Test–Driven Development

Course of
Software Engineering II
A.A. 2009/2010

Valerio Maggio, Ph.D. Student
Prof. Sergio Di Martino
Contents at Glance

· What is TDD?

· TDD and XP

· TDD Mantra

· TDD Principles and Patterns
1. Example Scenario
Software Development as a Learning Process

- One must learn by doing the thing; for though you think you know it, you have no certainty until you try

Sofocle
Software Development as a Learning Process

- Almost all projects attempts *something* new
- *Something* refers to
  - People involved
  - Technology involved
  - Application Domain
  - … (most likely) a combination of these
- For Customers and End-Users?
  - Experience is worse!
Software Development as a Learning Process

- Every one involved has to learn as the projects progresses
  - Resolve misunderstanding along the way
- There will be changes!!

Anticipate unanticipated changes
Feedback is a fundamental tool

• Team needs cycle of activities
  • Add new feature
  • Gets feedback about quality and quantity of work already done!

• Time Boxes

• Incremental and Iterative Development
  • Incremental: Dev. *feature by feature*
  • Iterative: improvement of features in response to feedback
Practices that support changes

1. Constant testing to catch regression errors
   a. Add new feature without *fear*
   b. Frequent manual testing is infeasible!!

2. Keep the code as simple as possible
   a. More time spent in reading code than writing it

3. *Simplicity* takes effort so...
   a. REFACTO!R!
2. Test Driven Development

TDD

ALL CODE IS GUILTY UNTIL PROVEN INNOCENT

CODESMACK
Test Driven Development

- We write tests *before* we write the code

- Testing as a *Design Activity*

- Testing to clarify ideas about *what* we want the code to do!
What is TDD?

- Test-Driven Development
- Test-First Programming
- Test-Driven Design
What is TDD?

- Iterative and incremental software development

- TDD objective is to DESIGN CODE and not to VALIDATE Code
  - *Design to fail* principle
TDD and Agile

- TDD concepts combines
  - Test First Programming
  - Refactoring

- TDD is an *agile practice*
- XP is an *agile methodology*
TDD and XP

- Core of XP
- No needs of others XP practices

- Avoid *software regression*
  - *Anticipate changes*

- *Product code smarter that works better*

- Reduce the presence of bugs and errors
  - “*You have nothing to lose but your bugs*”
3. TDD and Unit Testing
Unit test

- “Unit tests run fast. If they don't run fast they're not unit tests.”

- A test is not a *unit test* if:
  - communicate with DB
  - communicate with networking services
  - cannot be executed in parallel with other unit tests
Unit Test and TDD

- Testing code is released together with production code
- A feature is released only if
  - Has at least a Unit test
  - All of its unit tests pass
- Do changes without *fear*
  - *Refactoring*
- Reduce debugging
Unit Test and TDD

- Unit Tests overcome dependencies
  - How?
  - Stubs and Mock Objects
- www.mockobjects.com
-Mocks simulate interactions with real objects
  - Unit tests can continue to run fast...
  - ... but?
- Too many setup operations are bad!
Example

DBConnection.java

```java
public interface DBConnection {
    void connect();
    void close();
}
```

FakeDBConnection.java

```java
public class FakeDBConnection implements DBConnection {
    private boolean connected = false;
    private boolean closed = false;
    public void connect() {connected = true;}
    public void close() {closed = true;}
    public boolean validate(){return connected && closed;}
}
```
4. TDD Mantra
TDD Mantra

First step

Think

Think : step by step

Think about what we want the code to do
“We want to develop an innovative arithmetic library that handles only non-negative numbers”

aritLib.py
Red Bar: Writing tests

Think about the behavior of the class and its public interface
import aritLib
import unittest

class AritLibTest(unittest.TestCase):
    knownValues = ((0,0,0),(1,1,2),(2,3,5),(-1,-1,-1),(-10,10,-1),(10,-5,-1),)
    def testSum(self):
        for x, y, sum in self.knownValues:
            result = aritLib.add(x, y)
            self.assertEquals(sum, result)
class AritLibTest(unittest.TestCase):
  knownValues = ((0,0,0),(1,1,2),(2,3,5),(-1,-1,-1),(-10,10,-1),(10,-5,-1),)
  def testSum(self):
    for x, y, sum in self.knownValues:
      result = aritLib.add(x, y)
      self.assertEquals(sum, result)

ERROR: testAdd (__main__.AritLibTest)
----------------------------------------------
Traceback (most recent call last):
  File "AritLibTest.py", line 11, in testAdd
    result = aritLib.add(x,y)
AttributeError: 'module' object has no attribute 'add'
----------------------------------------------
Ran 1 test in 0.000s
FAILED (errors=1)
TDD Mantra

Third step

Think → Red bar → Green Bar

Failed Test

Green Bar : writing production code.

Write **ONLY** production code to pass previous test
TDD Mantra

Third step

Think → Red bar → Green Bar

Failed Test

Green Bar : writing production code.

aritLib.py

def add(x, y):
    if x < 0:
        return -1
    if y < 0:
        return -1
    return x+y
TDD Mantra

Third step

Think → Red bar → Green Bar

Failed Test

Green Bar: writing production code.

aritLib.py

```python
def add(x, y):
    if x < 0:
        return -1
    if y < 0:
        return -1
    return x+y
```

Ran 1 test in 0.000 s
OK

Run previous tests without modifications
TDD Mantra

Fourth step

Think → Red bar → Green Bar → Refactoring

Failed Test

**Refactoring:** refactor developed feature

During refactoring we DO NOT have to modify semantic of developed feature!!
TDD Mantra

**Fourth step**

**Think** → **Red bar** → **Green Bar** → **Refactoring**

Before:
```python
def add(x, y):
    if x < 0:
        return -1
    if y < 0:
        return -1
    return x+y
```

After:
```python
def add(x, y):
    if x < 0 or y < 0:
        return -1
    return x+y
```
TDD Mantra

Fouth step

Think → Red bar → Green Bar → Refactoring

Failed Test

After

```python
def add(x, y):
    if x < 0 or y < 0:
        return -1
    return x+y
```

Ran 1 test in 0.000 s
OK

Run previous tests without modifications
Principles

- Code once, test twice
- Clean code that works

- KISS: Keep It Short & Simple
- YAGNI: You Ain’t Gonna Need It
- DRY: Don't repeat yourself
Banana Spelling?

• “I can spell banana but I never know when to stop”

WHEN TO STOP?

• When code works
• When all tests are done
• When there's no duplicated code
Bad smells …

There's something wrong when:

• It is necessary to test *private* and/or *protected* methods.

• We need *white box testing*.

• We need to configure system before run tests.

• Tests run intermittently.

• Tests run slowly.
“Unit tests run fast. If they don’t run fast, they aren’t unit tests.”

Testing Speed is important because:

- If the tests were not fast then they would be a distraction.
- If the tests were not fast then it would not run with high frequency
  - Benefit of the TDD?
5. TDD Patterns

- **RED**: test fails
- **GREEN**: test passes
- **REFACTOR**

Core of TDD

LD CLEAN Code + Tests
TDD Patterns

Red Bar patterns:

• Begin with a simple test.
• If you have a new idea
  • add it to the test list
  • stay on what you're doing.
• Add a test for any faults found.
• If you can not go on throw it all away and change it.
TDD Patterns

Testing patterns:

• If the test takes too long to work then divide it into simpler parts.
• If tests need some complex objects then use mock objects.
• Store execution log of tests
• If you work alone leave the last test of the day broken
• If you work in a team leave ever tests running.
TDD Patterns

Green Bar patterns:

- Writing the easier code to pass the test.
- Write the simpler implementation to pass current test
- If an operation has to work on collections
  - write the first implementation on a single object
  - then generalizes.
Test Readability

- Test names describe features

```java
public class TargetObjectTest {
    @Test public void test1() { [...]
    @Test public void test2() { [...]
    @Test public void test3() { [...]
}

public class TargetObjectTest {
    @Test public boolean isReady() { [...]
    @Test public void choose(Picker picker) { [...]
```
6. Focused Integration Testing and eEnd2End Testing
Three types of unit tests

- Functional unit testing
- Integration unit testing
- Logic unit testing
Focused integration testing

A focused integration test is focused on testing:

- communication with the database
- network communication
- communication with the filesystem
- communication with external objects
Focused integration testing (2)

- If you need a lot of integration tests then there's something wrong.

- Ex. If all the business objects speak directly with the database then the code have not a good design!

- The code that talks too much with the outside world is neither very cohesive nor well coupled.
End-to-end Testing

*Used to test the whole system:*

Test of whole stories using the system, the GUI user to the database ...

**Cons:**

- Difficult to accomplish.
- Difficult to set.
- Difficult to detect errors.
- Very slow.
- Not automated.
Execution speed of tests

- Unit tests
  - one hundred per second.
- Focused integration tests
  - ten per second.
- End-to-end tests
  - several seconds for each test.
7. TDD and Legacy Code
Legacy Code

“Legacy code is code without tests”

Problems:

- Lack of documentation
- Difficult to understand in depth
- It is not designed thinking of "testability"
Legacy Code (2)

Steps to address the legacy code:

• Start typing tests to see if the legacy code (a part of) was well understood.

• Fit the test until it works well.

• What code has been tested?

• What areas need testing?

• What are the risks of the code?
8. Conclusions
Social Implications

- TDD handles “the fears” during software development

- Fears has a lot of negative aspects:
  - makes it uncertain
  - removes the desire to communicate
  - makes it wary of the feedback
  - makes nervous
Social Implications (2)

- TDD handles the "fears" during development:
  - New (small) release only if the code has exceeded 100% of the test set.
  - The design goes hand in hand with development.
  - TDD allows programmers to perfectly know the code.
TDD Benefits

- It keeps the code simple
- Rapid development
- The tests are both design and documentation
- Easy to understand code
- Bugs found early in development
- Less debugging
- Low cost of change
TDD Limits

- High learning curve
- Managers are reluctant to apply
- Requires great discipline
- Difficult to implement the GUI
- Difficult to apply to Legacy Code