Test Driven Code Generation
(A Quick Introduction)

`Raphael.Marvie@lifl.fr`

LIFL – University of Lille (France)

Geneva – July 4th, 2006

EuroPython 2006

Motivations

Software Quality

√ Test Driven Development

- ▶ Clean code that works
- ▶ Flow of small validated steps
- ▶ Untested software is nothing

√ Automation

- ▶ Avoid repetitive tasks (error prone)
- ▶ Focus on the non trivial aspects (the value)
- ▶ Guarantee software consistency

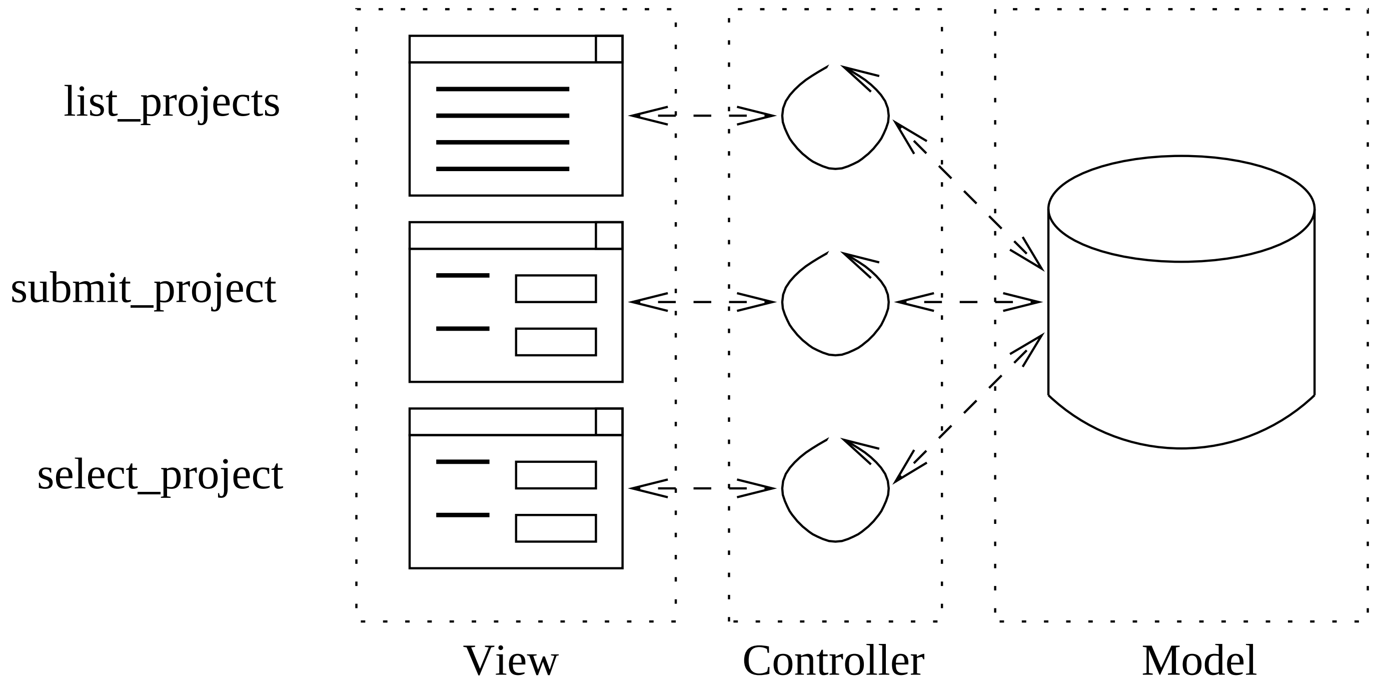
Productivity

- √ Test Driven Development
  - ▶ Reduce waste
  - ▶ Reduce bug tracking time
  - ▶ Non regression insurance
- √ Automation
  - ▶ 100 $\times$ 5 minutes is one working day
  - ▶ Code, builds, tests, documentation, web site, etc.
  - ▶ Automated tasks do not forget

TDD and Automation

- √ Scope : automation of code production
  - ▶ Only one aspect but an important one
  - ▶ Use of code generation
- √ How to get them work together ?
  - ▶ How to build a code generator using TDD ?
  - ▶ How to test drive generated code (and when) ?

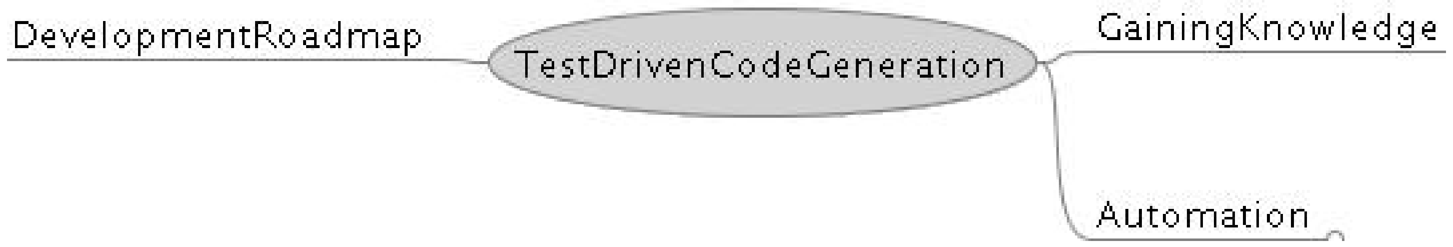
Application Example



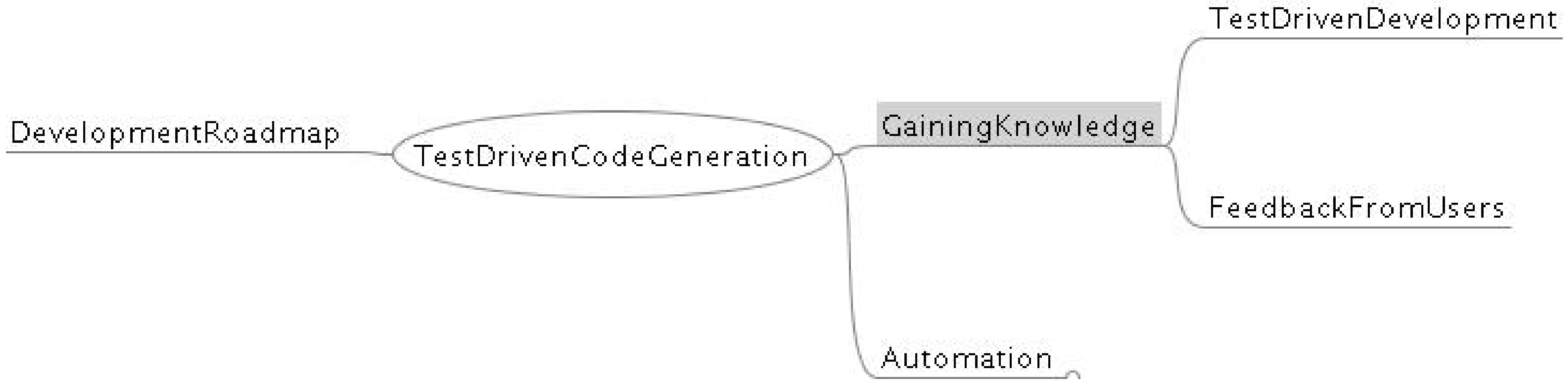
- MVC based web application
- CherryPy, Cheetah, and `unittest`

Principles and implementations

Principles


DevelopmentRoadmap
TestDrivenCodeGeneration
GainingKnowledge
Automation

Gaining Knowledge about the System



Test Driven Development

√ K.Beck TDD Loop (or TDD mantra)

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.

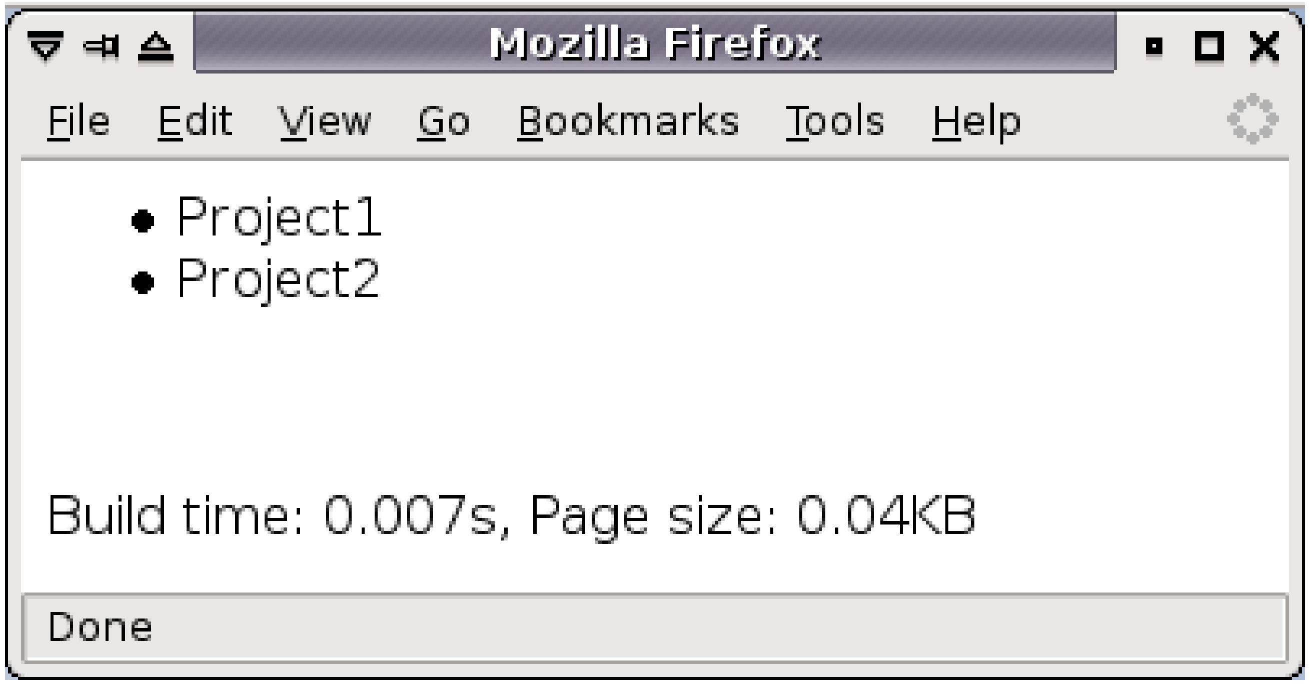
Write a test

```
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.assertEqual(expected_list_projects,
                         self.view.list_projects())
```

Make the Test Pass

```
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)
```

Get Feedback from Users



- Project1
- Project2

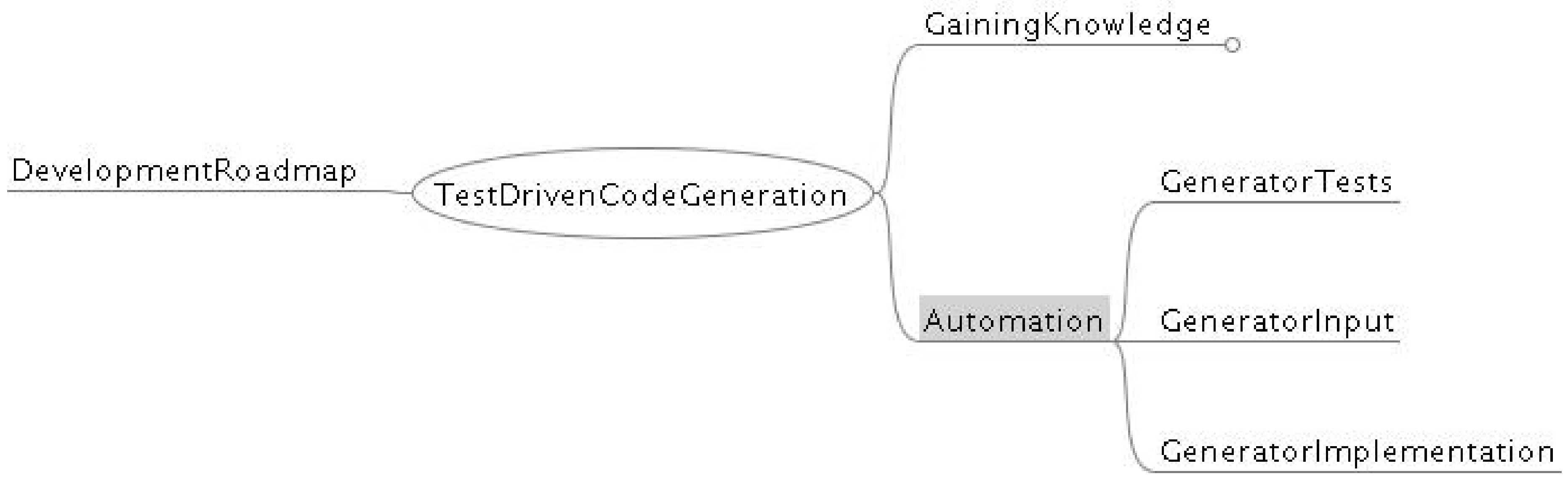
Build time: 0.007s, Page size: 0.04KB

Done

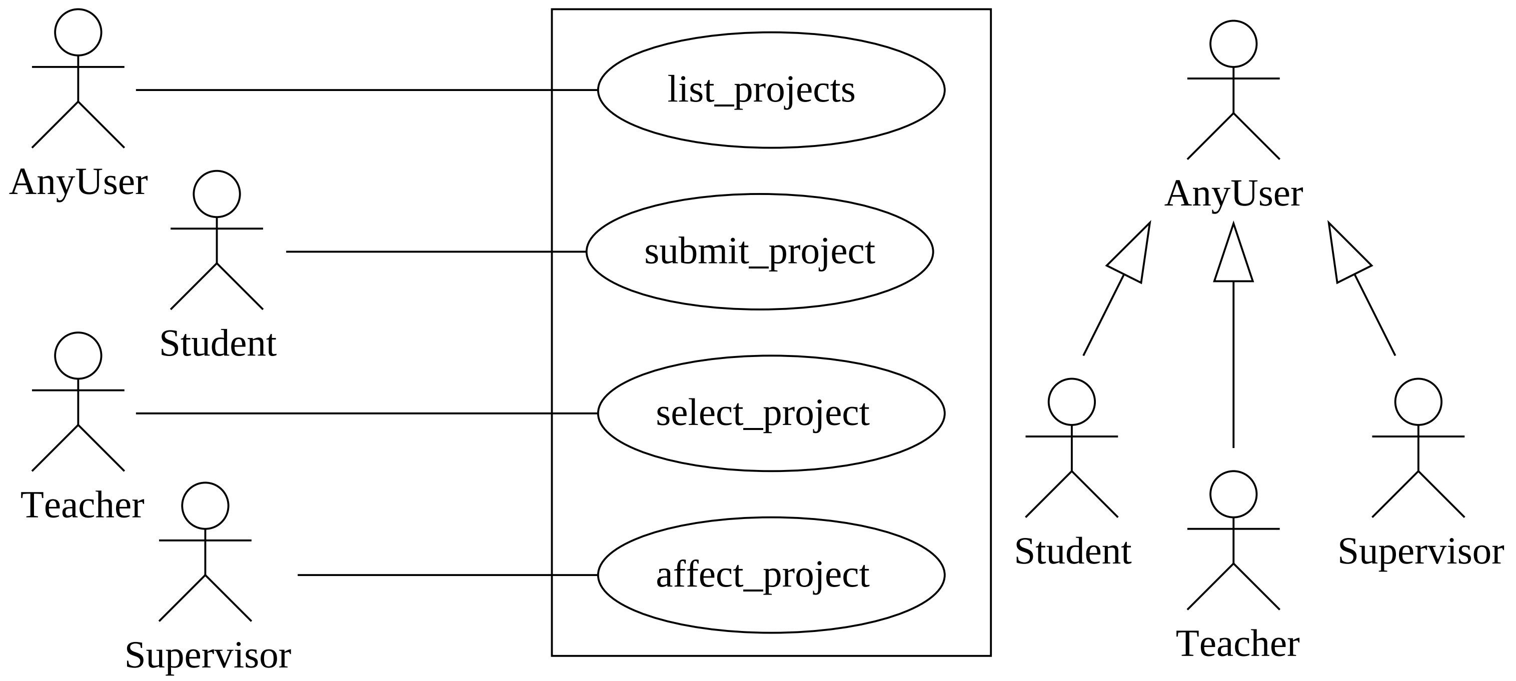
√ Simple demo application that works

√ Fancy details are for later

Automation


DevelopmentRoadmap
TestDrivenCodeGeneration
GainingKnowledge
Automation
GeneratorTests
GeneratorInput
GeneratorImplementation

Define the Generator Input



√ Use Cases of the application

Define the Generator Test (i)

```
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'
'''
```

√ Skeletons of the view implementations

Define the Generator Test (ii)

```
class TestViewGenerator(unittest.TestCase):
    def setUp(self):
        self.generator = generator.ViewGenerator()
    def test_generate(self):
        model = {'AnyUser': ['list_projects']}
        self.assertEqual(self.generator.generate(model),
                         view_test)
```

- √ Very similar to a controller test
- √ Tests can also be refactored

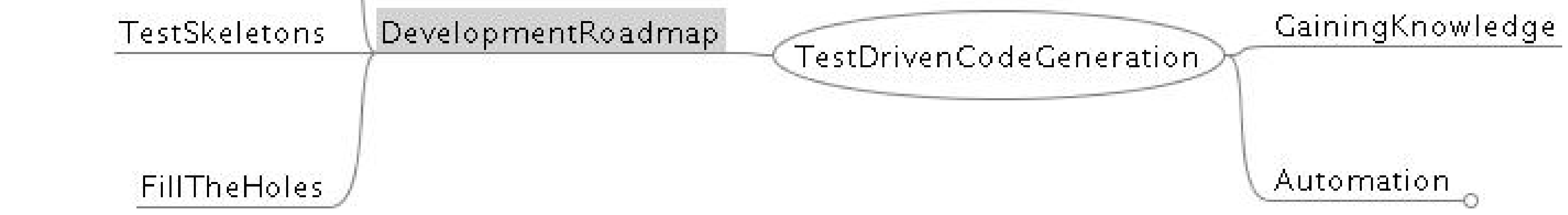
Implement the Generator (i)

```
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'
```

Implement the Generator (ii)

```
  #end for
#end for
'''
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)
```

Development Roadmap



Generating Test Skeletons

```
import unittest
import controller
class TestAnyUserView(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())
```

Conclusion

Double TDD Loop

1. Have some working code.
   (a) Write the test for a small part of the application
   (b) Implement the selected part of the application
   (c) Refactor if necessary
2. Extract the parts that can be generated & define the input
3. Develop the piece of the generator for this part
   (a) Write the test for the generation of this small part of the application.
   (b) Implement the part of the generator that produces this small part of the application.
   (c) Refactor the generator if necessary.

That’s all folks !