introducing unit tests

Sven Gorts is a passionate Java developer and enthousiastic promoter of agile development principles. In his working environment Sven advocates pair programming and test driven development. Within object technology his personal intrests are design patterns and refactoring.

Hans Keppens is part of the EMC BDG team that develops an automated acceptance testing framework for the Centera product. Hans is mainly interested in efficient and productive teams, and in the 'tools/practices' that help teams reach that goal, like pair programming, test-driven development, short releases, simple design, collective code ownership, continuous integration, ...

Agenda

- Unit Testing
- Bringing Code Under Test
- Characteristics & Compromises
- Case Study: Critical Dependencies
- Tips & Tricks

Unit Testing

Definition: *a **unit test** is a test, coded in software, that verifies the functionality of an individual software component in isolation of the overall system.*

So we'll be talking about unit tests. But exactly what is a unit test ?

Loosely defined: a unit test is a test, coded in software, that verifies the functionality of an individual software component in isolation of the overal system.

Benefits Of Unit Testing

- Rapid Feedback
  - problems are spotted earlier
  - pinpointing bugs and regressions
- Automation
  - run frequently at low cost
- Confidence
  - measured quality

As eXtreme Programmers, Test Driven Developers, Agilists we know the benefits:

•Our unit tests help us to gain rapid feedback, problems can be spotted earlier. Tests are also a great help help for pinpointing bugs and regressions. A decent suite of unit tests can save lots of debugging time.

•Unit tests can be automated, thereby increasing their value. Once we have an automated testSuite we can run them over and over at low cost.

•Since unit tests are a form of measured quality we can be confident in our work. With unit testing the risk of latest hour surprises is low.

A Forgotten Best Practice ?

- Unit testing is not a new practice, in fact its a rather old one
- Its importance has long been ignored by most of the software industry
- Unit testing practice recently gained popularity because of its promotion in XP and other agile methods

In a way unit testing has long been a forgotten best practice of the software industry. In fact it's an old practice that despite its benefits has long been ignored by most of the software industry. In recent years unit testing has regained popularity because of its promotion in XP and other agile methods.

Things Are Changing

- xUnit framework has been ported to most popular programming languages
  - CppUnit, NUnit, PyUnit, JUnit, and many more
- We've seen the publication of the first books on unit testing
  - TDD by example, JUnit Recipes, xUnit Patterns, ...
- Nowadays most developers know what unit tests are, some write unit tests daily

=> Things Are Really Taking Off

But things are changing:

•The xUnit framework has been ported too most popular programming languages.

•We now have CppUnit, NUnit PyUnit, JUnit, and many more

•We've also seen the first good books on unit testing:

• *"Test Driven Development By Example"* (Kent Beck)

• *"JUnit Recipes"* (JB Rainsberger)

•upcoming *"xUnit Patterns"* (Gerard Meszaros)

•Most developers today now what unit tests are. Unit testing has found its way to the mainstream development practices. So things are really taking off.

So What Is The Problem ?

=> Does your project have unit tests ?

- Existing code doesn't have tests
- New development without tests

=> Instant legacy

- What can we do about it ?

So why this session ? What's the problem ?

Allow me to answer that with the following question:

Does your project have unit tests ?

In practice many teams still don't have unit tests and the problem is often twofold:

•Many existing code does not have unit tests

•New development is often done without tests

Instant legacy

As a result many teams find themselves trapped within a *"Legacy Culture"*. Code is often hard to comprehend and difficult to test and as a result new development proceeds without writing tests. Most of the team members fear change and prefer not to fix what ain't broke in order to avoid regressions. As a result these teams are missing all the benefits of agile development.

Bringing Code Under Test

```
public class AlarmHandler {
   ...
  public void addAlarm(String alarm)      { ... }

  public void removeAlarm(String alarm) { ... }

  public boolean isActive(String alarm)    { ... }
   ...
}
```

- Apparently no tests available
- Before making changes write some tests
  - Learn how the code currently behaves
  - Lock down current behavior

We've been asked to work on the system's AlarmHandler class today. A quick peek at the source learns that it's not so bad but apparently there are no unit tests available.

We're new to this code so before making any changes to the code we we want to be confident that we don't break anything. By writing some unit tests for the AlarmHandler class we can learn how the code currently behaves, and at the same time lock down the current behavior of the code.

Writing The First Test

```
public class AlarmHandlerTest extends TestCase {

  public void testAddAlarmSetsAlarmStatus() {



  }

}
```

- We start with a classic unit test
  - Create test class, extend TestCase
  - Set up a test method

For our first test let's start with something simple:

•add an alarm to the alarmhandler and verify that its status is set.

Setting Up The Fixture

```
public class AlarmHandlerTest extends TestCase {

  public void testAddAlarmSetsAlarmStatus() {
    AlarmHandler handler = new AlarmHandler();
    handler.addAlarm("red alert");


  }

}
```

- Create instance
- Add alarm

So we create an AlarmHandler instance, set the alarm

Validating The Result

```
public class AlarmHandlerTest extends TestCase {

  public void testAddAlarmSetsAlarmStatus() {
    AlarmHandler handler = new AlarmHandler();
    handler.addAlarm("red alert");

    assertTrue(handler.isActive("red alert"));
  }

}
```

- Add assert to check activation

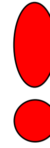
  =>Seems like a good test, let's run it

And our final step is to add an assert to verify the alarm is really activated.

Well this seems like a good test. Let's run it.

Red Bar, Missing Config File

```
public void testAddAlarmSetsAlarmStatus() {
  AlarmHandler handler = new AlarmHandler();
  handler.addAlarm("red alert");

  assertTrue(handler.isActive("red alert"));
}
```



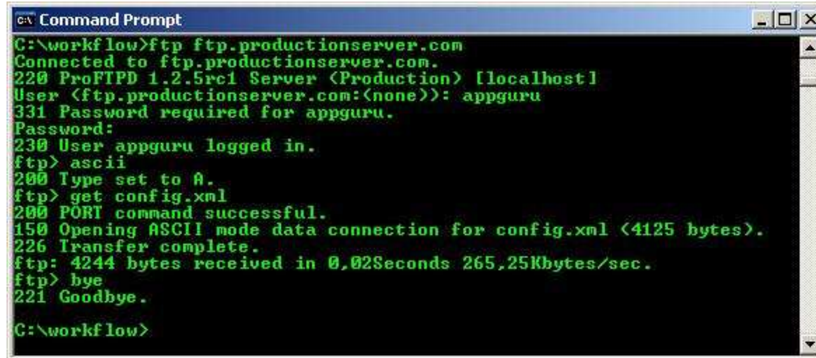
```
java.io.FileNotFoundException: c:\workflow\config.xml (The system cannot find the path specified)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:106)
  at java.io.FileInputStream.<init>(FileInputStream.java:66)
  at be.legacy.project.ConfigManager.<init>(ConfigManager.java:20)
  at be.legacy.project.ConfigManager.getInstance(ConfigManager.java:34)
  at be.legacy.project.MessagingEngine.init(MessagingEngine.java:11)
  at be.legacy.project.AlarmHandler.<init>(AlarmHandler.java:5)
  at be.legacy.project.AlarmHandlerTest.testAddAlarmSetsAlarmStatus(AlarmHandlerTest.java:8)
```

- Why do we need a configuration file ?

So, we've run our test and surprisingly we got a red bar. Well that's a bit strange. But apparently an exception was thrown, lets have a closer look at the console output.

We may wonder why instantiating an AlarmHandler requires a configuration file. As we can see from the traces, instantiating an AlarmHandler implicitly starts some other things, one of them being the ConfigManager singleton that requires the presence of a config.xml file.

Getting The Config File



- We can use the config.xml from the server
- Let's see whether this makes the bar green

Okay, we've *'borrowed'* the config.xml file from the server and downloaded it to the workflow folder. Let's run our test again to see whether this does the trick.

Red Bar, Database Not Running

```
public void testAddAlarmSetsAlarmStatus() {
  AlarmHandler handler = new AlarmHandler();
  handler.addAlarm("red alert");

  assertTrue(handler.isActive("red alert"));
}
```

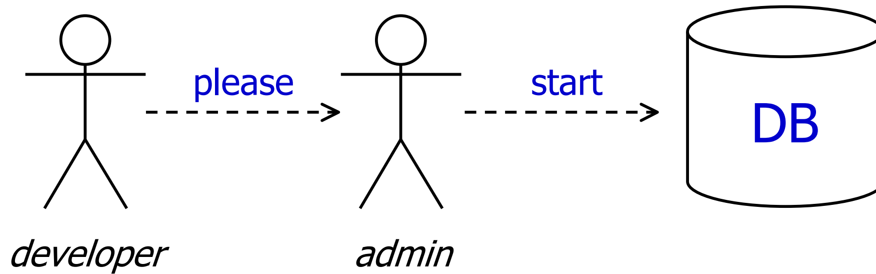
```
java.sql.SQLException: Connection refused: connect
  at oracle.jdbc.dbaccess.DBError.check_error(DBError.java:203)
  at oracle.jdbc.driver.OracleConnection.<init>(OracleConnection.java:100)
  at oracle.jdbc.driver.OracleDriver.connect(OracleDriver.java:146)
  at java.sql.DriverManager.getConnection(DriverManager.java:512)
  at java.sql.DriverManager.getConnection(DriverManager.java:171)
  at be.legacy.project.MessagingEngine.init(MessagingEngine.java:15)
  at be.legacy.project.AlarmHandler.<init>(AlarmHandler.java:5)
  at be.legacy.project.AlarmHandlerTest.testAddAlarmSetsAlarmStatus(AlarmHandlerTest.java:8)
```

- Sigh, what’s wrong this time ?

And again we have a red bar. What’s wrong this time ?

Apparently our AlarmHandler makes use of MessagingEngine which needs access to the database.

Start Database

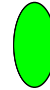
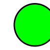


- We ask our admin to start the database
- And give our test another try

We hurry to our database administrator and ask to start the database.

Green Bar, Finally

```
public void testAddAlarmSetsAlarmStatus() {
  AlarmHandler handler = new AlarmHandler();
  handler.addAlarm("red alert");

  assertTrue(handler.isActive("red alert"));
}
```

- Green bar, hurray

Mini Retrospective

Writing a unit test turns out to be difficult

- We spend almost twenty minutes only to get one test up and running
- We needed to set up a configuration file, start a database, ...

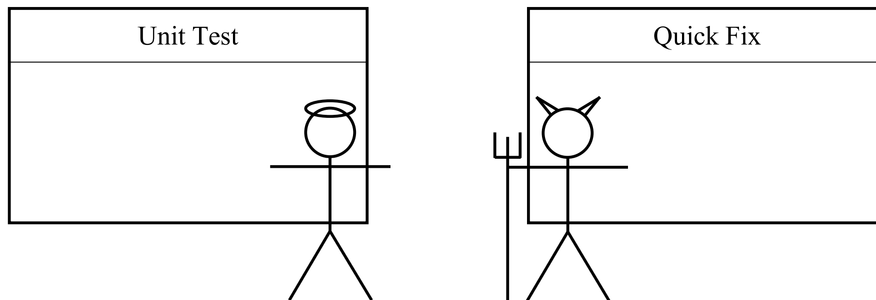
=> Let's see why we face such difficulties

We spend more than twenty minutes just to get one test up and running

Things certainly went harder than expected:
we had to set up a configuration file, start up the database

Need to lower our expectations of what our tests will look like

Characteristics and Compromises



What Makes a Good Unit Test ?

self containing

100 % green

side effect free

fast

fine-grained

reliable

self documenting

Let's first have a look at the characteristics of a good unit test.

We expect tests to be:

- self containing:
  each test is capable to prepare its fixture
- 100% green
  we want our source to be production quality all of the time
- side-effect free:
  tests whether passing or failing may not affect other tests
- fast:
  performant tests allows us to run them often
- fine-grained:
  focus on single component, method / class
- reliable:
  free of false negatives / false positives
- self-documenting
  tests act as a form of executable documentation

Realizing Ideals Can Be Hard

- Unit testing is complicated by
  - Global variables, singletons
  - Dependencies upon concrete classes
  - Hardcoded filenames, directory structures
  - Databases, external processes
  - OS environment, native calls
  - ...

•presence of global variables or singletons

•Dependencies upon concrete classes, rather than interfaces

•hard coded filenames, directory structures, configuration data

•databases, external processes

•os environment, native calls

The bubble chart gives an overview of common testing difficulties encountered when bringing an existing code base under test.

When preparing for this session we tried to order them according to priority. But soon we realized that which of them is more important will depend upon the actual development environment.

During development the relative weight of each of these problems will change:

•When there are only few tests in place, we may accept the fact that they're slow, however, as the suite of tests grows, the speed of the tests may become more critical.

no tests available: surely can't refactor to make code more testable

nonrepeatable test: an external side effect caused by a test is not cleared

interdependent test: tests depend upon each others side effects

broken test: some tests are found broken, should we fix or delete ?

slow tests: the suite of tests takes to long to run them often

unstable test: test may pass or fail unpredictably

resource/environment independent test: can't test without the availability of external resource

half automated test: tests are unable to automatically perform their setup so manual intervention is required

Progress First

=> We need to be realistic !

- We don't have tests yet
- We're not able to write perfect tests
- Usable tests now, good tests later

=> Let's see how this works

With all the testing difficulties encountered in the first code example it's quite obvious that we can't live up to our testing ideals from day one.

Sure, we want our test to be: self containing, side-effect free and the like but we need to be realistic: So far we don't have tests yet and it's unlikely that we're able to write ideal tests from the start.

What we need are usable tests:

•Tests we can write in a reasonable amount of time

•Tests that give us a good return of investment

So our goal is to get tests in place while keeping the balance between effort and gain

Case Study: Critical Dependencies

Definition: *a **critical dependency** is a dependency upon an external library or resource, for which availability in the development environment cannot be guaranteed.*

- *Database connections*
- *Dos or unix shell commands*
- *Sockets, external protocols*
- *Native calls*

Examples of critical dependencies can be:

•Database connections
If our tests need access to the database in order to run we may not be able to run our tests in case the database is down or when working from a location from which the database is not reachable.

•Dos or unix shell commands.
If our tests require operating system calls to be executed and our code contains unix or linux shell commands our tests may not be runnable from a mac or pc.

•Sockets, external protocols
If our tests have to set up a socket to call to external processes for which no dummy or stub is available we may not be able to run our tests.

•Native calls
If our tests make calls to native libraries that are only available on-target, for instance in an embedded environment, our tests won't run from our development machine.

Hardware.execute() 1 / 13

HardwareController depends upon Hardware class *(critical dependency)*

- For some reason it is not available

```
public class HardwareController {
  ...

  void process() {
    String command = null;
      ...
    if ( !active && ctr2Set )
      command = "incr ctr2;";

    Hardware.execute( command );
      ...
  }

  ...
}
```

Our basic problem with bringing the HardwareController class under test is that by calling Hardware.execute() it has a critical dependency upon the Hardware class.

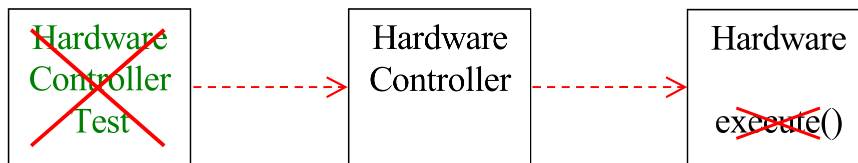
Unfortunately, for some reason, the resource required to run Hardware.execute in our development environment is not available.

Some reasons:

•Hardware.execute() is intended to be run in an embedded environment and there's no supporting library available to allow it to run in the development environment.

•Hardware.execute() calls to an expensive external resource that needs to be shared by other team members, including non-developers.

Class Diagram 2 / 13



- Code is difficult to test
  - Due to the transitive dependency upon the Hardware resource
- Before we start refactoring
  - First goal: bring the code under test

The HardwareController class is difficult to test because of its direct depencency upon the Hardware class.

By transitivity, the critical dependency embedded in the Hardware class prevents unit testing the HardwareController without the availability of the underlying Hardware resource.

Extract execute() Method 3 / 13

- Extract Method execute()
- Minimize impact: we don't have any tests in place yet
- Method serves as substitution point to hook in behavior for unit tests

```
public class HardwareController {
  ...
  void process() {
    String command = null;
     ...
    if ( !active && ctr2Set )
      command = "incr cntr2;";

    execute(command);
     ...
  }

  public void execute(String command) {
    Hardware.execute( command );
  }
     ...
}
```

By extracting an execute method that hides the direct call to Hardware.execute() we have encapsulated the critical dependency upon the Hardware class.

We need to perform this refactoring really carefully because we don't have any unit tests in place yet .

The extracted execute method will serve as a substitution point where we can hook in the behavior for our unit tests.

Subclass To Test 4 / 13

```
public class HardwareControllerTester extends HardwareController {
  private String _command;

  public void execute(String command) {
    _command = command;
  }

  public String lastCommand() {
    return _command;
  }
}
```

- Extends HardwareController
- Overrides execute()
- Stores incoming commands

Now that we have extracted the execute() method in the HardwareController class, we can subclass HardwareController with a HardwareControllerTester in order to evesdrop incoming commands.

The implementation of the HardwareControllerTester is rather straightforward:

•we extend HardwareController, and override the execute() method

•we keep a field to store incoming commands

sidenote: **Testing the measurement tool**

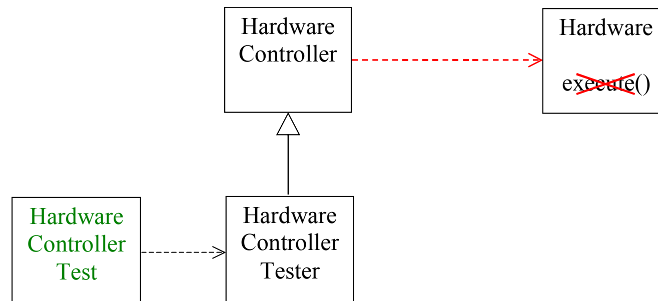
Combination of execute() and lastCommand() can be unittested itself. We can think of this as calibrating our measurement tool.

HardwareControllerTest 5 / 13

```
public class HardwareControllerTest extends TestCase {

  public void testProcessSendsIncrCntr2Command() {
    HardwareControllerTester controller = new HardwareControllerTester();

    controller.process();

    assertEquals("incr cntr2;", controller.lastCommand());
  }
}
```

- Using the HardwareControllerTester we're able to write a test that indirectly tests the HardwareController class

Critical Dependencies 6 / 13



- HardwareController is tested indirectly
- We now have our unit test(s) in place
- So we can refactor

Extract Executor Interface 7 / 13

```
public interface Executor {
  void execute(String command);
}


public class HardwareController implements Executor {
  ...

  public void execute(String command) {
    Hardware.execute( command );
  }

  ...
}
```

...

- Executor specifies execute behavior

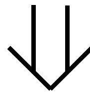
Self Delegate 8 / 13

```
public class HardwareController implements Executor {
  …
  Executor executor = this;
  …

  void process() {
…   String command = null;
    …
    if ( !active && ctr2Set )
      command = "incr cntr2;";

    executor.execute(command);
    …
  }
}
```

- HardwareController now delegates to an executor, using itself as component

HardwareExecutor 9 / 13

```
public class HardwareController implements Executor {
  ...
  public void execute(String command) {
    Hardware.execute( command );
  }
  ...
}
```



```
public class HardwareExecutor implements Executor {
  public void execute(String command) {
    Hardware.execute( command );
  }
}
```

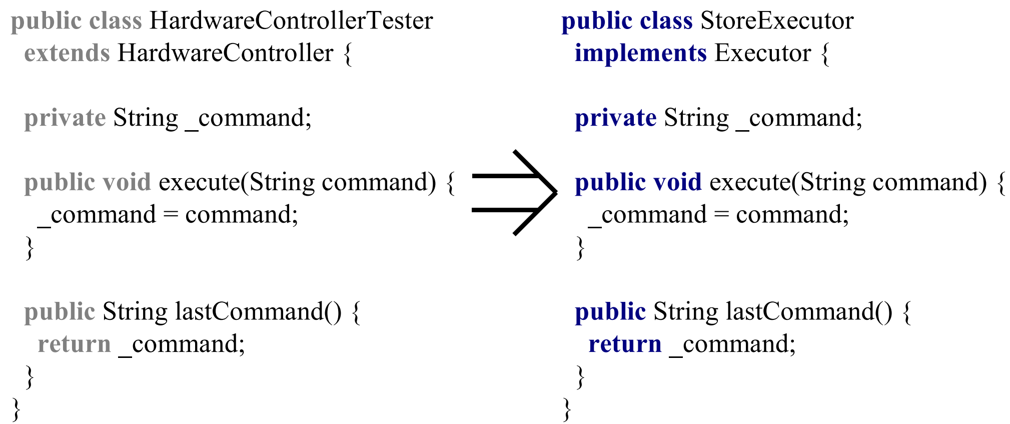
- HardwareExecutor implementation can be copied from HardwareController class

Parameterize Constructor 10 / 13

```
public class HardwareController implements Executor {
 …

 private final Executor executor;

 HardwareController( Executor executor ) {
   this.executor = executor;
 }

 …
}
```

- We generalize HardwareController so that it can accept any kind of Executor

Upgrade To StoreExecutor 11 / 13



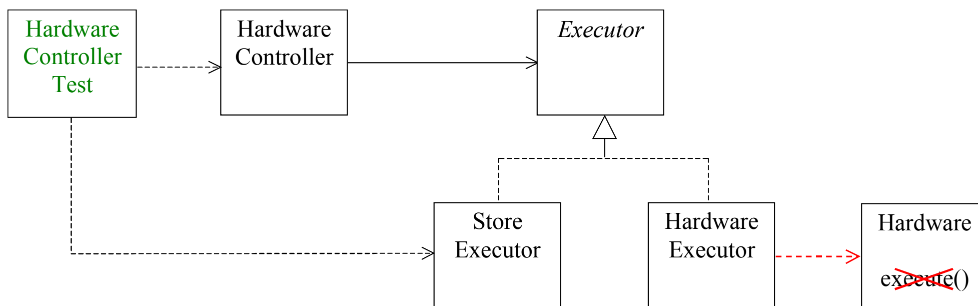
- We now upgrade HardWareController-Tester to a StoreExecutor class

Update Test Code 12 / 13

```
public class HardwareControllerTest extends TestCase {

 public void testProcessSendsIncrCntr2Command() {
   StoreExecutor storeExecutor = new StoreExecutor();
   HardwareController controller = new HardwareController(storeExecutor);

   controller.process();

   assertEquals("incr cntr2;", storeExecutor.lastCommand());
 }
}
```

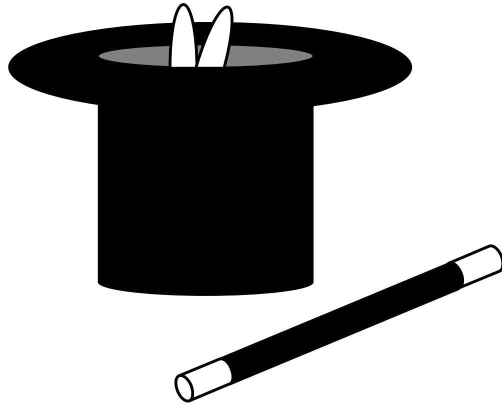
- And we update our unit test

Class Diagram 13 / 13



- The resulting design
  - HardwareController is unit-tested
  - Test independent from hardware

Tips & Tricks



Troubled By Singletons

- Singletons make testing difficult:
  - Unintentional side effects
  - Lack of substitution point
- But there's a trick: resetInstance()

```
static void resetInstance() {
  manager = null;
}
```

Especially the presence of singletons in a design can make testing difficult.

There are several reasons for this:

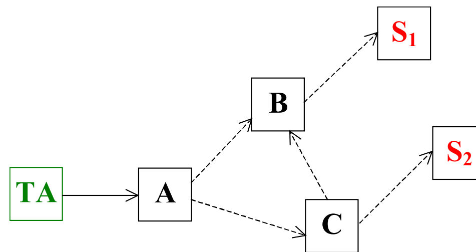
•singletons are glorified globals
=> unintentional side effects

•singletons serve as single access point
=> lack of substitution point

•attractor to all kinds of dependencies
=> it seems to be a kind of natural law in legacy systems that singletons
have to be dependent upon configuration files, databases or other kind
of dependencies.

When singletons are getting in the way of our unit tests creating a `resetInstance()` method often makes a good trick to get rid of most of the side-effects. Calls to `resetInstance()` certainly aren't the cleanest solution, especially since singletons are typically being depended on by classes from various packages. Verry likely reset `resetInstance()` method will have to be made public at some point. Yet this approach is often a good starting point because is enables us to bring more of our code under test.

Troubled By Singletons



- To write a reliable, repeatable test for A
  - call resetInstance() on S1 and S2
  - do this for setUp() and tearDown()

In order to write a reliable and repeatable test for class A:

•We may need to call resetInstance() on both S1 and S1 since either of them could have unintentional side-effects to our tests.

•For starters, its a good idea to do call resetInstance() both from within the setUp() and the teardown()

> •In the *setup()* the idea is to have a clearly defined fixture. Otherwise we can't be sure that we get a correctly initialized instance by the time our test is run.
>
> •For the *tearDown()* the idea is that each test is responsible for cleaning up its own side-effects. If not other tests might accidentally run or break as a result of a previous test.

File Dependent Instances

- Problem: instantiation depends upon a hardcoded configuration file
- Solution: set up the configuration file from within your test

```
protected void setUp() throws Exception {
  Properties properties = new Properties();
  properties.put("serverIp","192.168.4.88");
  properties.put("serverPort","3688");
    ... more properties ...
  properties.store(new FileOutputStream("C:/workflow/config.properties"),"");

  _controller = new FlowController();        /* requires hardcoded file */
}
```

•sometimes difficulty is dependency upon hardcoded paths or filenames

•manually setting them up isn't enough (tests are not self-containing)

•a first step can be to write the configuration file from within the test

•we can now run this test repeatedly.

File Dependent Instances

- Problem: tests run slow due to excessive file access
- Solution: parameterize the constructor to allow configuring the test instance from within the test

```
protected void setUp () throws Exception {
    Properties properties = new Properties();
    properties.put("serverIp","192.168.4.88");
    properties.put("serverPort","3688");
     ... more properties ...

    _controller = new FlowController(properties);   /* now accepts properties */
  }
```

•if many tests use the same file we could try to write the file once

•however, a better solution is to deal with the file dependency

•doing so will often significantly speed up the unit tests

Commented Code

- Smell: code differs between development environment and integration environment and is being commented in and out

```
{
    ...
    stubbedCode();

    // productionCode();
    ...
}
```

development environment

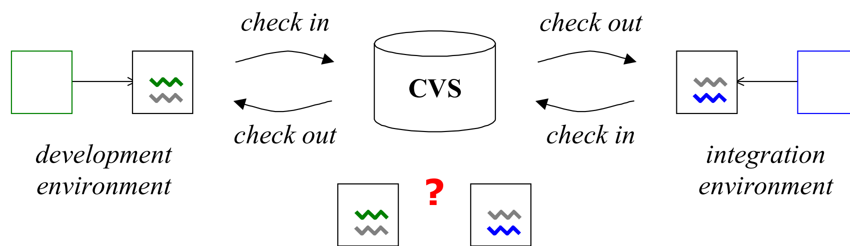
```
{
    ...
    // stubbedCode();

    productionCode();
    ...
}
```

integration environment

In some environments developers start the early iterations writing their components against a stubbed version of the code, transitioning to the production version when the project gets nearby integration. During this period its often seen that the production version of the code and the stubbed version of the code is being commented in and out. Running directly with the production version from the code is often problematic because the production code hides many dependencies underneat.

Commented Code

- Problem: code version that lives in the source code repository is always wrong



Even though commenting out a bit of code may seem innocent its practice is problematic in a project setting. From the development environment developers can run their tests against the stubbed version, however running agains the productioncode breaks the tests. In the integration environment the production code is required do meaningfull testing. Anyhow the practice of commenting in and out source code leads to a version in the source code repository that is always wrong: either the unit tests or acceptance tests will fail.

Introducing A Testing Flag

```
private boolean _testing;

{
  ...
  if ( _testing ) {
    stubbedCode();
  } else {
    productionCode();
  }
  ...
}

void setTesting() {
  _testing = true;
}
```

- Solution:
  introduce a testing flag
- Far from ideal, but sufficient to keep our tests running
- It's a start, not the end

We can take the first step towards an improved situation by introducing a testing flag to distinguish between the stubbed code and production code. From our unit tests we can call the setTesting() method to ensure the stubbed version is called.

Sure, the obtained solution is still far from ideal. We have production code being mixed with test code and some extra code clutter due to the conditional, but at least we can keep our tests running while refactoring towards a better solution.

Conclusion

- If there are no tests today
  => You can write some tomorrow
- Code can be made testable
- Using small steps
  - Test a little
  - Refactor a little

Discussion

- Questions ?

References

<www.refactoring.be>

- Working Effectively With Legacy Code
  (Michael Feathers)
- xUnit Patterns: Refactoring Test Code
  (Gerard Meszaros)