Test Driven Development in Practice

Course Materials

Gojko Adzic

Test Driven Development in Practice: Course Materials

Gojko Adzic

Neuri Limited
25 Southampton Buildings
London WC2A 1AL
United Kingdom
+44(0)2079932982
http://neuri.co.uk/training

Version 2010-10-29

About the author

Gojko runs Neuri Ltd, a UK based consultancy that helps ambitious teams, from web startups to large financial institutions, implement test driven development, specification by example and agile testing practices.

Gojko is the author of several popular printed and online guides on acceptance testing, including *Bridging the Communication gap*, *Test Driven .NET Development with Fitnesse* and *Getting Fit with .NET*, and more than 200 articles about programming, operating systems, the Internet and new technologies published in various online and print magazines. He is the primary contributor to the DbFIT database testing library which is used by banks, insurance companies and bookmakers worldwide.

To get in touch, write to gojko@neuri.com or visit http://gojko.net.

Introduction

Before the rise of test-driven development, testing and coding were traditionally two separate activities. Programmers would write code and forget about it. Quality Assurance people tried to flush out as many bugs as they could before the release. From time to time, the software would be so bug-ridden that the QA engineers had to pull the plug. Sometimes things got even worse: a system would be delivered still full of bugs and customers would besiege the support staff with angry calls and e-mails. Problems never started out big, but they were allowed to grow between coding and testing.

TDD was a conceptual shift from this practice, spreading testing over the entire development process. Problems are not allowed to grow. Guided by the "test early, test often" principle, we do a bit more work up front, but that significantly reduces the effort required to support the code.

The following quotation from "Competing on the Basis of Speed",[1] a presentation given by Mary Poppendieck at Google Tech Talks on the 15th of December 2006, summarises the benefits of TDD very effectively:

When we started up in our plant...

...we had people in QA who used to try to find defects in our products, and we moved them all out on to the production line to figure out how to make stuff without defects in the first place...You will be amazed at how much faster you go when you make stuff and defects are caught when they occur instead of being found later.

TDD allows us to work amazingly fast, because it improves the whole process in several ways.

[1] **http://video.google.com/videoplay?docid=-5105910452864283694**

Quality from the start

TDD keeps problems small. Because tests take place early in the development process, rather than late, problems surface quickly and get solved before they grow. We can build quality into our products right from the start.

Early interface validation

The only way to know if an API makes sense is to use it. TDD makes developers eat their own dog food because tests are effectively the first API client. If the API does not make sense or if it is hard to use, the developers experience this first hand. Tests should be easy to write, and if they are not, then we need to change the code to make testing easier. This makes it easier for other people to use our code.

Divide and conquer

In order to test code modules in isolation, developers have to divide them into small independent chunks. This leads to better interfaces, clear division of responsibility, and easier management of code.

Safety net for the code

At the beginning of development, changes to software are quick and simple. As the code base grows, it becomes harder to modify. A simple change in one area often causes problems in a seemingly unrelated part of the code. Without tests, it soon becomes too hard and expensive to change anything. When we test all parts of the code, problems are quickly identified wherever they are.

Confidence = Productivity

As craftsmen, most developers take pride in their work and want to deliver quality software. Making changes to production code when we are not confident in the quality feels like walking on broken glass. Having tests tell us that we are on the right track, allows us to be more confident in our work and enables us to change the code faster.

Light at the end of the tunnel

Tests can be an effective way to describe specifications and requirements. If used properly, they can guide the development process, showing us what we need to implement. Once all tests pass, the work is done. This light at the end of the tunnel makes it easier to focus on the development effort.

Applying TDD to our project

Test-driven development is just a set of simple practices supported by a few lightweight tools, most of which are free. On the surface, these practices and tools aim to improve the quality of software with more rigorous and more efficient code checking. However, that is just the tip of the iceberg. There is much more to TDD then verifying code. As the name suggests, in test-driven development, tests play a proactive, leading role in the development process, rather than the reactive role of checking software after it is written.

Guiding the development

A strict implementation of a feature with TDD consists of three stages:

1. **Write a test for the feature and write just enough code to make the test compile and run, expecting it to fail the first time.**
2. **Change the underlying code until the test passes.**

3. **Clean up the code, integrate it better with the rest of the system and repeat the tests to check that we have not have broken anything in the process.**

This sequence is often called "red-green-refactor" or "red bar-green bar-refactor". The name comes from the status icon or status bar in most GUI test runners, which is red when tests fail and green when they pass.

Once the first test passes, we write another test, write more code, make the new test run, clean up again and retest. After we have repeated this cycle for all the tests for a specific feature, our work on the feature is done and we can move on to the next feature. Robert C. Martin summarised this connection between tests and production code in his *Three Rules of TDD*:[2]

1. **You are not allowed to write any production code unless it is to make a failing unit test pass.**
2. **You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.**
3. **You are not allowed to write any more production code than is sufficient to pass the one failing unit test.**

This approach may seem too strict or radical at first. Additional tests are often written after the first version of the code has been developed. Some people write the code first, then do the tests and modify the code until all the tests pass. This technique also works, as long as the tests are written in the same development cycle as the code, and extra care is taken to focus on the user stories, not the code.

Think about the intention, not the implementation

A test is pointless as a quality check for the code if it just replicates what the code does. Any bugs that found their way into the code will

[2] **http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd**

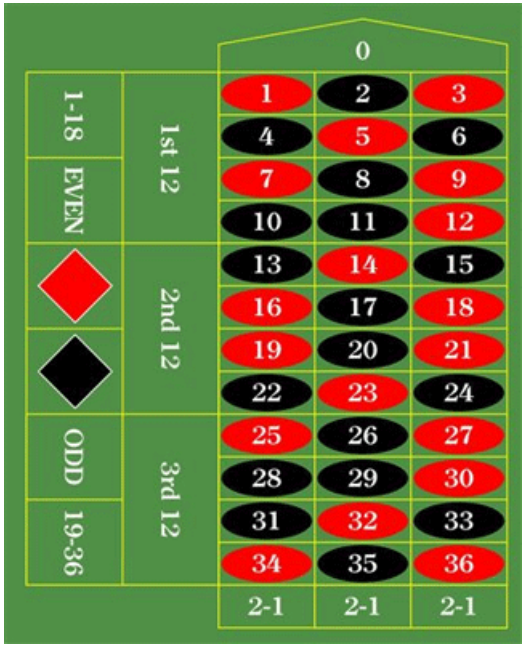
just be replicated in the test. Tests that just replicate code are very dangerous because they will give you a feeling of comfort, thinking that the code is tested properly, while bugs are waiting round the corner. Do not write the test by looking at the code, as you may get blinded by the implementation details. This is one of the reasons why it is better to write tests before the code, not after.

Tests that guide the development should always reflect the intention, not the implementation.

Part I. Exercises

Pretend it's magic: how would you use it?

The goal of this exercise is to make you think about object design from a perspective of a user, not from a perspective of a designer. So imagine this situation:



We are building a UI client for a roulette game, written by another team. The game should allow players to place bets, view and report on the winnings. To make it simple, we are not building in any fancy rules from the start. So here are our requirements:

1. **A player can place chips on number fields (0-36) to bet.**

2. **Bets can have different chip values. A player can place any number of chips on any number of fields.**
3. **The total number of chips is limited by the table and the table can refuse a bet if there are too many chips.**
4. **Up to eight players can join a game and place bets. The table might refuse to accept a new player if it is full - the system should display an error message to the player in that case.**
5. **Each player will be assigned a separate colour for chips when they join the table.**

Bonus requirements:

1. **After a bet is placed, the system will display the fields with chips, printing the colour and amount of chips next to each field. The chips should be ordered by field and colour. Different bets placed on the same field by the same player are displayed as one group of chips.**
2. **The system will calculate a percentage of the table covered by a player and display that after a bet is placed.**

Pretend it's magic and that the roulette game, developed by that other team, is a black-box with all the features that we need, exposed with the API that is really nice, logical and easy to use. You don't know and don't care how it works inside, it just works and it does all the heavy lifting (eg checking balances). We just need to integrate with it and use its functionality, handling input from players and output to the screen.

Your task is:

Demonstrate each of the requirements in pseudo-code, showing how you would use the API. The API can be anything you want.

Here are some ground rules:

- **You are not allowed to work alone - work in pairs**
- **You are not allowed to use a computer, especially not an IDE**

- **The pseudo-code should be similar to the programming language you use normally, but does not have to compile**
- **It should explain the examples in detail - you can leave some unimportant parts out but be very precise with key aspects**
- **The code should be as simple as possible - don't complicate it**
- **Remember that the API already has all the functionality. You just need to handle keyboard inputs and screen outputs. And to make things simple, we are working on a text UI, not fancy graphics.**

Don't turn the page

Pretend it's magic: part 2

Although the API is very nice, our project manager has some concern that the game developed by the other team does not follow the exact rules that we need.

Your next task is:

Modify the pseudo-code examples to describe how you would check that the code in the library actually follows the rules that we need. Replace console inputs with some reasonable values. Replace console output with a specification of post-conditions that should be true if the API works OK.

Growing code with unit tests (EX2)

The goal of this exercise is to learn how the red-green-refactor cycle works in practice and how to use basic unit testing tool features.

Your next task is:

- **Convert a pseudo-code example to a unit test.**
- **Implement the underlying domain code to make the tests pass.**
- **Select another pseudo-code example and repeat.**

Use the "Hello world" example in your IDE as a basic template to write unit tests.

Dealing with change (EX3)

The goal of this exercise is to demonstrate how unit tests help us evolve software when change requests come in. So our requirements are now changing. Instead of just placing bets on numbers, we should now support the following:

1. **Players can place bets on Odd or Even numbers.**
2. **Players can place "split" bets between any two neighbours. A bet placed between 1 and 2 covers both fields but counts as one bet. Split bets can only be placed on neighbours.**

Bonus features

1. **Players can place bets on any column by putting chips on the "2-1" field below the column.**
2. **Players can place "street" bets on any row (eg 1,2,3) by putting chips on the side next to the ending number.**
3. **Players can place bets on first half or second half by placing chips on 1-18 or 19-36 respectively**

Your next task is:

Describe the requirements using unit tests, then implement the code to make the tests pass.

Here are the rules:

- **You are not allowed to work alone - work in pairs but not the with same person that you have worked before**
- **You should write new unit tests to describe changes. You are not allowed to change any existing tests.**

- **You are not allowed to write any domain code unless in order to make a failing test pass**
- **Work on one requirement at a time - do not write tests to describe the next requirement until you have finished implementing the current one, confirmed by the relevant test passing.**
- **Old tests must keep passing.**

don't turn the page

Dealing with change - part 2 (EX3)

Now we can consolidate the code. Discuss with your partner how the API should change to make the code more consistent and elegant. Then implement the changes but following the rules:

1. **First change the tests to reflect the proposed API changes. Review and discuss whether that is what you wanted. The code does not have to compile at this step.**
2. **Change the API and the code to make the tests pass.**
3. **Work on one API change at a time. Don't change several things at once.**
4. **All tests should pass when you are done.**

Focusing Tests (EX4)

The goal of this exercise is to demonstrate focusing unit tests on code units that we are actually specifying. Our system now has the following new requirements:

1. **After all active players signal that they are done with placing bets, the system will start rolling the ball. The system should randomly choose a number between 0 and 36 (inclusive) where the ball will stop, to simulate rolling the ball on the roulette wheel.**
2. **Rolling isn't instantaneous (it normally takes 30-40 seconds), and a player can still place a bet up to ten seconds after the ball starts rolling.**
3. **The system should randomly choose a time for the ball to roll, between 30 and 40 seconds.**

Bonus features

1. **After the ball stops, the system determines winning bets. Bets on numbers win if the ball stops at that number. Odd/Even and other outside bets win if the ball stops in the correct range, but they all lose if the ball stops at 0.**

Your next task is:

Describe the new requirements using unit tests, then implement the code to make the tests pass.

You should already be familiar with the basic rules:

- **You are not allowed to work alone: work in pairs but not the with same person that you have worked before**
- **You should write new unit tests to describe changes. You are not allowed to change any existing tests.**

- **You are not allowed to write any domain code unless in order to make a failing test pass**
- **Work on one requirement at a time - do not write tests to describe the next requirement until you have finished implementing the current one, confirmed by the relevant test passing.**
- **Old tests must keep passing.**

In addition, for this exercise, think about what the requirement actually specifies and which part of the code is described by the requirement. In other words, *What are we testing/specifying here?*

don't turn the page

Focusing Tests, part 2 (EX4)

Discuss the following questions with your pair:

- **What makes the test hard to automate?**
- **Describe one relevant case as a realistic example**
- **What are we testing there (answer the following with Yes or No):**
  1. **System rolls the ball after all players place bets**
  2. **The ball falls on a random number between 0 and 36**
  3. **Chosen numbers are really random**
  4. **Players can place bets after the ball starts rolling, up to the cut-off time**
  5. **Wheel spinning really takes 30 seconds**
- **What would make the test easier to automate?**

Stubs in practice (EX5)

The purpose of this exercise is to learn the basic stub syntax. Your task for this exercise is:

Replace the custom built random number generator test stub with an out-of-the-box stub.

Use the "Hello world" example from your IDE as a guide.

Mocks in practice (EX6)

The goal of this exercise is to demonstrate how mock objects are used in practice. We now extend our system by introducing chip balances (amount of chips available to a player), and have two new requirements:

1. **The system will track chips available to a player and deduct any chips placed on the table from the available amount.**
2. **If a player attempts to place more chips on the table than available, the system refuses the bet.**

Bonus features

1. **When the ball stops rolling, the system will automatically pay the player for each individual winning bet, increasing his available balance. Numbers pay 1:36.**
2. **Bets placed on groups of numbers (neighbours, rows, columns) are treated as bets placed on individual numbers with proportionally reduced number of chips.**
3. **Bets placed on ODD, EVEN, RED, BLACK pay 1:2.**

Player chip balances are controlled using a "wallet service", so that we can plug in different implementations for different clients.

Your next task is:

Describe the requirements using unit tests to define the interface of wallet service, then implement the code to make the tests pass. Don't implement the wallet service interface yourself - use a test mock object instead.

You should already be familiar with the basic rules:

- **You are not allowed to work alone: work in pairs but not the with same person that you have worked before**
- **You should write new unit tests to describe changes. You are not allowed to change any existing tests.**
- **You are not allowed to write any domain code unless in order to make a failing test pass**
- **Work on one requirement at a time - do not write tests to describe the next requirement until you have finished implementing the current one, confirmed by the relevant test passing.**
- **Old tests must keep passing.**

What's wrong with this code? (EX7)

The purpose of this exercise is to to get you thinking about code improvements. In the exercise folder, you'll find a project with some existing code and tests. The code is very much like any other code that you might find on a project, grown from a simple code base with lots of feature changes.

Your next task is:

Look at the code and identify how it can be improved. Don't change any code yet - just make a list of things that you would like to change.

Here are some questions to get you started:

- **Is there something in particular that you don't like with the way that the code is written?**
- **If you were writing this from scratch, what would you change?**
- **Is the code easy to understand? If not, what would you do to improve that?**
- **Does any code belong somewhere else?**

Write down proposed changes and you'll present them later.

Clean-up (EX7)

The goal of this exercise is to demonstrate how good test coverage reduces cost of change and allows you to improve code design. Without tests, changes are not easy as we can break seemingly unaffected parts of the code. With tests, changes cost a lot less. So now that we have a list of proposed changes, let's clean the code up.

Your next task is:

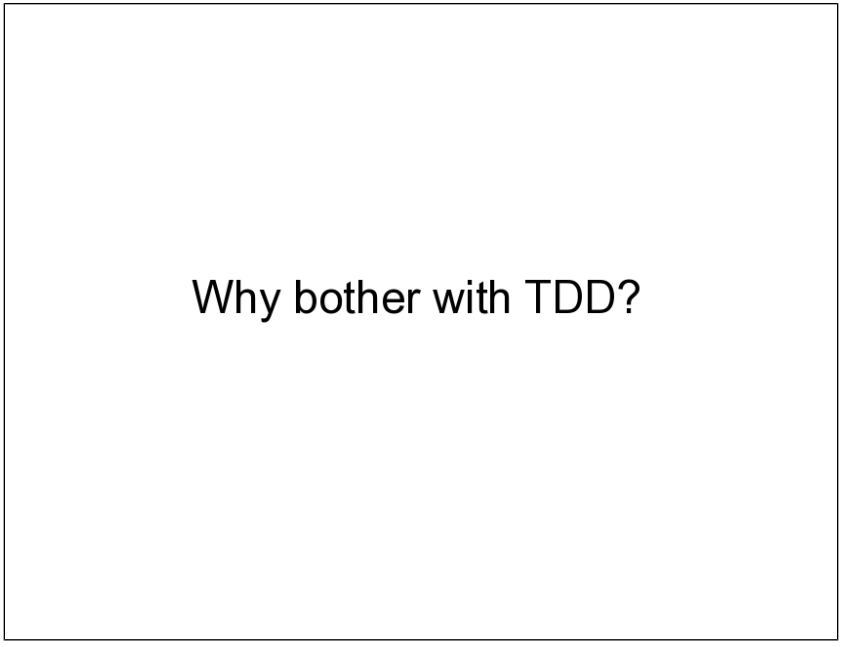
Implement the proposed changes to the code without breaking the functionality.

Here are the exercise rules:

- **Work in pairs**
- **Implement the changes in small steps, one by one**
- **If the change affects the API, you first have to modify the relevant test; changes that don't affect the API should not affect tests so you can skip this step.**
- **If the change involves moving code into a new class, you need to write unit tests for the new methods first**
- **After each change, verify that all tests pass and fix any problems that cause any tests to fail**
- **You are not allowed to disable or delete a test unless it is replaced by an equivalent test that reflects a proposed change**

Part II. Slides

Why bother with tdd?



When we started up in our plant...

...we had people in QA who used to try to find defects in our products, and we moved them all out on to the production line to figure out how to make stuff without defects in the first place...*You will be amazed at how much faster you go* when you make stuff and defects are caught when they occur instead of being found later.

Mary Poppendieck
Google Tech Talks 2006

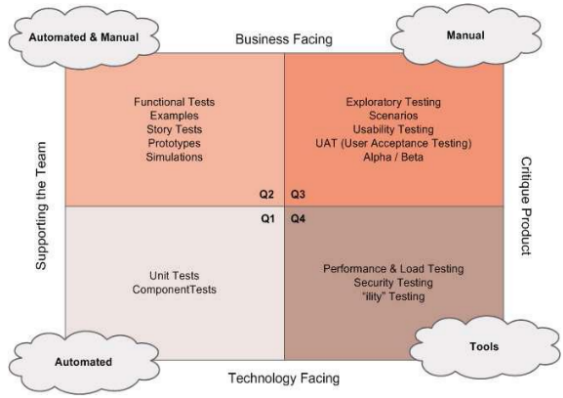
The Toyota Way: Eliminate Waste

TDD allows improves the whole process in several ways

- Quality from the start
- Early interface validation
- Divide and conquer
- Safety net for the code
- Better Confidence = Productivity
- Better focus for development

Test?

Tests in agile processes



- Agile testing quadrants idea by Brian Marick, picture by Janet Gregory

Red-green-refactor for success

Eliminate Waste

- Inspection to prevent defects is essential, inspection to discover them is waste
- Just-in-case code is the biggest source of waste in software development

Mary and Tom Poppendieck

Lean Software Development

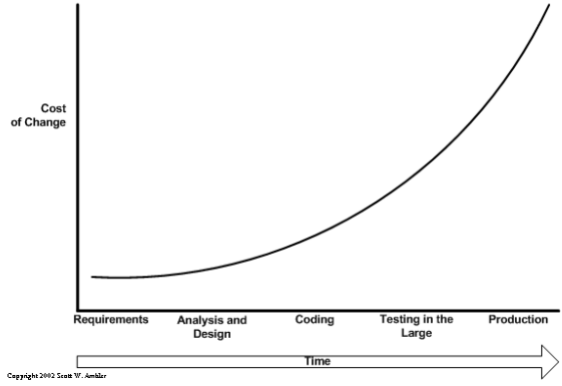
Two important XP principles

- KISS – Keep it Short and Simple (or keep it simple stupid)
- YAGNI - You Aren't Gonna Need It

Tests describe how we want to use the system

- So they describe what the system API is as well
  - Pseudo-code from the exercise gives you a good design of the solution,
  - fit for the purpose
  - no just-in-case methods
- *Problem: It only focuses on what we need now, there is no big picture!*

Cost of change in traditional development



What takes longer –
finding the problem or solving it?

Poka-yoke devices

- Inexpensive, frequent verifications
- Keep the cost of change relatively flat
- How do we make tests our poka-yoke devices?

Solution: Heavy automation

- Automated tests run quickly = inexpensively
- They can be executed often
- So if we break something, a test will tell us
- Unit test tools allow us to automate tests and speed up feedback

TDD Ideas

- Flesh out exactly what we need in the interface and no more
  - Describe features by writing code that uses the (not yet existing) target classes
- Explicitly state what we want it to do, so that we can check it later
  - Assert class and method functionality before we write them
- Make this verification very efficient
  - Automate it

Red-Green-Refactor

- Red – write a failing test
- Green – implement the test in the simplest possible way
- Refactor – change and redesign the code to improve it. Run the tests again to ensure that they pass.

Actually, has five steps!

- Think – what test will best move the code towards completion?
- Red
- Green
- Refactor
- Repeat – do it again, work in short cycles

James Shore

Three rules of TDD

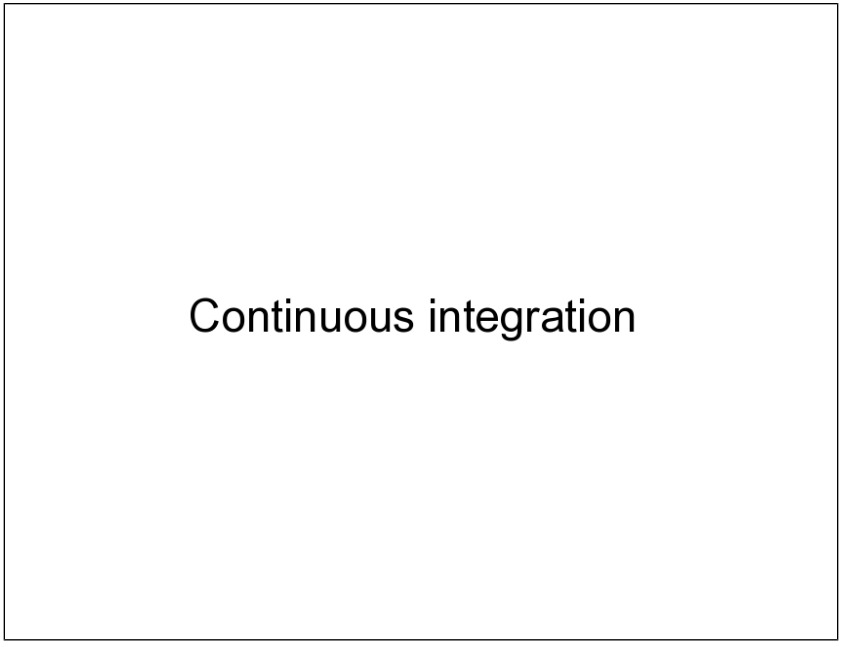
1. You are not allowed to write any production code unless it is to make a failing unit test pass.

2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.

3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Robert C. Martin

Continuous integration



Broken builds

- Teams make lots of concurrent changes, possible conflicts and bugs
  - Integration problems
  - Broken tests
  - Bugs
- 3rd party changes
- Software doesn't build = you cannot deploy it

Snowball effect

- Concurrent changes mean that pairs don't always have the latest code
- Small problems accumulate
  - Once they do, it's too late
- Fix broken windows
- Always keep software in shippable state

Keeping the system shippable

- Have a single version of truth
  - Keep software in central version control,
  - Check-in and integrate frequently
  - Build from that, make it repeatable - automate
- Make it self-validating
  - Execute tests/analytics as part of the build
- Speed up feedback
  - Run full build frequently
  - Alert on problems

Continuous integration

- Started as a practice to have one machine dedicated to integration where people would rotate to fix issues
- Automated with CruiseControl, now lots of other tools as well

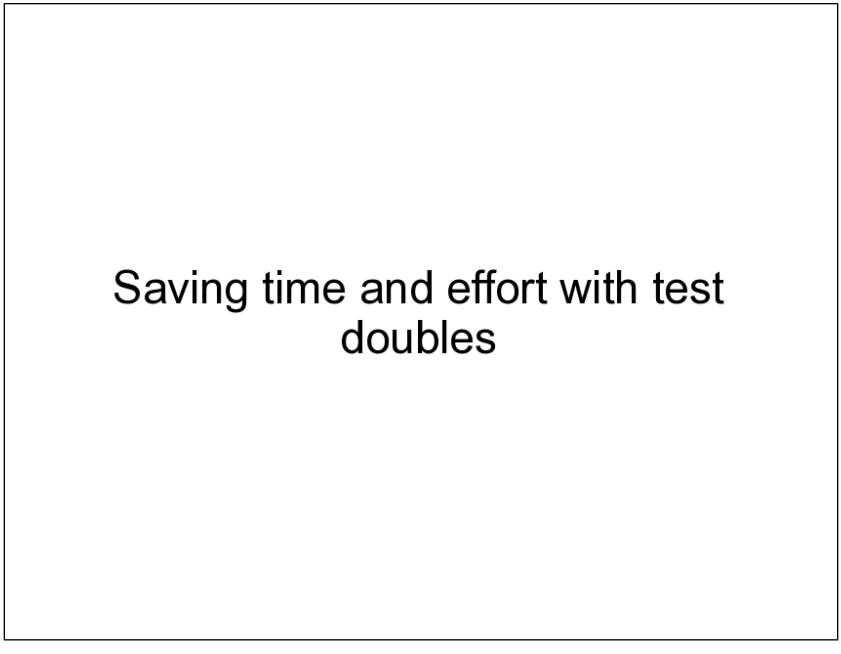
CI tools

- Automatically check out code from source control
- Build and rebuild dependent modules
- Execute tests and reports
- Alert in case of problems

Advantages of Continuous Integration

- Avoiding the snowball effect
- Early feedback
- Stable working environment
- Shippable software
- **“Triple reduction of project delivery time”**
  (http://www.accu-usa.org/Slides/accu_continuous_integration_best_practises.pdf)

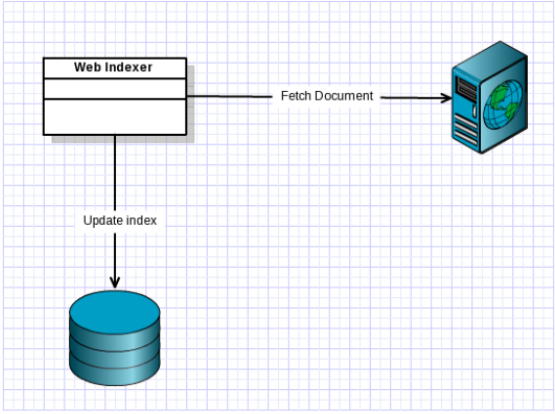
Saving time and effort with test doubles



Things that are not easy to automate cause problems for TDD

- Complex dependencies
  - services/objects that require complex setup
- Slow or unreliable
  - External systems
- Not easily repeatable
  - Databases, Timers, Random events

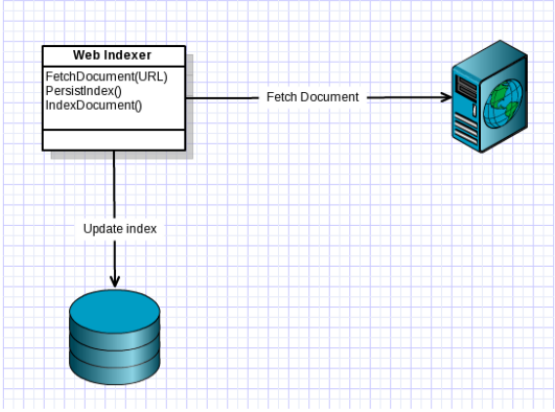
Web crawler (eg google)



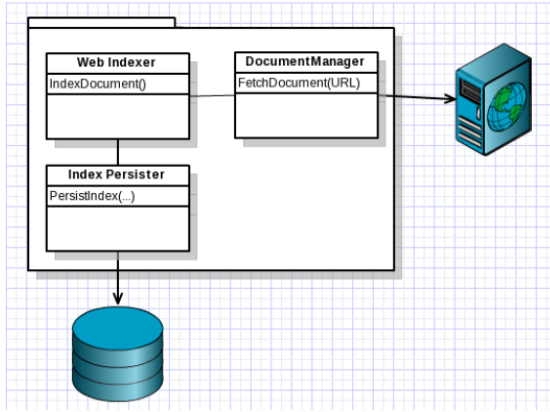
Problems

- Requires a lot of setup
- Slow (if going out or to the db)
- Fragile (lots of code can affect the outcome)

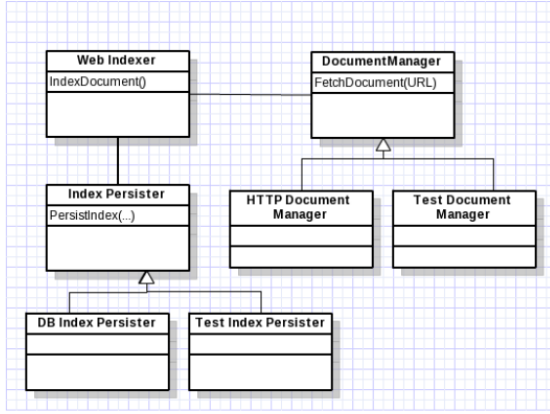
Lack of focus, lots of responsibilities



Break it apart …



Isolate SUT from complex dependencies



Test double (or “imposter”)

Replace a component on which the system under test depends with a test-specific equivalent – a test double

Variants:

- Dummy Object
- Test Stub
- Test Spy
- Mock Object
- Fake Object

Gerard Meszaros

Test stub

- Control point for indirect inputs (provide expected state)
- Allows us to test paths dependent on external inputs

```
source=new StubDocumentSource();
i=new Indexer(source);
source.NextDocument=”<html><a href='x'></a></html>”;
idx=i.index(“http://www.google.com”);
Assert.AreEqual(1, idx.LinkCount);
```

When to use a test stub

- The dependencies do not allow us to exercise the SUT with the necessary indirect inputs.
- Slow tests because of external communication

Test spy

- Observation point for indirect outputs (record state for later verification, not interactions)

```
var repository=new SpyRepository();
var indexer=new DocumentIndexer();
indexer.index(document);
Assert.AreEqual(reformat(repository.Content),
reformat(document));
```

When to use a test spy

- verifying indirect outputs and cannot predict the value of all attributes of the interactions
- want assertions to be visible in the test and mock object expectations are not sufficiently intent-revealing.
- Test requires test-specific equality therefore we cannot use the standard definition of equality
- A failed assertion cannot be reported effectively back

Mock Object

- Observation point for indirect outputs as exercised (record interactions for later verification)

```
var connection=new MockConnection();
var indexer=new Indexer(connection);
indexer.index();
Assert.AreEqual("connect",connection.Call[0]);
Assert.AreEqual("load",connection.Call[1]);
Assert.AreEqual("disconnect",connection.Call[2]);
```

When to use a Mock Object

- Verify behaviour by observing side-effects
- Often because the real thing is not yet available
- Not a control or observation point

Fake Object

- Replace functionality for reasons other than verification (does the same thing but a lot simpler)
- Not a control or observation point

```
var simple=new FakeIndexer(); // only does words
var webCrawler=new WebCrawler(simple, repository)
webCrawler.crawl(stubWebSite);
Assert.AreEqual(150,repository.WordCount);
```

When to use a fake object

- The real thing is not yet available
- When the real thing makes testing too difficult or slow, and interaction sequences would require very complex mock/stub setup

Dummy Object

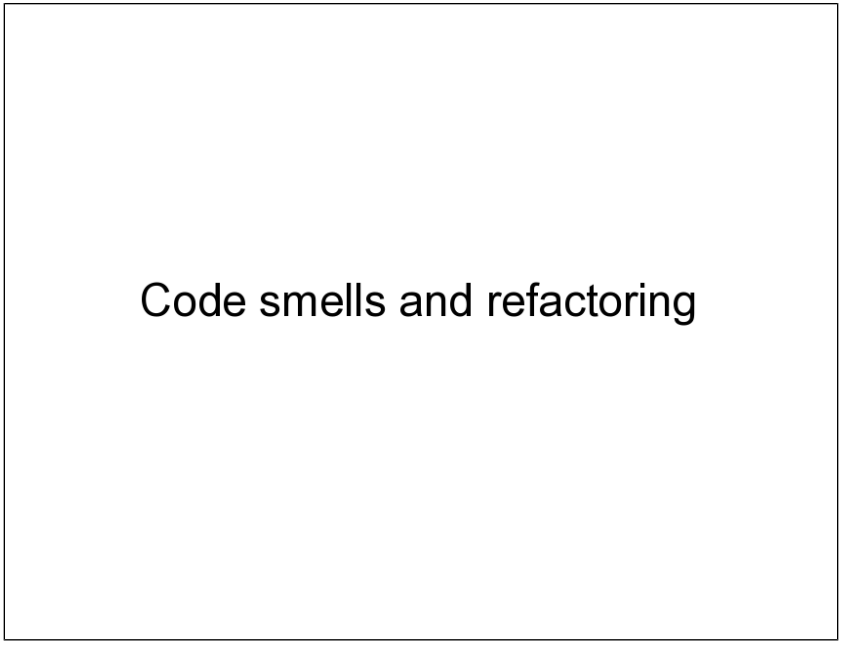
- Alternative to the value patterns (replace a complex parameter required by method signatures that test doesn't really care about)

```
var dummy=new DummyAuthenticator();
var webCrawler=new WebCrawler(stubRepository);
// stubSite does not ask for authentication
webCrawler.crawl(stubSite, dummy);
```

Mock Libraries

- Automatically provide test doubles
- Record and verify behaviour of doubles on demand
- Verify external communication

Code Smells And Refactoring



Code needs housekeeping

- Duplication and inconsistency creeps in
- Old code does not reflect new knowledge or domain insights
- New modules might be integrated better
- Technical debt
- New versions of frameworks deprecate code
- …

Code smells

A hint that something has gone wrong somewhere in your code.

Duplication - Makes code hard to modify

- Same code in different methods of a class – extract to a new method
- Same code in two subclasses – extract to a parent class method
- Same code in different unrelated classes – extract class
- Similar code – separate similar bits from different bits and extract

Comments in method body – suggest unclear responsibilities

- “often used as a deodorant” - Fowler and Beck
- If in method body, break up into separate methods
- If in method header, rename method
- If stating preconditions, introduce assertions

Long methods – suggest unclear responsibilities

- Doing too much - extract parts into separate methods
  - comments
  - Loops
  - conditionals/switch statements
- Lots of temporary variables – extract temps into queries
- Complex conditional logic – extract conditionals and actions into methods

Large class – unclear or shared responsibilities

- Too many fields – extract classes
- Not using all fields all the time – extract class or subclass
- Too much code – remove duplication, see how clients use the class and extract interface for different uses, then break apart

Data makes it difficult to understand and use

- Long parameter lists
  - Related primitives – extract into a class
  - Lots of fields from an object – pass the whole object
- Primitives used for “small tasks” - replace data value with object, replace type codes with classes or strategies
- Groups of fields that always go together (data clump) – extract class

System is hard to change

- Improvements of a class always require changes to several methods (Divergent change) – extract variations into a separate class
- Improvements require lots of small changes to lots of classes (Shotgun Surgery) – extract affected methods and fields to a single class

Data and processing dispersed

- Method is more interested in another class than its own (Feature envy) – move method to other class, extract method to split into relevant parts
- Classes looking into others private parts – move methods and fields, extract classes with common parts, hide delegates
- Subclasses not needing all inherited fields or methods (refused bequest) – push down methods and fields, replace inheritance with composition

Many more

- See “further reading” for more resources
- Programming best practices for a particular language or framework
  - Returning non-final collections from methods
  - Unsafe exception handling in final
  - Unnecessary singletons
  - ...

Refactoring

A series of *small* steps, each of which changes the program's internal structure without changing its behaviour

Martin Fowler

Why refactor?

- To improve software design
  - Design should evolve with knowledge
  - Prevent potential future bugs
  - Make it more flexible or reusable
- To understand software better
  - Make things more explicit
- To help find bugs
  - Clarify code while debugging

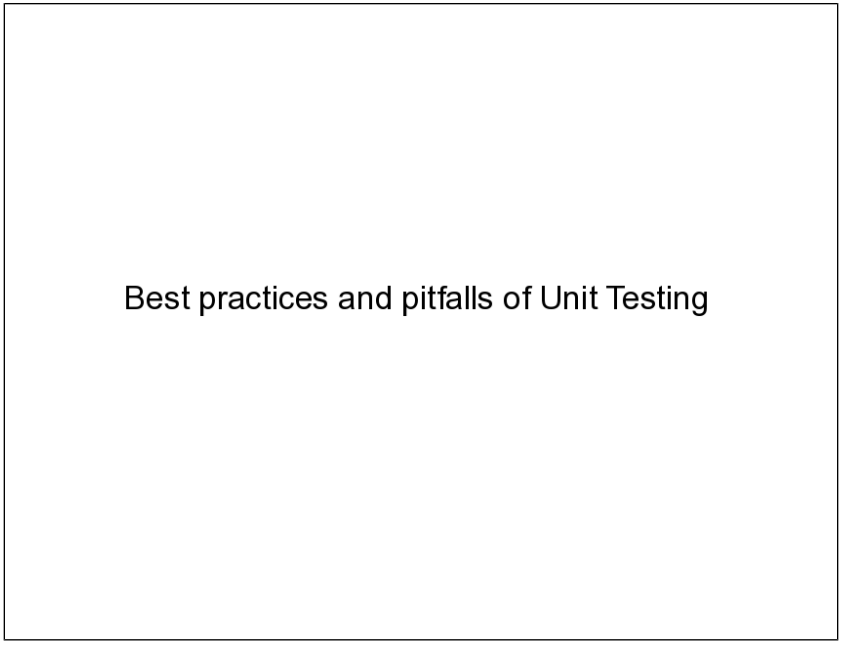
When refactor?

- A continuous activity – don't schedule
- while adding functionality
  - Make underlying code easier to change & extend
- while troubleshooting
  - Restructure code to understand it and pin-point problems
- during code reviews
  - Allows you to look at the code from a higher level

Things that make it problematic

- Published APIs
- Database changes
- Code without tests

Best practices and pitfalls of unit testing



Unit tests are poka-yoke devices

- Source-inspection
  - Write tests before you writing code
- Inexpensive verification
  - Automated, quick, providing very quick feedback
- Inspection to prevent defects
  - Relevant, describing code features

A test is not a unit test if...

- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (editing config files...) to run it

Michael Feathers

Unit tests also improve communication

- Good tests explain the intent of code!
  - Choose test names carefully
  - Make tests easy to read and understand

Dangers of mocking/stubbing

- Very easy to over-specify software
  - Tests become a pain to maintain
  - They impede refactoring
- Very easy to over-complicate
  - Tests become hard to understand
- Very easy to hide problems
  - Make sure you have integration tests as well

Classical and Mockist Testing

- Classical: real objects if possible and a double if it's awkward to use the real thing.
- Mockist: always use a mock for any object with interesting behaviour.

Martin Fowler

Classical

- check results, not how we got there
- Likely to suffer from coarse grained tests
- Tests not tied to implementation
- Prefer domain-model design

Mockist

- check how we got to the results
- Very isolated tests
- Tests presume implementation – hard to refactor
- Prefer outside-in design

Martin Fowler

Five steps of TDD

- Think is the most difficult one for beginners
  - come up with tests that really help you move forward
- Red and Green should be fairly short
- Refactor is where most of the work should be
- Work in short cycles and repeat

Unit tests are not full QA

- Unit tests are there to support development and flesh out design
  - Not to cover all possible permutations of arguments
  - Not to perform full end-to-end integration tests
  - Not to perform non-functional/stress/security tests
- You still need those other tests, but unit tests are not where they should be
  - Some might even be automated using unit-test tools, but separate them

Signs that something is wrong

- 3rd party API changes cause tests to fail
- Many tests fail when a single behaviour changes
- Data-sensitive tests
- order of test execution is important
- conditional logic in tests
- UI change causes tests to fail (interface sensitivity)

Test coverage

- Automated tools can tell you which parts of code aren't covered by tests
  - May be a good idea to restructure code or write some additional tests
- 100% test coverage does not mean bug-free

Don't retro-fit tests

- Microsoft issued TDD guidelines in 2005 suggesting design-red-green – completely wrong
- Retrofitting tests increases test coverage but doesn't help avoid just-in-case code
- Much better to do proper red-green-refactor

Resources

Books and articles

[1] Gojko Adzic. Copyright © 2009. Neuri. *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing*. 0955683610.

[2] Gojko Adzic. Copyright © 2008. Neuri. *Test Driven .NET Development With FitNesse*. 0955683602.

[3] Gerard Meszaros. Copyright © 2007. Addison Wesley. *xUnit Test Patterns: Refactoring Test Code*. 0131495054.

[4] Lisa Crispin and Janet Gregory. Copyright © 2009. Addison-Wesley Professional. *Agile Testing: A Practical Guide for Testers and Agile Teams*. 0201835959.

[5] Shigeo Shingo. Copyright © 1986. Productivity Press. *Zero Quality Control: Source Inspection and the Poka-Yoke System*. 0915299070.

[6] Mary Poppendieck and Tom Poppendieck. Copyright © 2003. Addison-Wesley Professional. *Lean Software Development: An Agile Toolkit*. 0321150783.

[7] Martin Fowler. Copyright © 1999. Addison-Wesley Publishing Company. *Refactoring: Improving the Design of Existing Code*. 0201485672.

[8] Kent Beck. Copyright © 2002. Addison-Wesley Publishing Company. *Test Driven Development By Example*. 0321146530.

[9] Gerald M. Weinberg and Donald C. Gause. Copyright © 1989. Dorset House Publishing Company. *Exploring Requirements: Quality Before Design*. 0932633137.

[10] Michael Feathers. Copyright © 2004. Prentice Hall. *Working Effectively with Legacy Code*. 0131177052.

[11] Steve Freeman. Nat Pryce. Copyright © 2009. Addison Wesley. *Growing Object-Oriented Software, Guided by Tests*. 0321503627.

Online resources

- TDD Problems *A collection of classic TDD problems that you can use to practice your skills*: http://sites.google.com/site/tddproblems
- Michael Feathers *A Set of Unit Testing Rules*: http://www.artima.com/weblogs/viewpost.jsp?thread=126923
- Mary Poppendieck *Competing on the basis of Speed*: http://video.google.com/videoplay?docid=-5105910452864283694
- Robert C. Martin *Three rules of TDD*: http://butuncle-bob.com/ArticleS.UncleBob.TheThreeRulesOfTdd
- Jim Shore *Red-Green-Refactor*: http://jamesshore.com/Blog/Red-Green-Refactor.html
- Gojko Adzic *Thought-provoking TDD Exercise*: http://gojko.net/2009/02/27/thought-provoking-tdd-exercise-at-the-software-craftsmanship-conference/
- Gojko Adzic *TDD as if you meant it, revisited*: http://gojko.net/2009/08/02/tdd-as-if-you-meant-it-revisited/
- Martin Fowler *Mocks Aren't Stubs*: http://martin-fowler.com/articles/mocksArentStubs.html
- Gerard Meszaros *XUnit Patterns*: http://xunitpatterns.com/
- Gojko Adzic *When TDD goes bad*: http://gojko.net/2008/02/25/when-tdd-goes-bad/
- Gojko Adzic *Mary Poppendieck - test driven development redefined*: http://gojko.net/2009/10/16/mary-poppendieck-test-driven-development-redefined/
- Taxonomy of bad code smells: http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm

- **Martin Fowler *Refactoring Home Page*: http://www.refactoring.com**
- **Source Making *Refactoring*: http://sourcemaking.com/refactoring**
- **Gojko Adzic *What have we learned from 10 years of TDD?*: http://gojko.net/2009/03/11/qcon-london-2009-what-have-we-learned-from-10-years-of-tdd/**
- **Code Smells: http://c2.com/cgi/wiki?CodeSmell**
- **Slava Imeshev, Continuous Integration Benefits, Challenges and Best Practices: http://www.accu-usa.org/Slides/accu_continuous_integration_best_practises.pdf**
- **Alistair Cockburn: Hexagonal architecture http://alistair.cockburn.us/Hexagonal+architecture**
- **Eric Evans: Folding Together DDD and Agile http://skillsmatter.com/podcast/design-architecture/folding-together-ddd-agile**
- **Gojko Adzic: TDD, DDD and BDD http://skillsmatter.com/podcast/design-architecture/ddd-tdd-bdd**

Tools

- **JUnit: http://junit.org/**
- **Mockito: http://code.google.com/p/mockito/**
- **MbUnit: http://www.mbunit.com/**
- **Gallio: http://www.gallio.org/**
- **Rhino Mocks: http://www.ayende.com/projects/rhino-mocks.aspx**
- **Moq: http://code.google.com/p/moq/**

Mailing Lists

- **Refactoring: http://tech.groups.yahoo.com/group/refactoring/**
- **Agile Testing: http://tech.groups.yahoo.com/group/agile-testing**
- **Test Driven Development: http://tech.groups.yahoo.com/group/testdrivendevelopment/**