is correct (which should be, as we already wrote it using TDD).

```
import codeop
def has_same_meaning(code1, code2):
   '''Do the two python code strings have the same meaning?'''
   return codeop.compile_command(code1) == codeop.compile_command(code2)
```

Figure 24: Function for comparing the meaning of two code strings

These two solutions may not be substitutable: when you want to test incomplete code (a slice of a class or so) or non *Python* code the second solution is not usable, when you want to test complete *Python* code (a function, a class, or a module) I prefer the second one (so much easier). However there may exist situations when it would not be applicable.

The generated code presented in this section does not directly reflect the examples presented in previous ones: The hierarchy of use cases is not taken into account. This paper main goal is to be a presentation / teaching support, certainly not a real world product (even if it has been used to develop the application that manages the master students technical projects for two years).

6 Third step: TDD generated

For the moment, the generator we have produced only generates skeletons of the application various pieces. Developing the application behavior still means the writing of code (hand-written one). As hand-written code should be written after its unit tests, the developer still have to write tests. In order to ease your life, skeletons of the tests should also be generated. This defines a kind of road map of all the expected test cases to be defined and to pass.

6.1 Generation of the unit test skeletons

In the same way we have defined the generator on the basis of expected code, we can extend it to generate the skeletons of unit tests on the basis of the tests we wrote to begin this paper (see section 3 and section 4).

The approach is very similar to the generation of the application skeletons: we have to write first the test of the generator and then the generator itself (tests that test tests). The oracle for the generation of tests is extracted from the tests we wrote earlier. Figure 25 presents the expected test skeleton for controllers. For each operation of each controller class, a test operation is defined.

You may wonder why the addition of the test case in the suite is commented out. Following Kent Beck principle *one test at a time*, all the test should not be run at first. So, we start by commenting all the tests, and the developer will uncomment them one at a time, doing the *TDD loop* (*Write Test / Implement*

/ Refactor) and never leaving a test broken. In the meantime, the list of tests represents a road map so it is better to specify them all, the uncommenting of tests represents a progress bar of the development process.

```
test_ctrl_test = '''
import unittest
import controller
class TestAnyUserController(unittest.TestCase):
    def setUp(self):
        pass
    def tearDown(self):
        pass
    def test_list_projects(self):
        self.fail('Not Implemented')
def suite():
    suite = unittest.TestSuite()
    # suite.addTest(unittest.makeSuite(TestAnyUserController))
    return suite
if __name__ == '__main__':
    unittest.TextTestRunner(verbosity=2).run(suite())
'''
```

Figure 25: Test case oracle for the generation of controller test skeletons

6.2 Using the generator

Using code generation in a project implies the definition of a development process that is a little different from the Write Test / Implement / Refactor. Generators can be used in different ways, mostly depending on the generator itself (purpose, abilities, and so on).

Code generation as we tackle it in this paper represents a bootstrap of the application development process. The development process associated to our generator is the following one: Steps 1 and 2 are performed only once at the beginning, then steps 3 to 5 represent an iteration of development. The choice of operations may vary from implementing all the controller operations first then all the view operations to implementing both the controller and the view for an operation then moving to the next operation. The second approach sounds better as it provides at the end of each iteration a working (thus demonstrable) version of the application, getting feedback on your work as fast as possible.

1. Define the model of the application using the input format.
2. Generate the test and implementation skeletons of the application.
3. Write a test for an operation of the application.
4. Implement the operation associated to the test.
5. Refactor if necessary.

A more interesting approach would be to extend the cycle 3-5 to a cycle 1-5 (the complete development process is then iterative). You may wonder (like the author does) how a developer can have a complete model of her/his application in a single shot. However, this second approach is also a little bit more complicated to implement. A simple first step towards this solution is to organize the code in a more fine-grained fashion: one file per class and one package instead of a module for each part of the application. Nevertheless, this would not be enough: It would also be necessary to be able to add an operation to an existing class without breaking the class (some of its operations may have already be implemented). Adding operations at the end of existing classes could be a first option (not that good if you try it for real).

Warning You have to remember when using code generation that running the generator may take less than a second to destroy hours of work. If you re-generate code for a project when hand written code has been produced, you may loose the code you have written (for example the controller module implementation). Sometimes, you can clearly separate generated code from hand written one, sometimes you cannot. If it is not possible to separate the two kinds of source code, do not allow the (re)generation of the application on top of an existing one (that may have already been partly completed). This is the reason for which the existence of the folder to store the file is not checked, if the folder exist then the generation fails. So you can be sure not to loose work, and you have to wonder first: "Should I remove the folder and its content or not?"

7 Conclusion

This paper has introduced *Test-Driven Code Generation* (TDCG). TDCG can be seen as hints for the development as well as the integration of code generators in a *Test-Driven Development* (TDD) context. Using a simple web application as an example, we have underlined two key aspects of TDCG: (a) The writing using TDD of examples of the code to be generated in order to define oracles for the generator tests, and (b) the generation of test skeletons in order to apply TDD when completing the application (considering that the generated code represents only part of the whole application code).

Currently, we have put the focus on the basis of *Test-Driven Code Generation.* The organization of the generated code does not really permits to develop iteratively. In the examples, the various generated classes are organized in one module per layer (model, controller, view). Once a use case has been completed, generating the next one will overwrite the code you hand-wrote. Ideally, you may want to generate the code skeleton for a first use case, implement the missing code for this use case, validate your development on this functional small part of the application, then going to the next use case. To support iterative development, we mentioned that the three layers should be considered as packages and each use case class should be stored in a distinct module. Then, use case could be developed one by one with customer validation for each iteration.

The second expected improvement of the solution presented in this paper is to go beyond generating skeletons of the various classes. Producing several web applications will permits you to identify some common behaviors that could also be generated. For example, we have mentioned four categories of interactions: input forms, output forms, input / output forms, and search forms. Providing more information for each use case regarding these categories may allow you to generate (at least part) of the class behavior. Providing more *meaning* at the description level improves the amount of code you can generate. Then, it is up to you to find the balance between generate-able code, hand written code, and the cost of writing a generator compared to the cost of writing the code by hand. You will certainly never generate 100% of your application code. Reaching 90+% of generated code is quite common. Most of the time, the last percents are the exceptional cases that happen rarely or for which providing enough description is not really easier than hand-writing the code (not mentioning the cost of implementing the generator itself).

Finally, having multiple generators—like a distinct generator for each layer—is another interesting improvement. Having one big tool that does everything is not very modular nor flexible. Having several small specific tools that can be composed to achieve a task is better. For example, other generators could be written for the production of *Tkinter* view classes (or any other graphical library you like), and multiple generators could be written for several persistent frameworks. Then, once the application description is defined, you choose the graphical and persistent frameworks you want and use the appropriate generators in order to produce the flavor(s) of the application you prefer.

References

[1] K. Beck. *Test-Driven Development by example.* Addison-Wesley, 2002. *ISBN: 0-321-14653-0.*

[2] K. Beck. *Extreme Programming Explained, Embrace Change.* Addison-Wesley, second edition, 2005. *ISBN: 0-321-27865-8.*

[3] Cheetah, The Python-Powered Template Engine. Web site. `http://www.cheetahtemplate.org/`.

[4] CherryPy is a pythonic, object-oriented web development framework. Web site. `http://www.cherrypy.org/`.

[5] A. Cockburn. *Agile Software Development.* Addison Wesley, 2002. *ISBN: 0-201-69969-9.*

[6] `codeop` – Compile Python code. Web site. `http://docs.python.org/lib/module-codeop.html`.

[7] Django, The web framework for perfectionists with dealines. Web site. `http://www.djangoproject.com/`.

[8] `doctest` – Test interactive Python examples. Web site. `http://docs.python.org/lib/module-doctest.html`.

[9] M. Fowler. *Refactoring, improving the design of existing code.* Addison-Wesley, 1999. *ISBN: 0-201-48567-2.*

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Westley Professional Computing, USA, 1995. *ISBN: 0-201-63361-2.*

[11] `httplib` – HTTP protocol client. Web site. `http://docs.python.org/lib/module-httplib.html`.

[12] A. Hunt and D. Thomas. *The Pragmatic Programmer, From Journeyman to Master.* Addison Wesley, 2000. *ISBN: 0-201-61622-X.*

[13] T. Ohno. *Toyota Production System: Beyond Large-Scale Production.* Productivity Press, 1988. *ISBN: 0-915-29914-3.*

[14] `pMock` is a Python module for testing Python code using mock objects. Web site. `http://pmock.sourceforge.net/`.

[15] M. Poppendieck and T. Poppendieck. *Lean Software Development, An Agile Toolkit.* Addison Wesley, 2003. *ISBN: 0-321-15078-3.*

[16] The `py.test` tool and library. Web site. `http://codespeak.net/py/current/doc/test.html`.

[17] SQLObject is a popular Object Relational Manager for providing an object interface to your database. Web site. `http://sqlobject.org/`.

[18] TurboGears is the rapid web development megaframework you've been looking for. Web site. `http://www.turbogears.org`.

[19] `unittest` – Unit testing framework. Web site. `http://docs.python.org/lib/module-unittest.html`.