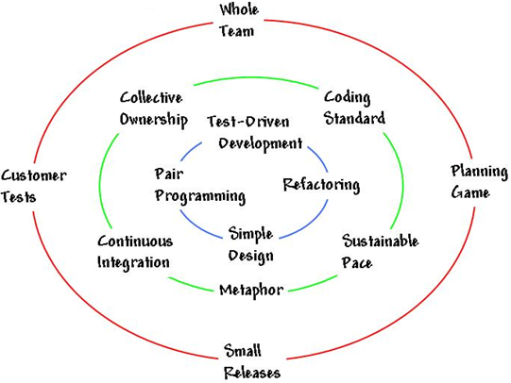
Growing Agile Technical practices

“A beginning is a very delicate time” ­­

Frank Herbert, Dune.

This book is about technical practices, so we should start by defining them. What technical practices? Coming up with a list of core technical practices can be a daunting task. Every single programmer will have its own core, so consensus is difficult. The title of the book mentions agile, so let’s start there.

What are agile technical practices? Still pretty open to interpretation, the agile manifesto does not mention any technical practices. SCRUM and Kanban do not mention any technical practices also. Can we use the Software Craftsmanship manifesto? No, no specific technical practices there as well. So what about Extreme Programing? XP does have a set of technical practices. The image below shows the technical practices advocated by XP. Now, what of these are individual practices? The inner circle is probably the best candidate. In absence of anything else we can start with these.

**

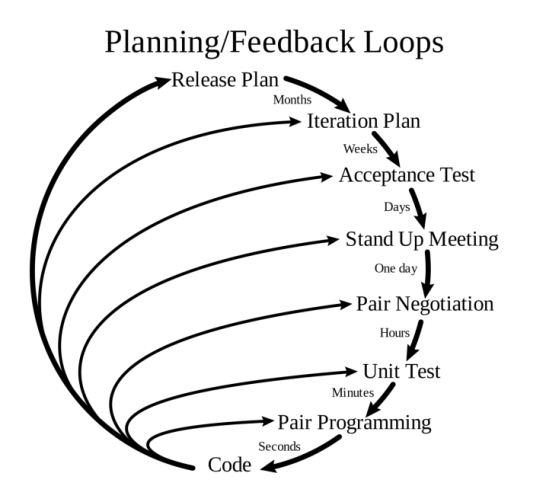
Now another question arises. Do we need a core of technical practices in agile? Let’s answer this question with a couple of questions. How often do projects become stagnated under the weight of technical debt? Why does this happen? One of the factors is probably technical debt. Or the lack of technical practices that constantly reduces the amount of technical debt in a project. This is one of the reasons why technical practices are so important to the success of an agile project. They allow us to continue delivering value by addressing the incidental complexity or tech debt as it emerges.

So are XP practices TDD, Pair programming, Refactoring and Simple Design our core practices? Would these practices help prevent tech debt accumulating. Simple design and refactoring look promising; let’s label them as design technical practices. What about pair programming and TDD, these are in a slightly different category; let’s label them feedback technical practices.

Having practices related to design and feedback creates an interesting situation. It allows for experimentation. Consider a design change and get feedback from the pair, then change the design and get feedback from tests. Rapid experimentation and feedback is a pillar of human evolution. Kent Beck on extreme programming:

*“Optimism is an occupational hazard of programming: feedback is the treatment.”*

*“Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, courage, and respect. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.”*

The importance of feedback loops in XP is illustrated in the following diagram:

Back to the XP technical practices: TDD, Pair programming, Refactoring and Simple Design. They are still big boxes we need to open and find out what is inside each of them. Let’s do that.

If we go more granular on pair programming we can find the following concepts and practices:

* Driver-navigator
* Ping-Pong/Popcorn
* Chess clock
* Pomodoro
* Pair Rotation

There are two big TDD schools the Classic/Classicist/Detroit school and the Outside-In/Mockist/London so let’s add both to the TDD box. Outside-In TDD relies heavily on mocks and both schools recommend using stubs. The Transformation Priority Premise is a great way to evolve code without adding complexity too soon, so can be seen as part of TDD or Simple Design we choose to add it to TDD. Mocks and stubs are often referred more generically as test doubles. Add test doubles to the mix and we end up with:

* Classic TDD
* Outside-In TDD
* Test Doubles
* Transformation Priority Premise

Doing the same exercise for refactoring we can end up with something like the following:

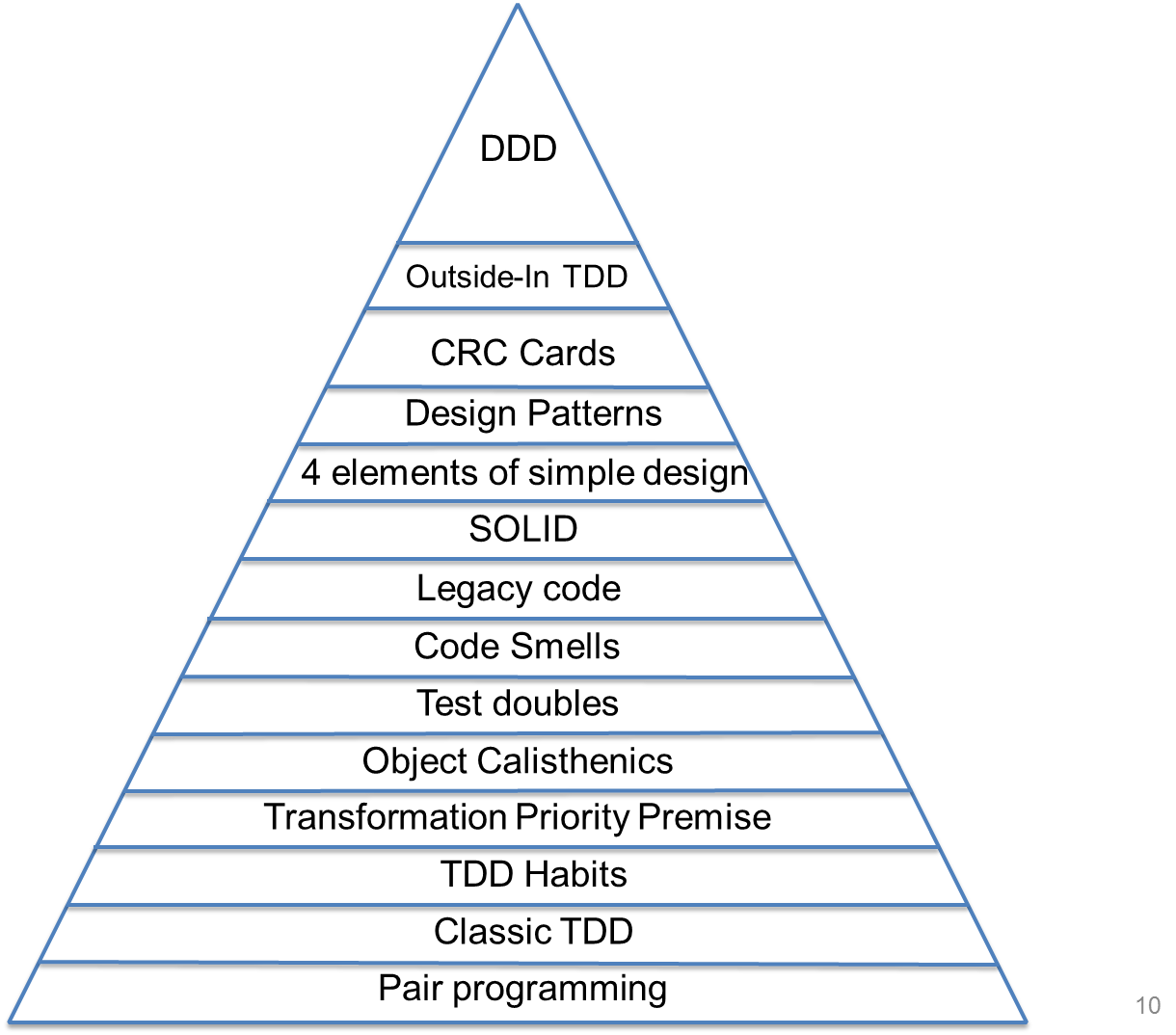
* IDE Productivity
* Refactoring smells
* Refactoring legacy code

Design is a much more controversial box regarding content, but XP speaks about simple design, so let’s stay with high level generally accepted design principles and practices such as:

* Object Calisthenics
* Code smells
* SOLID
* 4 elements of simple design
* Design Patterns
* CRC Cards
* Domain Driven Design

After we went a bit more granular in XP practices we need to address the next problem. In what order should one learn these subjects?

After spending many hours coaching and going through several iterations on the order of subjects we are now at this point:



**Links**

<http://en.wikipedia.org/wiki/Extreme_programming>

<http://www.extremeprogramming.org/>

<http://www.extremeprogramming.org/rules.html>

<http://ronjeffries.com/xprog/what-is-extreme-programming/>

**Reading list**

<http://www.amazon.co.uk/Extreme-Programming-Explained-Embrace-Change/dp/0321278658/ref=sr_1_3?ie=UTF8&qid=1428489551&sr=8-3&keywords=kent+beck>

# Pair programming

*“Pair programmers: Keep each other on task. Brainstorm refinements to the system. Clarify ideas. Take initiative when their partner is stuck, thus lowering frustration. Hold each other accountable to the team's practices. Pairing”*

Kent Beck

## Roles

* Driver
  + Responsible for typing, moving the mouse, etc
* Navigator
  + Responsible for reviewing the driver’s work. In addition to catching incidental mistakes, the navigator considers the code at a more strategic level: how will this fit with the rest of the code? Will this implementation require changes elsewhere? Could we design this program better?

## Driver/Navigator switch Techniques

### Ping Pong/Popcorn

|  |  |
| --- | --- |
| **Developer A** | **Developer B** |
| Writes a new test and sees that it fails |  |
|  | Implement the code needed to pass the test  Writes the next test and sees that it fails |
| Implement the code needed to pass the test  Writes the next test and sees that it fails |  |
|  | Implement the code needed to pass the test  Writes the next test and sees that it fails |

### Chess clock

Using a chess clock the pair defines a time interval to each one be the driver, when the clock rings or a natural breaking point happens the pair switch theirs roles. It’s important that both partners spend an equal amount of time in each role.

### Pomodoro

Spending the whole day pairing with someone can be very tiring. We need to justify and agree on every single line of code. Things that are trivial for one developer may not be as trivial for another developer. As eXtreme Programming becomes more popular, pair programming is becoming mandated in certain companies. By the end of the day, developers are exhausted, as they had no time for themselves.

How to improve it: Developers need some alone time. Regular breaks are important. Using the Pomodoro technique is a good way to force regular breaks.

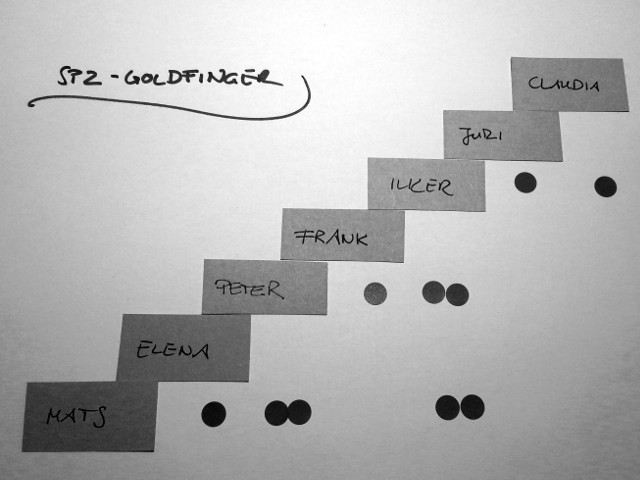
There are five basic steps to implementing the technique:

1. Decide on the task to be done
2. Set the pomodoro timer to n minutes (traditionally 25)
3. Work on the task until the timer rings; record with an x
4. Take a short break (3–5 minutes)
5. After four pomodori, take a longer break (15–30 minutes)

### Pair rotation

#### Pair stairs

Simple matrix of all team members (including testers, business analysts, sysops). Whenever two team members pair up, they mark their “pairing” with a dot, check or simple line. The team always strives to get the complete chart filled, which effectively means that the team has undergone all pairing combinations.



#### Links

<http://www.kusmin.eu/wiki/index.php/Pair_Programming>

<http://roger-almeida.blogspot.co.uk/2010/04/pairing-techniques.html>

<http://codurance.com/2015/03/15/rethinking-pair-programming/>

# 

# Classic TDD (Test Driven Development) I – Basics

*“Any fool can write code that a computer can understand. Good programmers write code that humans can understand”*

Martin Fowler

The Classicist approach is the original approach to TDD created by Kent Beck. It’s also known as Chicago/Detroit School of TDD.

## Benefits

### Debugging

What would programming be like if you were never more than a few minutes away from running everything and seeing it work? Imagine working on a project where you never have several modules torn to shreds hoping you can get them all put back together next Tuesday. Imagine your debug time shrinking to the extent that you lose the muscle memory for your debugging hot keys.

### Courage

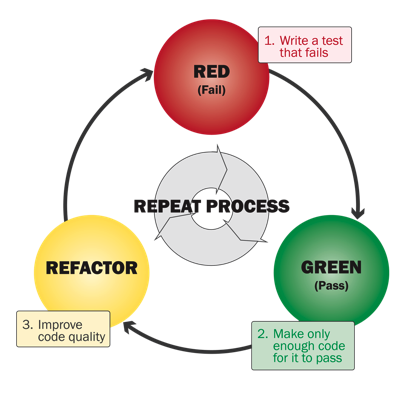
Have you ever seen code in a module that was so ugly that your first reaction was "Wow, I should clean this" Of course your next reaction was: "I'm not touching it!" Why? Because you were afraid you'd break it. How much code could be cleaned if we could conquer our fear of breaking it? If you have a suite of tests that you trust, then you are not afraid to make changes. You are not afraid to make changes! You see a messy function, you clean it. All the tests pass! The tests give you the courage to clean the code! Nothing makes a system more flexible than a suite of tests — nothing. If you have a beautiful design and architecture, but have no tests, you are still afraid to change the code. But if you have a suite of high-coverage tests then, even if your design and architecture are terrible, you are not afraid to clean up the design and architecture.

### Documentation

Have you ever integrated a third party package? They send you a nice manual written by a tech writer. At the back of that manual is an ugly section where all the code examples are shown. Where's the first place you go? You go to the code examples. You go to the code examples because that's where the truth is. Unit tests are just like those code examples. If you want to know how to call a method, there are tests that call that method every way it can be called. These tests are small, focused documents that describe how to use the production code. They are design documents that are written in a language that the programmers understand; are unambiguous; are so formal that they execute; and cannot get out of sync with the production code.

### Design

If you follow the three laws every module will be testable by definition and another word for testable is decoupled. In order to write your tests first, you have to decouple the units you are testing from the rest of the system. There's simply no other way to do it. This means your designs are more flexible, more maintainable, and just cleaner.



The three laws of TDD (Rules) Baby Steps

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.

Refactoring is missing from the above rules TDD without refactoring does not work.

In the refactoring phase we should look for duplication. But wait until we are sure of a duplication pattern. We should avoid premature abstractions. Use the rule of three: “Extract duplication only when you see it the third time.” The Rule of Three defers the duplication minimization until we have enough proof. Remember that “**Duplication is far cheaper than the wrong abstraction**”. Code that does not contain duplication is often referred as conformant with DRY principle (Do not repeat yourself).

## Three methods of going from red to green

1. **Fake it** is where you just return the exact value you need. If your test expects a zero from a method, use a “return 0;” statement. Usually you use this when you cannot tell how to implement certain functionality, or your previous steps were too large and you cannot figure out what went wrong. Something that works is better than something that doesn’t work!
2. **Use obvious implementation** when you are pretty sure of the code you need to write, so write it and see if the test passes. The majority of the time you will use this method to move forward with TDD quickly.
3. **Triangulation**[[1]](#footnote-1) is where you want to generalize certain behaviour, but are not sure how to do it. So you start with fake it and add additional tests that force the code to be more generic along a certain dimension.

## Main characteristics

* Design happens during the refactoring phase.
* Normally tests are state-based tests.
* During the refactoring phase, the unit under test may grow to multiple classes.
* Mocks are rarely used, unless when isolating external systems.
* No up-front design considerations are made. Design completely emerges from code.
* It’s a great way to avoid over-engineering.
* Easier to understand and adopt due to state-based tests and no design up-front.
* Often used in conjunction with the 4 Rules of Simple Design.
* Good for exploration, when we know what the input and desired output are but we don’t really know how the implementation looks like.
* Great for cases where we can’t rely on a domain expert or domain language (data transformation, algorithms, etc.)

## Problems

* Exposing state for tests purpose only.
* Unit under test becomes bigger than a class when classes emerge during the refactoring phase. This is fine when we look at that test in isolation but as classes emerge, they create life of their own, being reused by other parts of the application. As these other classes evolve, they may break completely unrelated tests, since the tests use their real implementation instead of a mock.
* Refactoring (design improvement) step is often skipped by inexperienced practitioners, leading to a cycle that looks more like RED-GREEN-RED-GREEN-…-RED-GREEN-MASSIVE REFACTORING.
* Due to its exploratory nature, some classes under test are created according to the “I think I’ll need this class with this interface (public methods)”, making them not fit well when connected to the rest of the system.
* Can be slow and wasteful since quite often we already know that we cannot have so many responsibilities in the class under test. The classicist advice is to wait for the refactoring phase to fix the design, only relying on concrete evidence to extract other classes. Although this is good for novices, this is pure waste for more experienced developers.

## Katas

### Fizz Buzz

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

#### Leap year

Write a function that returns true or false depending on whether its input integer is a leap year or not. A leap year is defined as one that is divisible by 4, but is not otherwise divisible by 100 unless it is also divisible by 400. For example, 2001 is a typical common year and 1996 is a typical leap year, whereas 1900 is an atypical common year and 2000 is an atypical leap year.

#### Nth - Fibonacci

Write some code to generate the Fibonacci number for the nth position ex:

int Fibonacci(int position)

First Fibonacci numbers in the sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

### Links

<http://gmarik.info/notes/programming/test-driven-development-kent-beck>

<http://codemanship.co.uk/parlezuml/blog/?postid=1157>

<http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd>

http://codurance.com/2015/05/12/does-tdd-lead-to-good-design/

<http://peripateticaxiom.blogspot.de/2006/05/complexity-and-test-first-0.html>

<http://programmer.97things.oreilly.com/wiki/index.php/The_Three_Laws_of_Test-Driven_Development>

# Classic TDD II - TDD Habits

*“I'm not a great programmer; I'm just a good programmer with great habits.”*

Kent Beck

* Principles
  + Tests should test one thing only
  + Test one logical assertion
  + Don't mix asserts of state and collaborations together
  + Make tests independent of each other
  + Don't refactor with failing test
  + Organise your unit tests to reflect your production code
  + Keep your tests and production code separate
  + Do not use production data and code to test production code
  + If your tests are difficult to write or maintain, consider changing the design
* Red
  + Create more specific tests to drive a more generic solution (Triangulate)
  + Give your test meaningful names (behaviour / goal-oriented) - that reflects your production
  + Write the assertion first and work backwards
  + See the test fail for the right reason
  + Ensure you have meaningful feedback from failing tests
  + Organize your test in Arrange, Act and Assert blocks
    - Arrange (aka Given) – all necessary preconditions and inputs.
    - Act (aka When) – on the object or method under test.
    - Assert (aka Then) – that the expected results have occurred.
    - Structure the code in tests to reflect these concepts.
* To Green
  + Write the simplest code to pass the test
  + Consider using Transformation Priority Premise
  + Write any code that makes you get to the refactor phase quicker
  + It is okay to write any code that you might improve at a later stage
* Refactor
  + Refactor aggressively and constantly
  + Treat tests as first class code
  + Use the IDE to refactor quickly and safely
  + Use the Rule of 3 to tackle duplication
    - Code can be copied once, but that when the same code is used three times, it should be abstracted. Or if you need something once, build it. If you need something twice, pay attention. If you need it a third time, abstract it.
    - But keep in mind that duplication is cheaper than the wrong abstractions

## Arrange, Act, Assert

1. Arrange aka Given – all necessary preconditions and inputs.
2. Act aka When – on the object or method under test.
3. Assert aka Then – that the expected results have occurred.

Structure the code in tests to reflect these concepts.

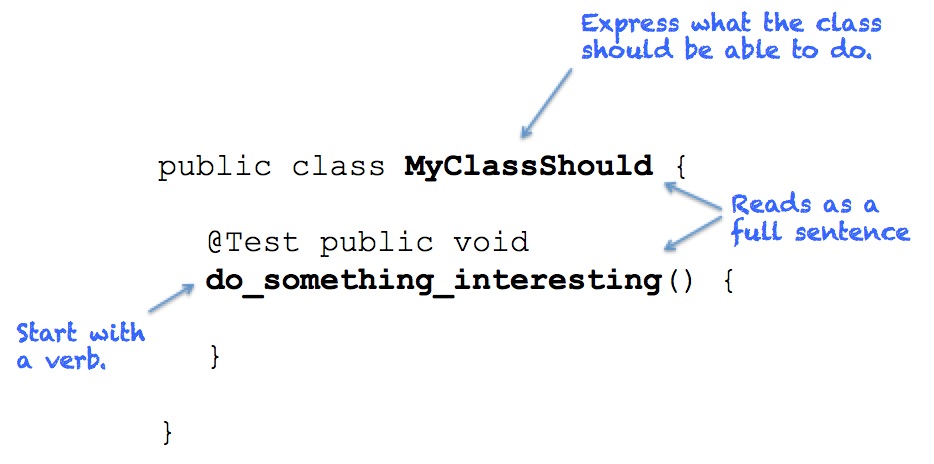
### Benefits

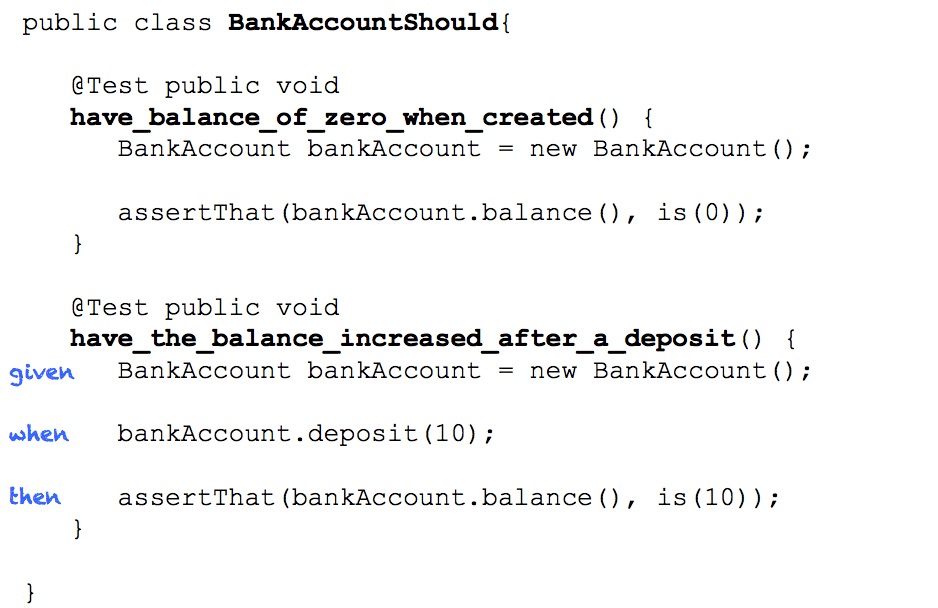
* Clearly separates what is being tested from the setup and verification steps.
* Clarifies and focuses attention on a historically successful and generally necessary set of test steps.
* Makes some Test Smells? more obvious:
  + Assertions intermixed with "Act" code.
  + Test methods that try to test too many different things at once.

## Unit Test Principles

* Fast
  + Unit tests have to be fast in order to be executed often. Fast means much smaller than seconds.
* Isolated
  + Clear where the failure happened. No dependency between tests (random order).
* Repeatable
  + No assumed initial state, nothing left behind, no dependency on external services that might be unavailable (databases, file system …).
* Self-Validating
  + No manual test interpretation or intervention. Red or green!
* Timely
  + Tests are written at the right time

## Naming test classes and methods





## Unit Test Smells

* Test Not Testing Anything
  + Passing test that at first sight appears valid but does not test the testee.
* Test Needing Excessive Setup
  + A test that needs dozens of lines of code to set up its environment. This noise makes it difficult to see what is really tested.
* Too Large Test / Assertions for Multiple Scenarios
  + A valid test that is, however, too large. Reasons can be that this test checks for more than one feature or the testee does more than one thing (violation of Single Responsibility Principle).
* Checking Internals
  + A test that accesses internals (private/protected members) of the testee directly (Reflection). This is a refactoring killer.
* Test Only Running on Developer’s Machine
  + A test that is dependent on the development environment and fails elsewhere. Use continuous integration to catch them as soon as possible.
* Test Checking More than Necessary
  + A test that checks more than it is dedicated to. The test fails whenever something changes that it checks unnecessarily. Especially probable when fakes are involved or checking for item order in unordered collections.
* Irrelevant Information
  + Test contains information that is not relevant to understand it.
* Chatty Test
  + A test that fills the console with text – probably used once to manually check for something.
* Test Swallowing Exceptions
  + A test that catches exceptions and lets the test pass.
* Test Not Belonging in Host Test Fixture
  + A test that tests a completely different testee than all other tests in the fixture.
* Obsolete Test
  + A test that checks something no longer required in the system. May even prevent clean-up of production code because it is still referenced.
* Hidden Test Functionality
  + Test functionality hidden in the SetUp method, base class or helper class. The test should be clear by looking at the test method only – no initialisation or asserts somewhere else.
* Bloated Construction
  + The construction of dependencies and arguments used in calls to testee makes test hard to read.
* Unclear Fail Reason
* Conditional Test Logic
  + Tests should not have any conditional test logic because it’s hard to read.
* Test Logic in Production Code
  + Tests depend on special logic in production code.
* Erratic Test
  + Sometimes passes, sometimes fails due to left overs or environment.

## Katas

### String Calculator

1. Create a simple String calculator with a method **int Add(string numbers)**
2. The method can take 0, 1 or 2 numbers, and will return their sum (for an empty string it will return 0) for example**“” or “1” or “1,2”**
3. Allow the Add method to handle an unknown amount of numbers
4. Allow the Add method to handle new lines between numbers (instead of commas).
   1. the following input is ok:  “1\n2,3”  (will equal 6)
   2. the following input is NOT ok:  “1,\n” (not need to prove it - just clarifying)
5. Support different delimiters
6. to change a delimiter, the beginning of the string will contain a separate line that looks like this:   “//[delimiter]\n[numbers…]” for example “//;\n1;2” should return three where the default delimiter is ‘;’ . The first line is optional. all existing scenarios should still be supported
7. Calling Add with a negative number will throw an exception “negatives not allowed” - and the negative that was passed.if there are multiple negatives, show all of them in the exception message

**stop here if you are a beginner**. Continue if you can finish the steps so far in less than 30 minutes.

1. Numbers bigger than 1000 should be ignored, so adding 2 + 1001  = 2
2. Delimiters can be of any length with the following format:  “//[delimiter]\n” for example: “//[\*\*\*]\n1\*\*\*2\*\*\*3” should return 6
3. Allow multiple delimiters like this:  “//[delim1][delim2]\n” for example “//[\*][%]\n1\*2%3” should return 6.
4. make sure you can also handle multiple delimiters with length longer than one char

### Bowling game

Write a program to score a game of Ten-Pin Bowling.

Input: string (described below) representing a bowling game

Output: integer score

#### The scoring rules:

* Each game, or "line" of bowling, includes ten turns, or "frames" for the bowler.
* In each frame, the bowler gets up to two tries to knock down all ten pins.
* If the first ball in a frame knocks down all ten pins, this is called a "strike". The frame is over. The score for the frame is ten plus the total of the pins knocked down in the next two balls.
* If the second ball in a frame knocks down all ten pins, this is called a "spare". The frame is over. The score for the frame is ten plus the number of pins knocked down in the next ball.
* If, after both balls, there is still at least one of the ten pins standing the score for that frame is simply the total number of pins knocked down in those two balls.
* If you get a spare in the last (10th) frame you get one more bonus ball. If you get a strike in the last (10th) frame you get two more bonus balls.
* These bonus throws are taken as part of the same turn. If a bonus ball knocks down all the pins, the process does not repeat. The bonus balls are only used to calculate the score of the final frame.
* The game score is the total of all frame scores.

#### Examples

* X indicates a strike
* / indicates a spare
* - indicates a miss
* | indicates a frame boundary
* The characters after the || indicate bonus balls

##### Example 1

##### X|X|X|X|X|X|X|X|X|X||XX -> Total score = 10 frames x 30 = 300

##### Example 2

##### 9-|9-|9-|9-|9-|9-|9-|9-|9-|9-|| -> Total score = 10 frames x 9 = 90

##### Example 3

##### 5/|5/|5/|5/|5/|5/|5/|5/|5/|5/||5 -> Total score = 10 frames x 15 = 150

##### Example 4

##### X|7/|9-|X|-8|8/|-6|X|X|X||81 -> Total score = 167

## Links

<http://codurance.com/2014/12/13/naming-test-classes-and-methods/>

### Reading list

<http://www.amazon.co.uk/Test-Driven-Development-Addison-Wesley-Signature/dp/0321146530/ref=sr_1_1?ie=UTF8&qid=1428489551&sr=8-1&keywords=kent+beck>

# Classic TDD III - Transformation Priority Premise

“As tests get more specific code gets more generic.” Uncle Bob

If we go back to how we evolve code in TDD, we have these methods:

1. **Fake it** is where you just return the exact value you need. If your test expects a zero from a method, use a “return 0;” statement. Usually you use this when you cannot tell how to implement certain functionality, or your previous steps were too large and you cannot figure out what went wrong. Something that works is better than something that doesn’t work!
2. **Use obvious implementation** when you are pretty sure of the code you need to write, so write it and see if the test passes. The majority of the time you will use this method to move forward with TDD quickly.
3. **Triangulation** is where you want to generalize certain behaviour, but are not sure how to do it. So you start with fake it and add additional tests that force the code to be more generic along a certain dimension.

“Use obvious implementation” is ambiguous, what does “obvious implementation” mean? It may mean something for you and something else for another developer. The transformations on the following table are a way to clarify ambiguity the “obvious implementation”.

|  |  |  |  |
| --- | --- | --- | --- |
| # | Transformation | Example | |
| **Start code** | **End** **code** |
| 1 | {} -> nil | {} | [return] nil |
| 2 | Nil -> constant | [return] nil | [return] “1” |
| 3 | Constant -> constant+ | [return] “1” | [return] “1” + “2” |
| 4 | Constant -> scalar | [return] “1” + “2” | [return] argument |
| 5 | Statement -> statements | [return] argument | [return] min(max(0, argument), 10) |
| 6 | Unconditional -> conditional | [return] argument | if(condition) [return] argument  else [return] 0 |
| 7 | Scalar -> array | dog | [dog, cat] |
| 8 | Array -> container | [dog, cat] | {dog=”DOG”, cat=”CAT”} |
| 9 | Statement -> tail recursion | a + b | a + recursion |
| 10 | If -> while | if(condition) | while(condition) |
| 11 | Statement -> recursion | a + recursion | recursion |
| 12 | Expression -> function | today – birth | CalculateBirthDate() |
| 13 | Variable -> assignment | day | Day = 10 |
| 14 | Case |  |  |

Transformations on the top of the list should be preferred to those that are lower. It is better (or simpler) to change a constant into a variable than it is to add an if statement. So when making a test pass, you try to do so with transformations that are simpler (higher on the list) than those that are more complex.

**Please do not take this table literally.** This is a starting point.

## Katas

### Roman Numbers

##### Definition

Given a positive integer number (eg 42) determine its Roman numeral representation as a String (eg "XLII"). You cannot write numerals like IM for 999.

##### Examples

|  |  |  |  |
| --- | --- | --- | --- |
| Arabic number | Roman numeral | Arabic number | Roman numeral |
| 1 | I | 60 | LX |
| 2 | II | 70 | LXX |
| 3 | III | 80 | LXXX |
| 4 | IV | 90 | XC |
| 5 | V | 100 | C |
| 6 | VI | 200 | CC |
| 7 | VII | 300 | CCC |
| 8 | VIII | 400 | CD |
| 9 | IX | 500 | D |
| 10 | X | 600 | DC |
| 20 | XX | 700 | DCC |
| 30 | XXX | 800 | DCCC |
| 40 | XL | 900 | CM |
| 50 | L | 1000 | M |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Arabic number | Roman numeral | Thousands | Cents | Tenths | Units |
| 846 | DCCCXLVI | - | DCC | XL | VI |
| 1999 | MCMXCIX | M | CM | XC | IX |
| 2008 | MMVIII | MM | - | - | VIII |

##### Example sequence

We start by writing the simplest unit test we can think off.

**[**TestFixture**]**

**public** class RomanConverterShould

**{**

**[**TestCase**(**1**,** "I"**)]**

**public** void ConvertNumberToRoman**(**int number**,** string expected**)**

**{**

var romanNumeral **=** **new** RomanConverter**().**Convert**(**number**);**

Assert**.**That**(**romanNumeral**,** Is**.**EqualTo**(**expected**));**

**}**

**}**

We start the implementation using the first transformation *no code to nil*

// 1 no code -> nil

**public** class RomanConverter

**{**

**public** string Convert**(**int number**)**

**{**

**return** **null;**

**}**

**}**

The first transformation is not enough to make the test pass so we apply the second transformation *nil to constant*. This is enough to make the test pass, so we stop evolving the code until we have a failing test.

// 2 nil -> constant

**public** class RomanConverter

**{**

**public** string Convert**(**int number**)**

**{**

**return** "I"**;**

**}**

**}**

We add a new failing test.

**[**TestFixture**]**

**public** class RomanConverterShould

**{**

**[**TestCase**(**1**,** "I"**)]**

**[**TestCase**(**2**,** "II"**)]**

**public** void ConvertNumberToRoman**(**int number**,** string expected**)**

**{**

var romanNumeral **=** **new** RomanConverter**().**Convert**(**number**);**

Assert**.**That**(**romanNumeral**,** Is**.**EqualTo**(**expected**));**

**}**

**}**

The next transformation *constant to* *variable* is not sufficient to, make the test pass.

// 4 Constant -> variable

**public** class RomanConverter

**{**

**public** string Convert**(**int number**)**

**{**

var result **=** "I"**;**

**return** result**;**

**}**

**}**

The next transformation *statement to statements* is sufficient to, make the test pass.

// 5 statement -> statements

**public** class RomanConverter

**{**

**public** string Convert**(**int number**)**

**{**

**return** new String(‘I’, number)**;**

**}**

**}**

We add a new failing test.

**[**TestFixture**]**

**public** class RomanConverterShould

**{**

**[**TestCase**(**1**,** "I"**)]**

**[**TestCase**(**2**,** "II"**)]**

**[**TestCase**(**3**,** "III"**)]**

**public** void ConvertNumberToRoman**(**int number**,** string expected**)**

**{**

var romanNumeral **=** **new** RomanConverter**().**Convert**(**number**);**

Assert**.**That**(**romanNumeral**,** Is**.**EqualTo**(**expected**));**

**}**

**}**

Applying the next transformation *unconditional to conditional* is not sufficient to make the test pass, but is a step forward.

// 6 unconditional -> conditional

**public** class RomanConverter

**{**

**public** string Convert**(**int number**)**

**{**

var result **=** "I"**;**

**if** **(**number **>=** 1**)**

**{**

result **+=** "I"**;**

**}**

**return** result**;**

**}**

**}**

Adding another conditional will make the test pass, but we have duplication.

// 6 unconditional -> conditional

**public** class RomanConverter

**{**

**public** string Convert**(**int number**)**

**{**

var result **=** "I"**;**

**if** **(**number **>** 1**)**

**{**

result **+=** "I"**;**

**}**

**if** **(**number **>** 2**)**

**{**

result **+=** "I"**;**

**}**

**return** result**;**

**}**

**}**

Applying the next transformation *variable to array* removes duplication and complies with the existing test.

// 7 variable -> array

**public** class RomanConverter

**{**

**public** static **readonly** string**[]** Results **=** **{** "I"**,** "II"**,** "III" **};**

**public** string Convert**(**int number**)**

**{**

**return** Results**[**number **-** 1**];**

**}**

**}**

We add a new failing test.

**[**TestFixture**]**

**public** class RomanConverterShould

**{**

**[**TestCase**(**1**,** "I"**)]**

**[**TestCase**(**2**,** "II"**)]**

**[**TestCase**(**3**,** "III"**)]**

**[**TestCase**(**4**,** "IV"**)]**

**public** void ConvertNumberToRoman**(**int number**,** string expected**)**

**{**

var romanNumeral **=** **new** RomanConverter**().**Convert**(**number**);**

Assert**.**That**(**romanNumeral**,** Is**.**EqualTo**(**expected**));**

**}**

**}**

To make the test pass we don’t need to apply the next transformation we can make the test pass by adding a new element to the array.

// no transformation

**public** class RomanConverter

**{**

**public** static **readonly** string**[]** Results **=** **{** "I"**,** "II"**,** "III", "IV" **};**

**public** string Convert**(**int number**)**

**{**

**return** Results**[**number **-** 1**];**

**}**

**}**

While adding a new element to the array was enough to make the test pass. We now spot some duplication on character ’I’ by applying the next transformation *statement to tail recursion* we can get rid of this duplication. Since we are trying to follow the transformation table we applied the *array to collection* transformation before the tail recursion.

// 8 array -> collection and 9 statement -> tail recursion

**public** class RomanConverter

**{**

**public** static **readonly** IDictionary**<**int**,** string**>** Results **=** **new** Dictionary**<**int**,** string**>**

**{**

**{**1**,** "I"**},**

**{**4**,** "IV"**},**

**};**

**public** string Convert**(**int number**)**

**{**

**if** **(**Results**.**ContainsKey**(**number**))**

**{**

**return** Results**[**number**];**

**}**

**return** Results**[**1**] +** Convert**(**number **-** 1**);**

**}**

**}**

We add a few more failing tests but since the last transformation was still allowing us to make tests pass we waited until we had duplication to refactor.

**[**TestFixture**]**

**public** class RomanConverterShould

**{**

**[**TestCase**(**1**,** "I"**)]**

**[**TestCase**(**2**,** "II"**)]**

**[**TestCase**(**3**,** "III"**)]**

**[**TestCase**(**4**,** "IV"**)]**

**[**TestCase**(**5**,** "V"**)]**

**[**TestCase**(**6**,** "VI"**)]**

**[**TestCase**(**7**,** "VII"**)]**

**[**TestCase**(**8**,** "VIII"**)]**

**public** void ConvertNumberToRoman**(**int number**,** string expected**)**

**{**

var romanNumeral **=** **new** RomanConverter**().**Convert**(**number**);**

Assert**.**That**(**romanNumeral**,** Is**.**EqualTo**(**expected**));**

**}**

**}**

No other transformations required, simply adding new values to the dictionary allowed us to make the tests pass. We can now spot duplication, again around character ‘I’.

**public** class RomanConverter

**{**

**public** static **readonly** IDictionary**<**int**,** string**>** Results **=**

**new** Dictionary**<**int**,** string**>**

**{**

**{**1**,** "I"**},**

**{**4**,** "IV"**},**

**{**5**,** "V"**},**

**{**6**,** "VI"**},**

**{**7**,** "VII"**},**

**{**8**,** "VIII"**},**

**};**

**public** string Convert**(**int number**)**

**{**

**if** **(**Results**.**ContainsKey**(**number**))**

**{**

**return** Results**[**number**];**

**}**

**return** Results**[**1**] +** Convert**(**number **-** 1**);**

**}**

**}**

To fix the duplication we apply again the transformation *statement to tail recursion* to fix this. We don’t yet need to move to the next transformation.

// 9 statement -> tail recursion

**public** class RomanConverter

**{**

**public** static **readonly** IDictionary**<**int**,** string**>** Results **=**

**new** Dictionary**<**int**,** string**>**

**{**

**{**1**,** "I"**},**

**{**4**,** "IV"**},**

**{**5**,** "V"**},**

**};**

**public** string Convert**(**int number**)**  
 **{**

**if** **(**Results**.**ContainsKey**(**number**))**

**{**

**return** Results**[**number**];**

**}**

**if** **(**number **>** 5**)**

**{**

const string result **=** "V"**;**

**return** result **+** Convert**(**number **-** 5**);**

**}**

**return** Results**[**1**] +** Convert**(**number **-** 1**);**

**}**

**}**

Again we add more failing tests, and again, the last transformation is still making tests pass.

**[**TestFixture**]**

**public** class RomanConverterShould

**{**

**[**TestCase**(**1**,** "I"**)]**

**[**TestCase**(**2**,** "II"**)]**

**[**TestCase**(**3**,** "III"**)]**

**[**TestCase**(**4**,** "IV"**)]**

**[**TestCase**(**5**,** "V"**)]**

**[**TestCase**(**6**,** "VI"**)]**

**[**TestCase**(**7**,** "VII"**)]**

**[**TestCase**(**8**,** "VIII"**)]**

**[**TestCase**(**9**,** "IX"**)]**

**[**TestCase**(**10**,** "X"**)]**

**[**TestCase**(**40**,** "XL"**)]**

**[**TestCase**(**44**,** "XLIV"**)]**

**public** void ConvertNumberToRoman**(**int number**,** string expected**)**

**{**

var romanNumeral **=** **new** RomanConverter**().**Convert**(**number**);**

Assert**.**That**(**romanNumeral**,** Is**.**EqualTo**(**expected**));**

**}**

**}**

**public** class RomanConverter

**{**

**public** static **readonly** IDictionary**<**int**,** string**>** Results **=**

**new** Dictionary**<**int**,** string**>**

**{**

**{**1**,** "I"**},**

**{**4**,** "IV"**},**

**{**5**,** "V"**},**

**{**9**,** "IX"**},**

**{**10**,** "X"**},**

**{**40**,** "XL"**},**

**};**

**public** string Convert**(**int number**)**

**{**

**if** **(**Results**.**ContainsKey**(**number**))**

**{**

**return** Results**[**number**];**

**}**

**if** **(**number **>** 40**)**

**{**

const string result **=** "XL"**;**

**return** result **+** Convert**(**number **-** 40**);**

**}**

**if** **(**number **>** 10**)**

**{**

const string result **=** "X"**;**

**return** result **+** Convert**(**number **-** 10**);**

**}**

**if** **(**number **>** 5**)**

**{**

const string result **=** "V"**;**

**return** result **+** Convert**(**number **-** 5**);**

**}**

**return** Results**[**1**]** **+** Convert**(**number **-** 1**);**

**}**

**}**

We spot duplication around if statements, so we refactor the code to the next transformation *if to while*. This gets rid of if statement duplication, but we now have while statement duplication.

// 10 if -> while

**public** class RomanConverter

**{**

**public** static **readonly** IDictionary**<**int**,** string**>** Results **=**

**new** Dictionary**<**int**,** string**>**

**{**

**{**1**,** "I"**},**

**{**4**,** "IV"**},**

**{**5**,** "V"**},**

**{**9**,** "IX"**},**

**{**10**,** "X"**},**

**{**40**,** "XL"**},**

**};**

**public** string Convert**(**int number**)**

**{**

**if** **(**Results**.**ContainsKey**(**number**))**

**{**

**return** Results**[**number**];**

**}**

string result **=** string**.**Empty**;**

**while** **(**number **>=** 40**)**

**{**

result **+=** "XL"**;**

number **-=** 40**;**

**}**

**while** **(**number **>=** 10**)**

**{**

result **+=** "X"**;**

number **-=** 10**;**

**}**

**while** **(**number **>=** 5**)**

**{**

result **+=** "V"**;**

number **-=** 5**;**

**}**

**while** **(**number **>=** 4**)**

**{**

result **+=** "IV"**;**

number **-=** 4**;**

**}**

**while** **(**number **>=** 1**)**

**{**

result **+=** "I"**;**

number **-=** 1**;**

**}**

**return** result**;**

**}**

**}**

We apply the same *if to while* transformation to the remaining if statement and this allows us to get rid of the duplicated while statements. In order for this to work it’s more convenient to have the **dictionary reversed so we also do this.**

**// 10 if -> while**

**public class RomanConverter**

**{**

**public static readonly IDictionary<int, string> mappings =**

**new Dictionary<int, string>**

**{**

**{40, "XL"},**

**{**10**,** "X"**},**

**{**9**,** "IX"**},**

**{**5**,** "V"**},**

**{**4**,** "IV"**},**

**{**1**,** "I"**},**

**};**

**public** string Convert**(**int number**)**

**{**

string result **=** string**.**Empty**;**

var mappingsEnumerator **=** mappings**.**GetEnumerator**();**

**while** **(**mappingsEnumerator**.**MoveNext**())**

**{**

var mapping **=** mappingsEnumerator**.**Current**;**

**while** **(**number **>=** mapping**.**Key**)**

**{**

result **+=** mapping**.**Value**;**

number **-=** mapping**.**Key**;**

**}**

**}**

**return** result**;**

**}**

**}**

We add more failing tests but the last transformation is sufficient to make all new tests pass. So we refactor the code to make it more readable and we are done. We decided not to refactor the outer while loop to a foreach loop. Although this could simplify the code, a foreach loop is not in the transformation premise list so we decided against it for the purposes of this post.

**[**TestFixture**]**

**public** class RomanConverterShould

**{**

**[**TestCase**(**1**,** "I"**)]**

**[**TestCase**(**2**,** "II"**)]**

**[**TestCase**(**3**,** "III"**)]**

**[**TestCase**(**4**,** "IV"**)]**

**[**TestCase**(**5**,** "V"**)]**

**[**TestCase**(**6**,** "VI"**)]**

**[**TestCase**(**7**,** "VII"**)]**

**[**TestCase**(**8**,** "VIII"**)]**

**[**TestCase**(**9**,** "IX"**)]**

**[**TestCase**(**10**,** "X"**)]**

**[**TestCase**(**40**,** "XL"**)]**

**[**TestCase**(**50**,** "L"**)]**

**[**TestCase**(**90**,** "XC"**)]**

**[**TestCase**(**100**,** "C"**)]**

**[**TestCase**(**400**,** "CD"**)]**

**[**TestCase**(**500**,** "D"**)]**

**[**TestCase**(**900**,** "CM"**)]**

**[**TestCase**(**1000**,** "M"**)]**

**[**TestCase**(**846**,** "DCCCXLVI"**)]**

**[**TestCase**(**1999**,** "MCMXCIX"**)]**

**[**TestCase**(**2008**,** "MMVIII"**)]**

**public** void ConvertNumberToRoman**(**int number**,** string expected**)**

**{**

var romanNumeral **=** **new** RomanConverter**().**Convert**(**number**);**

Assert**.**That**(**romanNumeral**,** Is**.**EqualTo**(**expected**));**

**}**

**}**

**public** class RomanConverter

**{**

**public** static **readonly** IDictionary**<**int**,** string**>** arabicsToRomans **=**

**new** Dictionary**<**int**,** string**>**

**{**

**{**1000**,** "M"**},**

**{**900**,** "CM"**},**

**{**500**,** "D"**},**

**{**400**,** "CD"**},**

**{**100**,** "C"**},**

**{**90**,** "XC"**},**

**{**50**,** "L"**},**

**{**40**,** "XL"**},**

**{**10**,** "X"**},**

**{**9**,** "IX"**},**

**{**5**,** "V"**},**

**{**4**,** "IV"**},**

**{**1**,** "I"**},**

**};**

**public** string Convert**(**int number**)**

**{**

var romanNumeral **=** string**.**Empty**;**

var arabicsToRomansEnumerator **=** arabicsToRomans**.**GetEnumerator**();**

**while** **(**arabicsToRomansEnumerator**.**MoveNext**())**

**{**

var arabicToRoman **=** arabicsToRomansEnumerator**.**Current**;**

var arabicNumeral **=** arabicToRoman**.**Key**;**

var romanNumeral **=** arabicToRoman**.**Value**;**

**while** **(**number **>=** arabicNumeral**)**

**{**

result **+=** romanNumeral**;**

number **-=** arabicNumeral**;**

**}**

**}**

**return** romanNumeral**;**

**}**

**}**

**}**

### **Conclusion**

As with any technique or practice, after you experiment it you should decide if you find it useful and should embrace it or not. Another option is to see if you can shape it to your needs. For example do you think it makes sense to have an order or scale of complexity? Good do you agree with Uncle Bob’s scale? If a scale still makes sense then come up with your own, or make slight changes to Uncle Bob’s scale until you find one that works for you.

## Links

<http://blog.8thlight.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html>

<http://en.wikipedia.org/wiki/Transformation_Priority_Premise>

<http://www.javacodegeeks.com/2013/01/tdd-and-the-transformation-priority-premise.html>

<https://vimeo.com/97516288>

# Design I

*“Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away.”*

Antoine de Saint-Exupery

TDD is not enough

DRY is not enough.

TDD will punish you if you don’t understand design

If your tests are difficult to write your design is probably wrong

### Object Calisthenics, 9 steps to better software design

* Only one level of indentation per method
* Don't use the ELSE keyword
* Wrap ~~all~~ primitives and strings, *if it has behaviour*
* First class collections
* One dot per line
* Don't abbreviate
* Keep all entities small
  + 10 files per package
  + 50 lines per class
  + 5 lines per method
  + 2 arguments per method
* No classes with more than two instance variables
* No getters/setters/properties

### Use only one level of indentation per method.

Ever stare at a big old method wondering where to start? A giant method lacks cohesiveness. One guideline is to limit method length to 5 lines, but that kind of transition can be daunting if your code is littered with 500-line monsters. Instead, try to ensure that each method does exactly one thing – one control structure, or one block of statements, per method. If you’ve got nested control structures in a method, you’re working at multiple levels of abstraction, and that means you’re doing more than one thing.

As you work with methods that do \*exactly\* one thing, expressed within classes doing exactly one thing, your code begins to change. As each unit in your application becomes smaller, your level of re-use will start to rise exponentially. It can be difficult to spot opportunities for reuse within a method that has five responsibilities and is implemented in 100 lines. A three-line method that manages the state of a single object in a given context is usable in many different contexts.

Use the Extract Method feature of your IDE to pull out behaviours until your methods only have one level of indentation.

### Don’t use the else keyword

Every programmer understands the if/else construct. It’s built into nearly every programming language, and simple conditional logic is easy for anyone to understand. Nearly every programmer has seen a nasty nested conditional that’s impossible to follow, or a case statement that goes on for pages. Even worse, it is all too easy to simply add another branch to an existing conditional rather than factoring to a better solution. Conditionals are also a frequent source of duplication.

Object-oriented languages give us a powerful tool, polymorphism, for handling complex conditional cases. Designs that use polymorphism can be easier to read and maintain, and express their intent more clearly. But it’s not always easy to make the transition, especially when we have ELSE in our back pocket. So as part of this exercise, you’re not allowed to use ELSE. Try the Null Object pattern; it may help in some situations. There are other tools that can help you rid yourself of the else as well. See how many alternatives you can come up with.

### Wrap all primitives and strings

In C# int is a primitive, not a real object, so it obeys different rules than objects. It is used with a syntax that isn’t object-oriented. More importantly, an int on its own is just a scalar, so it has no meaning. When a method takes an int as a parameter, the method name needs to do all of the work of expressing the intent. If the same method takes an Hour as a parameter, it’s much easier to see what’s going on. Small objects like this can make programs more maintainable, since it isn’t possible to pass a Year to a method that takes an Hour parameter. With a primitive variable the compiler can’t help you write semantically correct programs. With an object, even a small one, you are giving both the compiler and the programmer additional info about what the value is and why it is being used.

Small objects like Hour or Money also give us an obvious place to put behavior that would otherwise have been littered around other classes. This becomes especially true when you apply the Rule X, and only the small object can access the value.

### Use only one dot per line

Sometimes it’s hard to know which object should take responsibility for an activity. If you start looking for lines of code with multiple dots, you’ll start to find many misplaced responsibilities. If you’ve got more than one dot on any given line of code, the activity is happening in the wrong place. Maybe your object is dealing with two other objects at once. If this is the case, your object is a middleman; it knows too much about too many people. Consider moving the activity into one of the other objects.

If all those dots are connected, your object is digging deeply into another object. These multiple dots indicate that you’re violating encapsulation. Try asking that object to do something for you, rather than poking around its insides. A major part of encapsulation is not reaching across class boundaries into types that you shouldn’t know about.

The Law of Demeter (“Only talk to your friends”) is a good place to start, but think about it this way: You can play with your toys, toys that you make, and toys that someone gives you. You don’t ever, \*ever\* play with your toy’s toys.

### Don’t abbreviate

It’s often tempting to abbreviate in the names of classes, methods, or variables. Resist the temptation – abbreviations can be confusing, and they tend to hide larger problems.

Think about why you want to abbreviate. Is it because you’re typing the same word over and over again? If that’s the case, perhaps your method is used too heavily and you are missing opportunities to remove duplication. Is it because your method names are getting long? This might be a sign of a misplaced responsibility, or a missing class.

### Keep all entities small

This means no class over 50 lines and no package over 10 files. Classes over 50 lines usually do more than one thing, which makes them harder to understand and harder to reuse. 50-line classes have the added benefit of being visible on one screen without scrolling, which makes them easier to grasp quickly.

What’s challenging about creating such small classes is that there are often groups of behaviors that make logical sense together. This is where we need to leverage packages. As your classes become smaller and have fewer responsibilities, and as you limit package size, you’ll start to see that packages represent clusters of related classes that work together to achieve a goal. Packages, like classes, should be cohesive and have a purpose. Keeping those packages small forces them to have a real identity.

### Don’t use any classes with more than two instance variables

Most classes should simply be responsible for handling a single state variable, but there are a few that will require two. Adding a new instance variable to a class immediately decreases the cohesion of that class. In general, while programming under these rules, you’ll find that there are two kinds of classes, those that maintain the state of a single instance variable, and those that coordinate two separate variables. In general, don’t mix the two kinds of responsibilities.

### Use first-class collections

Application of this rule is simple: any class that contains a collection should contain no other member variables. Each collection gets wrapped in its own class, so now behaviors related to the collection have a home. You may find that filters become a part of this new class. Also, your new class can handle activities like joining two groups together or applying a rule to each element of the group.

### Don’t use any getters/setters/properties

Another way this rule is commonly stated is “Tell, don’t ask”. Tell-Don't-Ask is a principle that helps people remember that object-orientation is about bundling data with the functions that operate on that data. It reminds us that rather than asking an object for data and acting on that data, we should instead tell an object what to do. This encourages to move behaviour into an object to go with the data.

## Katas

Ruleset

* Only one level of indentation per method
* Don't use the ELSE keyword
* Wrap ~~all~~ primitives and strings, *if it has behaviour*
* First class collections
* One dot per line
* Don't abbreviate
* Keep all entities small
  + 10 files per package
  + 50 lines per class
  + 5 lines per method
  + 2 arguments per method
* No classes with more than two instance variables
* No getters/setters/properties

### Tic Tac Toe

#### Rules

* X always goes first.
* Players alternate placing Xs and Os on the board until either:
  + One player has three in a row, horizontally, vertically or diagonally
  + All nine squares are filled.
* If a player is able to draw three Xs or three Os in a row, that player wins.
* If all nine squares are filled and neither player has three in a row, the game is a draw.

### Game of Life

<https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life>

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead. Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

* Any live cell with fewer than two live neighbours dies, as if caused by under-population.
* Any live cell with two or three live neighbours should live on to the next generation.
* Any live cell with more than three live neighbours dies, as if by over-population.
* Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

## Links

[www.xpteam.com/jeff/writings/objectcalisthenics.rtf](http://www.xpteam.com/jeff/writings/objectcalisthenics.rtf)

<http://www.frandieguez.com/blog/2012/12/object-calisthenics-write-better-object-oriented-code/>

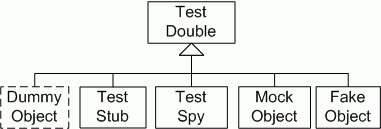
<http://www.slideshare.net/KLabCyscorpions-TechBlog/object-calisthenics-34557136>

# Test doubles

Test doubles is an extension to Test-Driven Development that supports good Object-Oriented design by guiding the discovery of a coherent system of types within a code base. It turns out to be less interesting as a technique for isolating tests from third-party libraries than is widely thought.

## Different types of test doubles

* **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
* **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.
* **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).
* **Mocks** are pre-programmed with expectations which form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting.
* **Spies** are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.



## Principles

### Allow Queries; Expect Commands

* Use Mocks for Commands
  + Commands are all about side effects, and Mocks are all about [Behaviour Verification](http://xunitpatterns.com/Behavior%20Verification.html): that is, that side effects occurred
* Use Stubs for Queries
  + Stubs mainly exist to 'make happy noises', and one of the ways they have to do that, is to return data from dependencies, when return data is required.

### Only mock classes you own

Mock Objects is a design technique so programmers should only write mocks for types that they can change. Otherwise they cannot change the design to respond to requirements that arise from the process. Programmers should not write mocks for fixed types, such as those defined by the runtime or external libraries. Instead they should write thin wrappers to implement the application abstractions in terms of the underlying infrastructure. Those wrappers will have been defined as part of a need-driven test. We have found this to be a powerful insight to help programmers understand the technique. It restores the pre-eminence of the design in the use of Mock Objects, which has often been overshadowed by its use for testing interactions with third-party libraries.

### Specify as little as possible in a test

When testing with Mock Objects it is important to find the right balance between an accurate specification of a unit's required behaviour and a flexible test that allows easy evolution of the code base. One of the risks with TDD is that tests become “brittle”, that is they fail when a programmer makes unrelated changes to the application code. They have been over-specified to check features that are an artefact of the implementation, not an expression of some requirement in the object. A test suite that contains a lot of brittle tests will slow down development and inhibit refactoring. The solution is to re-examine the code and see if either the specification should be weakened, or the object structure is wrong and should be changed. Following Einstein, a specification should be as precise as possible, but not more precise.

### Don’t use mocks to test boundary objects

If an object has no relationships to other objects in the system, it does not need to be tested with mock objects. A test for such an object only needs to make assertions about values returned from its methods. Typically, these objects store data, perform independent calculations or represent atomic values. While this may seem an obvious thing to say, we have encountered people trying to use mock objects where they don’t actually need to.

### Don’t add behaviour

Mock objects are still stubs and should not add any additional complexity to the test environment, their behaviour should be obvious [10]. We find that an urge to start adding real behaviour to a mock object is usually a symptom of misplaced responsibilities. A common example of this is when one mock has to interpret its input to return another mock, perhaps by parsing an event message. This introduces a risk of testing the test infrastructure rather than the target code. For example:

mock.expect(once()) .method("retrieve").with(eq(KEY1)) .willReturn(VALUE1);

mock.expect(once()) .method("retrieve").with(eq(KEY2)) .willReturn(VALUE2);

### Only mock your immediate neighbours

An object that has to navigate a network of objects in its implementation is likely to be brittle because it has too many dependencies. One symptom of this is tests that are complex to set up and difficult to read because they have to construct a similar network of mock objects. Unit tests work best when they focus on testing one thing at a time and only setting expectations on objects that are nearest neighbours. The solution might be to check that you are testing the right object, or to introduce a role to bridge between the object and its surroundings.

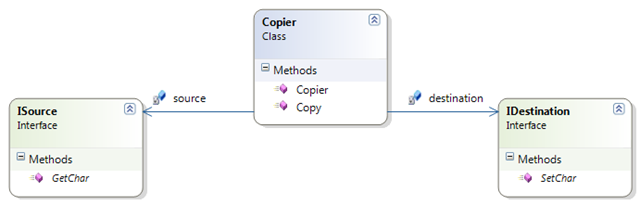
### Too Many Mocks

A similar problem arises when a test has to pass too many mock objects to the target code, even if they are all immediate neighbours. Again, the tests is likely to be complex to set up and hard to read. Again the solution might be to change misaligned responsibilities, or to introduce an intermediate role. Alternatively, it is possible that the object under test is too large and should be broken up into smaller objects that will be more focussed and easier to test

## Katas

### Character Copier kata

The character copier is a simple class that reads characters from a source and copies them to a destination one character at a time.

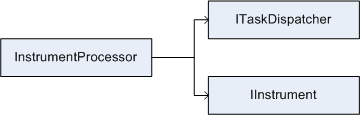


When the method Copy is called on the copier then it should read characters from the source and copy them to the destination until the source returns a newline (‘\n’).

The exercise is to implement the character copier using mocks for the source and the destination (try using spies (manually written mocks) and mocks written with a mocking framework.

### Instrument Processor Kata

In the Instrument Processor Kata, we are going to implement a class that gets tasks from a TaskDispatcher and executes them on an Instrument.



The InstrumentProcessor – the class to develop– has to implement the following interface:

public interface IInstrumentProcessor

{

void Process();

}

The dependencies that the InstrumentProcessor can use to do its job are a task dispatcher with the following interface:

public interface ITaskDispatcher

{

string GetTask();

void FinishedTask(string task);

}

The GetTask method returns the next task to execute on the instrument.

After the task was successfully executed on the instrument, the FinishedTask method has to be called by the InstrumentProcessor passing the task that was completed as the method argument.

The second dependency of the InstrumentProcessor is an instrument with the following interface:

public interface IInstrument

{

void Execute(string task);

event EventHandler Finished;

event EventHandler Error;

}

The Execute method starts the instrument, which will begin to execute the task passed to it. The method will return immediately (we assume that the instrument implementation is asynchronous).

The Execute method will throw an ArgumentNullException if the task passed in is null.

When the instrument finished executing then the Finished event is fired.

When the instrument detects an error situation during executing (note that the Execute method will already have returned the control flow to the caller due to its asynchronous implementation) then it fires the Error event.

The exercise is to implement the InstrumentProcessor in a way that

When the method Process is called then the InstrumentProcessor gets the next task from the task dispatcher and executes it on the instrument.

When the Execute method of the instrument throws an exception then this exception is passed on to the caller of the Process method.

When the instrument fires the Finished event then the InstrumentProcessor calls the task dispatcher’s FinishedTask method with the correct task.

When the instrument fires the Error event then the InstrumentProcessor writes the string “Error occurred” to the console.

## Links

<https://blog.8thlight.com/uncle-bob/2014/05/14/TheLittleMocker.html>

<https://en.wikipedia.org/wiki/Mock_object>

\*\* <http://martinfowler.com/articles/mocksArentStubs.html>

<http://www.jmock.org/oopsla2004.pdf>

<http://blog.8thlight.com/uncle-bob/2014/05/10/WhenToMock.html>

<http://codemanship.co.uk/parlezuml/blog/?postid=1225>

<http://codemanship.co.uk/parlezuml/blog/?postid=1167>

\*\* <http://blog.ploeh.dk/2013/10/23/mocks-for-commands-stubs-for-queries/>

# Design II

*“Design is choosing how you will fail.”*

Ron Fein

## Code Smells

A code smell is a surface indication that usually corresponds to a deeper problem in the system.

Firstly a smell is by definition something that's quick to spot. A long method is a good example of this just looking at the code and my nose twitches if I see more than a dozen lines of code.

Secondly smells aren't inherently bad on their own they are often an indicator of a problem rather than the problem themselves.

### Rigidity

The software is difficult to change. A small change causes a cascade of subsequent changes.

### Fragility

The software breaks in many places due to a single change.

The difference between fragility and rigidity is that fragile apps are rigid but it’s difficult to see the rigidity.

### Immobility

You cannot reuse parts of the code in other projects because of involved risks and high effort. Code is entangled.

### Viscosity

#### Viscosity of Design

Taking a shortcut and introducing technical debt requires less effort than doing it right.

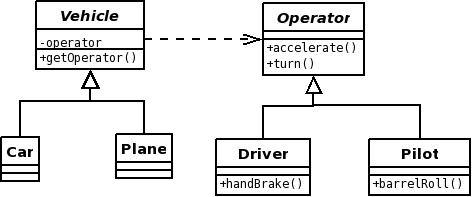
#### Viscosity of Environment

Building, testing and other tasks take a long time. Therefore, these activities are not executed properly by everyone and technical debt is introduced.

### List

|  |  |  |  |
| --- | --- | --- | --- |
| [**Duplicated Code**](https://sourcemaking.com/refactoring/duplicated-code)  Identical or very similar code exists in more than one location or duplicated knowledge. **DRY** violation. | [**Long Method**](https://sourcemaking.com/refactoring/long-method)  Methods should do only one thing. **SRP** violation.  “Functions should do one thing. Should do it well. Should do it only”. One level of abstraction. More than 5 to 15 lines. | [**Large Class**](https://sourcemaking.com/refactoring/large-class)  Classes should have only one responsibility. **SRP** violation.  More than 50 lines per class. | [**Long Parameter List**](https://sourcemaking.com/refactoring/long-parameter-list)  Ideal: 0 niladic Ok: 1 monadic Acceptable: 2 dyadic Debatable: 3 triadic avoid Special justification: 3+ polyadic.  **Connascence of position.** |
| [**Divergent Change**](https://sourcemaking.com/refactoring/divergent-change)  One class is commonly changed in different ways for different reasons. **OCP, SRP** violation. God class. | [**Shotgun Surgery**](https://sourcemaking.com/refactoring/shotgun-surgery)  Opposite of Divergent change. One change, forces lots of little changes in different classes. **DRY** violation. | [**Feature Envy**](https://sourcemaking.com/refactoring/feature-envy)  A class that uses methods or properties of another class excessively.  **Cohesion** violation. | [**Data Clumps**](https://sourcemaking.com/refactoring/data-clumps)  Same data items together in lots of places.  Special case of duplicated code. **Dry** violation |
| [**Primitive Obsession**](https://sourcemaking.com/refactoring/primitive-obsession)  Don't use primitive types as substitutes for classes. If the data type is sufficiently complex, use a class to represent it. **Connascence of meaning**. | [**Switch Statements**](https://sourcemaking.com/refactoring/switch-statements)  Can lead to same switch statement scattered about a program in different places. Can lead to **Dry** violation | [**Parallel Inheritance Hierarchies**](https://sourcemaking.com/refactoring/parallel-inheritance-hierarchies)  Special case of shotgun surgery. Creating a subclass of one class, forces subclass of another. | [**Lazy Class**](https://sourcemaking.com/refactoring/lazy-class)  A class that does too little. May be acting only as middle man or a data class or can be caused by speculative generality. **Cohesion** violation. |
| [**Speculative Generality**](https://sourcemaking.com/refactoring/speculative-generality)  The only users of a method or class are test cases. **YAGNI** violation. | [**Temporary Field**](https://sourcemaking.com/refactoring/temporary-field)  Class contains an instance variable set only in certain circumstances. **Cohesion** violation. | [**Message Chains**](https://sourcemaking.com/refactoring/message-chains)  Too many dots: Object.Prop.Prop  **Law of Demeter** violation. | [**Middle Man**](https://sourcemaking.com/refactoring/middle-man)  If a class is delegating all its work, cut out the middleman. Beware classes that are wrappers over other classes or existing functionality. Special case of **lazy class**. |
| [**Inappropriate Intimacy**](https://sourcemaking.com/refactoring/inappropriate-intimacy)  A class that has dependencies on implementation details of another class. Special case of feature envy. **Cohesion** violation. | [**Alternative Classes with Different Interfaces**](https://sourcemaking.com/refactoring/alternative-classes-with-different-interfaces)  If two classes are similar on the inside, but different on the outside, perhaps they can be modified to share a common interface. | [**Incomplete Library Class**](https://sourcemaking.com/refactoring/incomplete-library-class)  Adding missing functionality by not changing library can lead to functionality implemented in odd places. | [**Data Class**](https://sourcemaking.com/refactoring/data-class)  Classes that have fields, properties, and nothing else. Anaemic classes that contain no behaviour. Special case of lazy class. |
| [**Refused Bequest**](https://sourcemaking.com/refactoring/refused-bequest)  Usually means the hierarchy is wrong. **LSP** violation. | [**Comments**](https://sourcemaking.com/refactoring/comments)  Make effort to create code that expresses intent instead of comments. | Dead Code  Code that has no references to it, commented code. |  |

##### Parallel Inheritance Hierarchies



### Catalogue

|  |  |  |
| --- | --- | --- |
| **Group** | **Smells** | **Description** |
| **The bloaters**  *Immobility*  Needless Repetition  Needless Complexity  Opacity | Long method  Large class  Primitive obsession  Long parameter list  Data clumps | Bloater smells represents something that has grown so large that it cannot be effectively handled.  It seems likely that these smells grow a little bit at a time. Hopefully nobody designs, e.g., Long Methods.  Primitive Obsession is actually more of a symptom that causes bloats than a bloat itself. The same holds for Data Clumps*.*When a Primitive Obsession exists, there are no small classes for small entities (e.g. phone numbers). Thus, the functionality is added to some other class, which increases the class and method size in the software. With Data Clumps there exists a set of primitives that always appear together (e.g. 3 integers for RGB colors). Since these data items are not encapsulated in a class this increases the sizes of methods and classes. |
| **The object-orientation abusers**  *Viscosity of Design*  Opacity | Switch statements  Temporary field  Refused bequest  Alternative classes with different interfaces | The common denominator for the smells in the Object-Orientation Abuser category is that they represent cases where the solution does not fully exploit the possibilities of object-oriented design.  For example, a Switch Statement might be considered acceptable or even good design in procedural programming, but is something that should be avoided in object-oriented programming. The situation where switch statements or type codes are needed should be handled by creating subclasses. Parallel Inheritance Hierarchies and Refused Bequest smells lack proper inheritance design, which is one of the key elements in object-oriented programming. The Alternative Classes with Different Interfaces smell lacks a common interface for closely related classes, so it can also be considered a certain type of inheritance misuse. The Temporary Field smell means a case in which a variable is in the class scope, when it should be in method scope. This violates the information hiding principle. |
| **The change preventers**  *Rigidity*  *Fragility* | Divergent change  Shotgun surgery  Parallel inheritance hierarchies | Change Preventers are smells is that hinder changing or further developing the software  These smells violate the rule suggested by Fowler and Beck which says that classes and possible changes should have a one-to-one relationship. For example, changes to the database only affect one class, while changes to calculation formulas only affect the other class.  The Divergent Change smell means that we have a single class that needs to be modified by many different types of changes. With the Shotgun Surgery smell the situation is the opposite, we need to modify many classes when making a single change to a system (change several classes when changing database from one vendor to another)  Parallel Inheritance Hierarchies, which means a duplicated class hierarchy, was originally placed in OO-abusers. One could also place it inside of The Dispensables since there is redundant logic that should be replaced. |
| **The dispensables**  *Viscosity of design*  Needless Complexity  Needless Repetition  Opacity | Lazy class  Data class  Duplicate code  Dead code  Speculative generality | The common thing for the Dispensable smells is that they all represent something unnecessary that should be removed from the source code.  This group contains two types of smells (dispensable classes and dispensable code), but since they violate the same principle, we will look at them together. If a class is not doing enough it needs to be removed or its responsibility needs to be increased. This is the case with the Lazy class and the Data class smells. Code that is not used or is redundant needs to be removed. This is the case with Duplicate Code, Speculative Generality and Dead Code smells. |
| **The couplers**  *Rigidity*  *Fragility*  Needless Complexity | Feature envy  Inappropriate intimacy  Message chains  Middle man | This group has four coupling-related smells.  One design principle that has been around for decades is low coupling (Stevens et al. 1974). This group has 3 smells that represent high coupling. Middle Man smell on the other hand represents a problem that might be created when trying to avoid high coupling with constant delegation. Middle Man is a class that is doing too much simple delegation instead of really contributing to the application.  The Feature Envy smell means a case where one method is too interested in other classes, and the Inappropriate Intimacy smell means that two classes are coupled tightly to each other. Message Chains is a smell where class A needs data from class D. To access this data, class A needs to retrieve object C from object B (A and B have a direct reference). The following example illustrates the message chain smell: A.getB().getC().getD().getTheNeededData()  Of course, one could make an argument that these smells should belong to the Object-Orientation abusers group, but since they all focus strictly on coupling, it makes the taxonomy more understandable if they are introduced in a group of their own. |

## Katas

#### Code Smell exercises (TFS)

## Links

##### <http://sourcemaking.com/refactoring/bad-smells-in-code>

##### <http://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf>

##### <http://blog.codinghorror.com/code-smells/>

##### <http://codurance.com/2016/03/03/cohesion-cornerstone-software-design/>

##### <http://connascence.io/pages/about.html>

##### <http://elijahmanor.com/javascript-smells/>

##### <http://elijahmanor.com/talks/js-smells/#/14>

**Testing Legacy Code**

“A common fallacy is to assume authors of incomprehensible code will be able to express themselves clearly in comments.”

Kevlin Henney

**Characterisation Tests**

Characterization test is a means to describe (characterize) the actual behaviour of an existing piece of software, and therefore protect existing behaviour of legacy code against unintended changes via automated testing. This term was coined by Michael Feathers.

Michael Feathers defines characterization tests as tests that characterize the actual behaviour of a piece of code. In other words, they don’t check what the code is supposed to do, as specification tests do, but what the code actually and currently does.

Having a set of characterization tests helps developers working with legacy code because they can run those tests after modifying their code and make sure that their modification did not cause any unintended or unwanted changes in functionality somewhere else.

1. Use a piece of code in a test harness.
2. Write an assertion that you know will fail.
3. Run the test and let the failure tell you what the actual behaviour is.
4. Change the test so that it expects the behaviour that the code actually produces.
5. Repeat.

**Katas**

<https://github.com/sandromancuso/trip-service-kata>

<https://github.com/emilybache/GildedRose-Refactoring-Kata>

**Golden Master**

The Golden Master technique is very useful when a clear input and output is easy to obtain on the system level. There are some cases where the Golden Master technique can be applied with difficulty or where it cannot be applied at all. We will discuss these situations further as well.

For all the given situations we need to think if the system tests generated with the Golden Master are enough, or we need to start adding other types of tests like unit tests, component tests, integration tests, security tests, e**tc.**

**The Golden Master technique is based on the following steps:**

1. Find the way the system delivers its outputs
   * Check for clear outputs of the system: console, data layer, logger, file system, etc.
2. Find a way to capture the output of the system without changing the production code
   * Think if that output can be “stolen” without changing the production code. For example you could “steal” the console output by redirecting the stream of the console to an in-memory stream. Another example would be injecting an implementation of a data layer interface that would write to another data source than the production code would.
3. Find a pattern of the outputs
   * The output of the system could be text or a data structures tree or another type of stream. Starting from this output type you can decide if you could go on to the next step.
4. Generate enough random inputs and persist the tuple input/output
   * In the case you can persist the outputs to a simple data repository, you can think about what are the inputs for the corresponding outputs. Here the magic consists in finding a good balance of random or pseudo-random inputs. We need inputs that would cover most of the system with tests, but in the same time the test would run in seconds. For example in the case of a text generating algorithm we need to decide if we want to feed the system with 1000 or one million entry data. Maybe 10 000 input data are enough. We will anyway check the tests coverage during the last stage.
   * We need to persist the pair input-output. The output would be the Golden Master, the reference data we will always check our SUT against.
5. Write a system test to check the SUT against the previously persisted data
   * Now that we have a way to “steal” the outputs and we have a guess about how to generate enough input-output pairs, but not too many, we can call the system and check it against the outputs for a given input. We need to check if the test touches the SUT and if it passes. We also need to check that the test runs fast enough, in seconds.
6. Commit the test
   * Always when a green test passes we need to commit the code to a local source control system. Why? So that we can easily revert to a stable testable system.
   * Important: Do not forget to commit also the golden masters (files, database, etc)!
7. Check test behaviour and coverage
   * In this stage I tend to do two things:
     + Use a test coverage tool to see where the system tests touch the SUT
     + Start changing the SUT in order to see the golden master test go red.
       - If the test does not go red, I can understand that the code base is not covered by tests in that area and I should not touch it during the next stages, until I have a basic safety net. Always after this step I will revert to the previous commit, not matter how small the change to the SUT was.
8. If not enough behaviours are covered, go to 3
   * In the case we found during the last stage some behaviours that were not covered by the golden master test, we need to write some more tests with other inputs and outputs. We go on until all the visible behaviours needing to be covered by tests are covered.

**Outcomes**

After this session we will have a basic safety net composed of system tests. These system tests will check the SUT against the golden masters. A very important output of the system would be the golden masters persisted in any form. All the system tests would need always to be called during the next refactoring phases.

**Kata**

<https://github.com/emilybache/GildedRose-Refactoring-Kata>

**Links**

<http://en.wikipedia.org/wiki/Characterization_test>

<http://www.infoq.com/news/2007/03/characterization-testing>

<http://www.slideshare.net/nashjain/working-effectively-with-legacy-code-presentation>

<http://blog.adrianbolboaca.ro/2014/05/golden-master/>

<http://java.dzone.com/articles/testing-legacy-code-golden>

<http://blog.thecodewhisperer.com/2014/09/28/surviving-legacy-code-with-golden-master-and-sampling/>

<http://approvaltests.com/>

# Design III

*“When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, and then add the feature.”*

Martin Fowler

## Refactoring I

“Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour”. Martin Fowler.

Refactor, not because you *know* the abstraction but because you want to *find* it.

Refactor if you answer “no” to any of the following questions regarding a module/class/method:

* Is it DRY?
* Does it have one responsibility?
* Does everything in it change at the same rate?
* Does it depend on things that change less often than it does?

## Resharper catalogue

|  |  |  |
| --- | --- | --- |
| *Rename*  Namespaces, types, methods, parameters, local variables, properties, fields and events | *Move*  Static Member, String to Resource to Folder. Type to Another File or Namespace, Type to Outer Scope. Types into Matching Files | *Safe delete*  Namespaces, types, methods, parameters, local variables, properties, fields and events |
| *Extract*  Interface, Method, Superclass, Class from parameters | *Introduce*  Field, Parameter, Variable | *Inline*  Field, Method, Variable |
| *Change signature*  Add, remove, rename, or reorder parameter(s), Change return type, Change parameter type(s), Rename method | *Pull Members Up* | *Push Members Down* |
| *Replace Constructor with Factory Method* | *Transform Out Parameters to Tuple* | *Use Base Type where Possible* |
| *Make*  Method Non-Static, Make Method Static | *Encapsulate Field*  With property or method | *Copy Type* |

|  |  |  |
| --- | --- | --- |
| **Group** | **Refactors** | **Description** |
| Composing methods | Extract Method  Inline Method  Extract Variable  Inline Temp  Replace Temp with Query  Split Temporary Variable  Remove Assignments to Parameters  Replace Method with Method Object  Substitute Algorithm | The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements. |
| Moving Features between Objects | Move Method  Move Field  Extract Class  Inline Class  Hide Delegate  Remove Middle Man  Introduce Foreign Method  Introduce Local Extension | These refactoring techniques show how to safely move functionality between classes, create new classes, and hide implementation details from public access. |
| Organizing Data | Self-Encapsulate Field  Replace Data Value with Object  Change Value to Reference  Change Reference to Value  Replace Array with Object  Duplicate Observed Data  Change Unidirectional Association to Bidirectional  Change Bidirectional Association to Unidirectional  Replace Magic Number with Symbolic Constant  Encapsulate Field  Encapsulate Collection  Replace Type Code with Class  Replace Type Code with Subclasses  Replace Type Code with State/Strategy  Replace Subclass with Fields | These refactoring techniques help with data handling, replacing primitives with rich class functionality. Another important result is untangling of class associations, which makes classes more portable and reusable. |
| Simplifying Conditional Expressions | Decompose Conditional  Consolidate Conditional Expression  Consolidate Duplicate Conditional Fragments  Remove Control Flag  Replace Nested Conditional with Guard Clauses  Replace Conditional with Polymorphism  Introduce Null Object  Introduce Assertion | Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well. |
| Simplifying Method Calls | Rename Method  Add Parameter  Remove Parameter  Separate Query from Modifier  Parameterize Method  Replace Parameter with Explicit Methods  Preserve Whole Object  Replace Parameter with Method Call  Introduce Parameter Object  Remove Setting Method  Hide Method  Replace Constructor with Factory Method  Replace Error Code with Exception  Replace Exception with Test | These techniques make method calls simpler and easier to understand. This, in turn, simplifies the interfaces for interaction between classes. |
| Dealing with Generalisation | Pull Up Field  Pull Up Method  Pull Up Constructor Body  Push Down Method  Push Down Field  Extract Subclass  Extract Superclass  Extract Interface  Collapse Hierarchy  Form Template Method  Replace Inheritance with Delegation  Replace Delegation with Inheritance | Abstraction has its own group of refactoring techniques, primarily associated with moving functionality along the class inheritance hierarchy, creating new classes and interfaces, and replacing inheritance with delegation and vice versa. |

## Techniques

* Leave things better than you found them
* Never be more than 2 minutes away from checking in and going home
  + Try to stay in the green for as long as possible
  + Commit as often as possible
    - Allows you to work in small and safe increments
    - Allows easy rollbacks
* Arm yourself
  + Tests
    - Don’t change production code that is not covered by tests
    - If you need to add tests start from shortest to deepest branch
      * Characterization tests
      * Golden master tests
  + Tools (Resharper, NCrunch)
    - Prefer IDE automated refactoring
  + Patterns
    - Refactor to design patterns when appropriate
    - Detect code smells and apply refactors
* Remove the 3 C’s
  + Clutter – Anything in your code that does not add value
    - Format the code
    - Delete dead code
    - Delete useless comments
    - Delete unnecessary code
  + Complexity
    - Rename bad names
    - Extract smaller methods from long methods
    - Extract methods from deep conditionals
    - Extract constants from magic numbers and strings
    - Refine scope for improper variable scoping
    - Encapsulate where we find missing encapsulation
  + Cleverness – “If it’s simple and elegant, you wouldn’t refer to it as ‘clever’”
    - Make abbreviated code explicit
    - Encapsulate cryptic code in methods
* Remove the 3 D’s
  + Duplication
  + Duplication
  + Duplication

## Katas

<https://github.com/emilybache/Tennis-Refactoring-Kata>

### Resharper Workshop

<https://github.com/JetBrains/resharper-workshop>

<https://github.com/rasmuskl/ReSharperCourse/blob/master/Source/ReSharper%20Exercises.pdf>

## Links

<https://www.jetbrains.com/resharper/features/code_refactoring.html>

## Books

http://martinfowler.com/books/refactoring.html

# Design IV

### Smell -> Refactor catalogue

|  |  |  |
| --- | --- | --- |
| **Switch Statements**  Replace Conditional with Polymorphism  Replace Type Code with Subclasses  Replace Type Code with State/Strategy  Move Accumulation to Visitor  Replace Conditional Dispatcher with Command  Replace Parameter with Explicit Methods  Introduce Null Object | **Inappropriate Intimacy**  Move Method  Move Field  Extract Class  Hide Delegate  Replace Inheritance with Delegation | **Large Class**  Extract Class  Extract Subclass  Extract Interface  Replace Data Value with Object  Replace Conditional Dispatcher with Command  Replace Implicit Language with Interpreter  Replace State-Altering Conditionals with State |
| **Primitive Obsession**  Replace Data Value with Object  Encapsulate Composite with Builder  Introduce Parameter Object  Extract Class  Move Embellishment to Decorator  Replace Conditional Logic with Strategy  Replace Implicit Language with Interpreter  Replace Implicit Tree with Composite  Replace State-Altering Conditionals with State  Replace Type Code with Class  Replace Type Code with State/Strategy  Replace Type Code with Subclasses  Replace Array With Object | **Duplicated Code**  Chain Constructors  Extract Composite  [Extract Method](https://sourcemaking.com/refactoring/extract-method)  [Extract Class](https://sourcemaking.com/refactoring/extract-class)  [Form Template Method](https://sourcemaking.com/refactoring/form-template-method)  [Introduce Null Object](https://sourcemaking.com/refactoring/introduce-null-object)  Factory Method  [Pull Up Method](https://sourcemaking.com/refactoring/pull-up-method)  [Pull Up Field](https://sourcemaking.com/refactoring/pull-up-field)  [Substitute Algorithm](https://sourcemaking.com/refactoring/substitute-algorithm)  Adapter | **Long Method**  [Extract Method](https://sourcemaking.com/refactoring/extract-method)  Compose Method  [Introduce Parameter Object](https://sourcemaking.com/refactoring/introduce-parameter-object)  Move Accumulation to Collecting Parameter  Move Accumulation to Visitor  Decompose Conditional  Preserve Whole Object  Replace Conditional Dispatcher with Command  Replace Conditional Logic with Strategy  Replace Method with Method Object  Replace Temp with Query |
| **Divergent Change**  Extract Class | **Shotgun Surgery**  Move Method  Move Field  Inline Class | **Feature Envy**  Extract Method  Move Method  Move Field |
| **Long Parameter List**  Replace Parameter with Method  Introduce Parameter Object  Preserve Whole Object | **Data Clumps**  Extract Class  Preserve Whole Object  Introduce Parameter Object | **Parallel Inheritance** Hierarchies  Move Method  Move Field |
| **Middle Man**  Remove Middle Man  Inline Method  Replace Delegation with Inheritance | **Data Class**  Move Method  Encapsulate Field  Encapsulate Collection | **Message Chains**  Hide Delegate  Extract Method  Move Method |
| **Speculative Generality**  Collapse Hierarchy  Rename Method  Remove Parameter  Inline Class | **Temporary Field**  Extract Class  Introduce Null Object | **Lazy Class**  Collapse Hierarchy  Inline Class |
| **Refused Bequest**  Push Down Field  Push Down Method  Replace Inheritance with Delegation | **Alternative Classes with Different Interfaces**  Unify Interfaces with Adapter  Rename Method  Move Method | **Incomplete Library Class**  Introduce Foreign Method  Introduce Local Extension |
| **Comments**  Rename Method  Method  Introduce Extract Assertion |  |  |

## Katas

<https://github.com/emilybache/Tennis-Refactoring-Kata>

## Reading List

<http://www.amazon.co.uk/Refactoring-Improving-Design-Existing-Technology/dp/0201485672/ref=sr_1_1?ie=UTF8&qid=1428489273&sr=8-1&keywords=refactoring>

## Links

<https://www.youtube.com/watch?v=_NnElPO5BU0>

<http://sourcemaking.com/refactoring>

<https://www.youtube.com/watch?v=WpKb1XqSiUs>

# Design V

## Solid + 1 principles



## Cohesion

### Definition

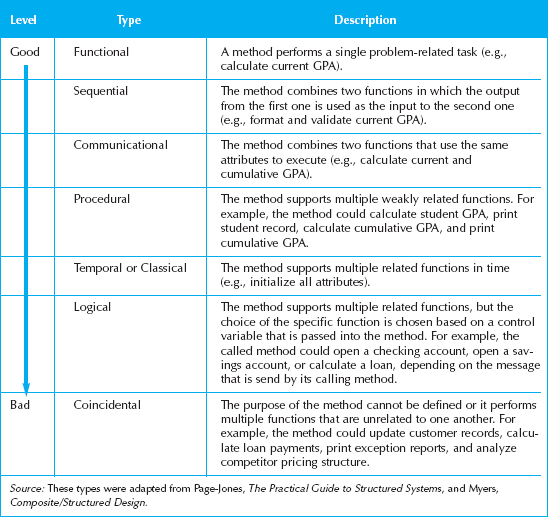
**Cambridge Dictionary**

Cohesion (noun): when the members of a group or society are united. Cohesive (adjective): united and working together effectively.

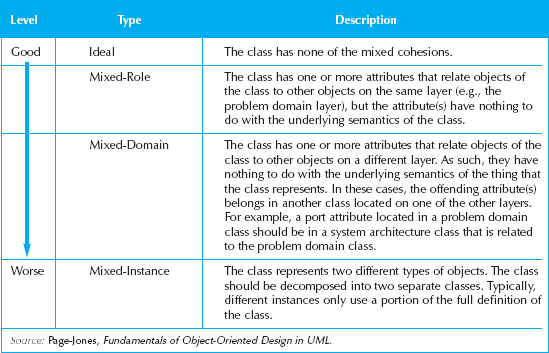
**Wikipedia**

In computer programming, cohesion is a measure of how strongly related and focused the various responsibilities of a software module are.

### Method Cohesion

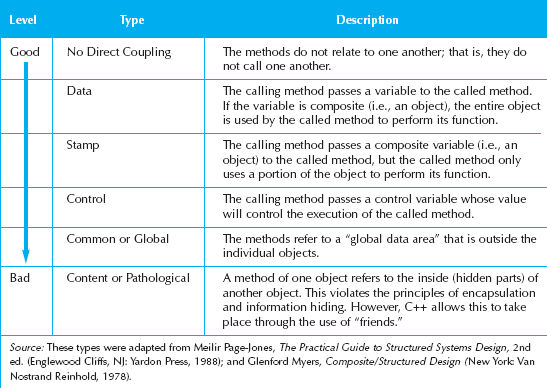


### Class Cohesion



## Coupling

* Interdependency among modules
* Interaction coupling through message passing
* **Law of Demeter**. Messages should be sent only by an object:
  + to itself
  + to objects contained in attributes of itself or a superclass
  + to an object that is passed as a parameter to the method
  + to an object that is created by the method
  + to an object that is stored in a global variable

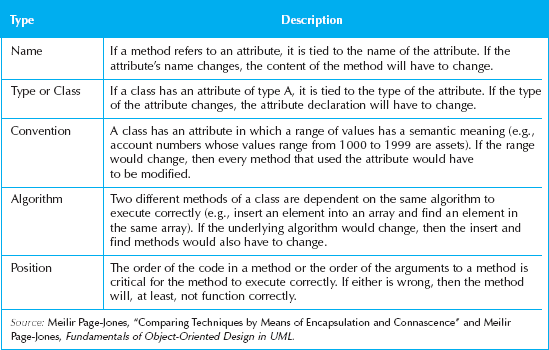


We should avoid writing code that looks like: dog.getBody().getTail().wag(); colloquially known as a “Train Wreck”. This is bad because this one line depends on the interfaces and implied structure of three different objects. This style laces structural dependencies between unrelated objects throughout a code base. The solution is described by the heuristic "Tell, Don't Ask", so we rewrite our example as: dog.expressHappiness(); and let the implementation of the dog decide what this means.

## Connascence

Creating the need to change another module as a result of changing one

* Minimize overall connascence
* Minimize across encapsulation boundaries
* Maximize within encapsulation boundary



## Single responsibility



In object-oriented programming, the single responsibility principle states that every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.

The term was introduced by Robert C. Martin. Martin described it as being based on the principle of cohesion, as described by Tom DeMarco in his book Structured Analysis and Systems Specification. Cohesion is a measure of the strength of association of the elements inside a module. A highly cohesive module is a collection of statements and data items that should be treated as a whole because they are so closely related. Any attempt to divide them up would only result in increased coupling and decreased readability.

“We refer to a sound line of reasoning, for example, as coherent. The thoughts fit, they go together, they relate to each other. This is exactly the characteristic of a class that makes it coherent: the pieces all seem to be related, they seem to belong together, and it would feel somewhat unnatural to pull them apart. Such a class exhibits cohesion.” Glenn Vanderburg.

Gather together those things that change for the same reason, and separate those things that change for different reasons. This principle is often known as the single responsibility principle, or SRP. In short, it says that a subsystem, module, class, or even a function, should not have more than one reason to change.

“This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together.” Doug McIlroy.

Extract

Inject

Refactor

## Open Closed



**“**The principle stated that a good module structure should be both open and closed:

Closed, because clients need the module's services to proceed with their own development, and once they have settled on a version of the module should not be affected by the introduction of new services they do not need.

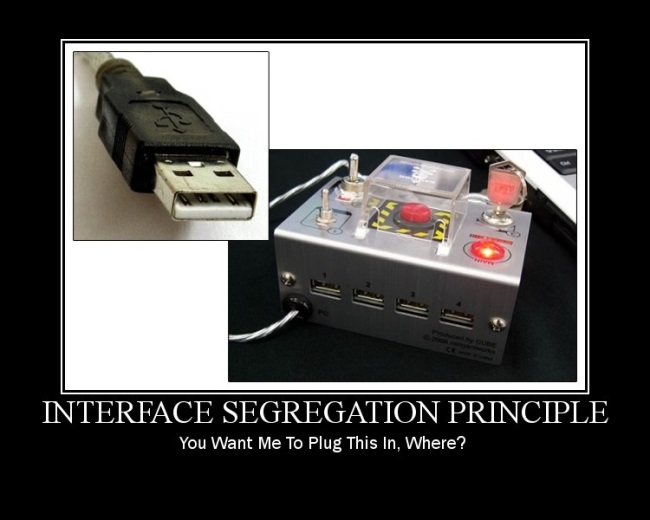
Open, because there is no guarantee that we will include right from the start every service pot**entially useful to some client.”** Bertrand Meyer**,** Object-Oriented Software Construction

“Published Interface is a term I used (first in Refactoring) to refer to a class interface that's used outside the code base that it's defined in. The distinction between published and public is actually more important than that between public and private. The reason is that with a non-published interface you can change it and update the calling code since it is all within a single code base. [...] But anything published so you can't reach the calling code needs more complicated treatment.” Martin Fowler

Identify points of **predicted variation** and create a stable interface around them.

If you predict more variation than is actually warranted, you waste effort on **over-engineering**. If you predict less variation than is required, you usually end with lots of **breaking changes**.

## Interface Segregation



The dependency should be on the interface, the whole interface, and nothing but the interface.

We refer to a sound line of reasoning, for example, as coherent. The thoughts fit, they go together, they relate to each other. This is exactly the characteristic of an interface that makes it coherent: the pieces all seem to be related, they seem to belong together, and it would feel somewhat unnatural to pull them apart. Such an interface exhibits cohesion.

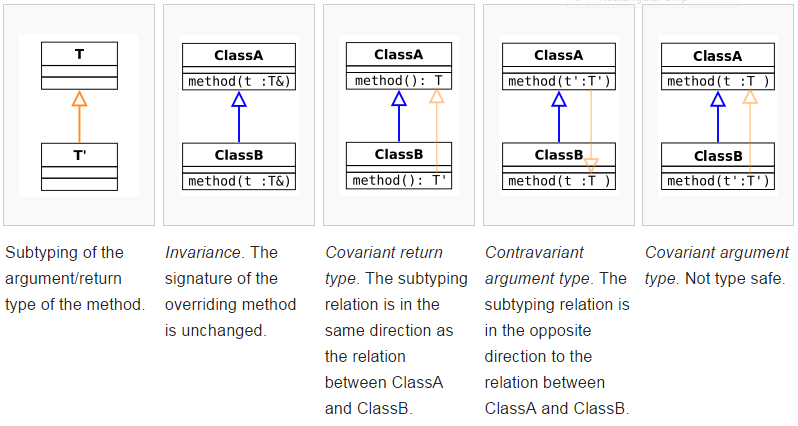
## Liskov Substitution



“A type hierarchy is composed of subtypes and super types. The intuitive idea of a subtype is one whose objects provide all the behaviour of objects of another type (the super type) plus something extra. What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that **for all programs P defined in terms of T, the behaviour of P is *unchanged* when o1 is substituted for o2**, then S is a subtype of T”. Barbara Liskov "Data Abstraction and Hierarchy".

### Conformance

Another way to state this principle is to say that subclasses should be contravariant of overridden method input types and covariant of overridden method output types; or that a derived class should be less or equally restrictive on inputs and more or equally restrictive on outputs for overridden methods.

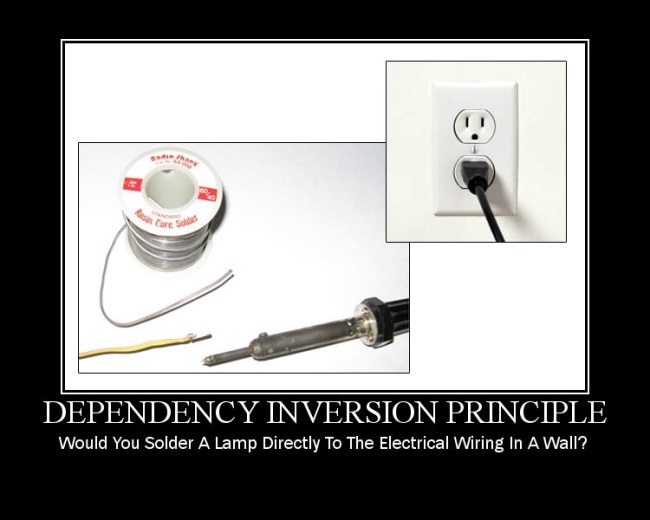


Liskov violation

Exceptions are the hidden return values in C#. The same conformance rules apply. Do not throw exceptions from derived classes that are not the same type or subtypes from exceptions thrown by base class.

Remember the refused bequest code smell?

## Dependency Inversion



In object-oriented programming, the dependency inversion principle refers to a specific form of decoupling where conventional dependency relationships established from high level, policy-setting modules to low-level, dependency modules are inverted (i.e. reversed) for the purpose of rendering high-level modules independent of the low-level module implementation details.

Likelihood of change

Less

More

Always depend on things on your left

## Solid/ Cohesion / Coupling

* Single Responsibility Principle (Cohesion by Consistency/Usage)
* Open Closed Principle (Cohesion by Change)
* Liskov Substitution Principle (Conformance)
* Interface Segregation Principle (Cohesion by Consistency/Usage)
* Dependency injection Principle (Coupling, connascence of name/type if using service locator anti pattern)

## The 4 C’s in design

* Cohesion
* Coupling
* Connascence
* Conformance

## Balanced Abstraction Principle

The Balanced Abstraction Principle defines that all code constructs grouped by a higher-level construct should be on the same level of abstraction. That means:

* All instructions inside a method should be at the same level of abstraction
* All public methods inside a class should be at the same level of abstraction
* All classes inside a package/namespace
* All sibling packages/namespace inside a parent package/namespace
* All modules, sub-systems, etc.

The principle also applies to tests—all tests for a single unit (method, class, module, system) should be at the same level of abstraction.

BAP and SRP

Code that complies with the Single Responsibility Principle has a higher chance to also be compliant to the Balanced Abstraction Principle. However, this is not always the case and the opposite is not always true.

**Solid/non Solid examples**

## Links

<http://codurance.com/2015/01/27/balanced-abstraction-principle/>

<http://martinfowler.com/bliki/PublishedInterface.html>

<http://yowconference.com.au/slides/yow2013/Henney-SOLIDDeconstruction.pdf>

##### <http://codurance.com/2016/03/03/cohesion-cornerstone-software-design/>

##### <http://connascence.io/pages/about.html>

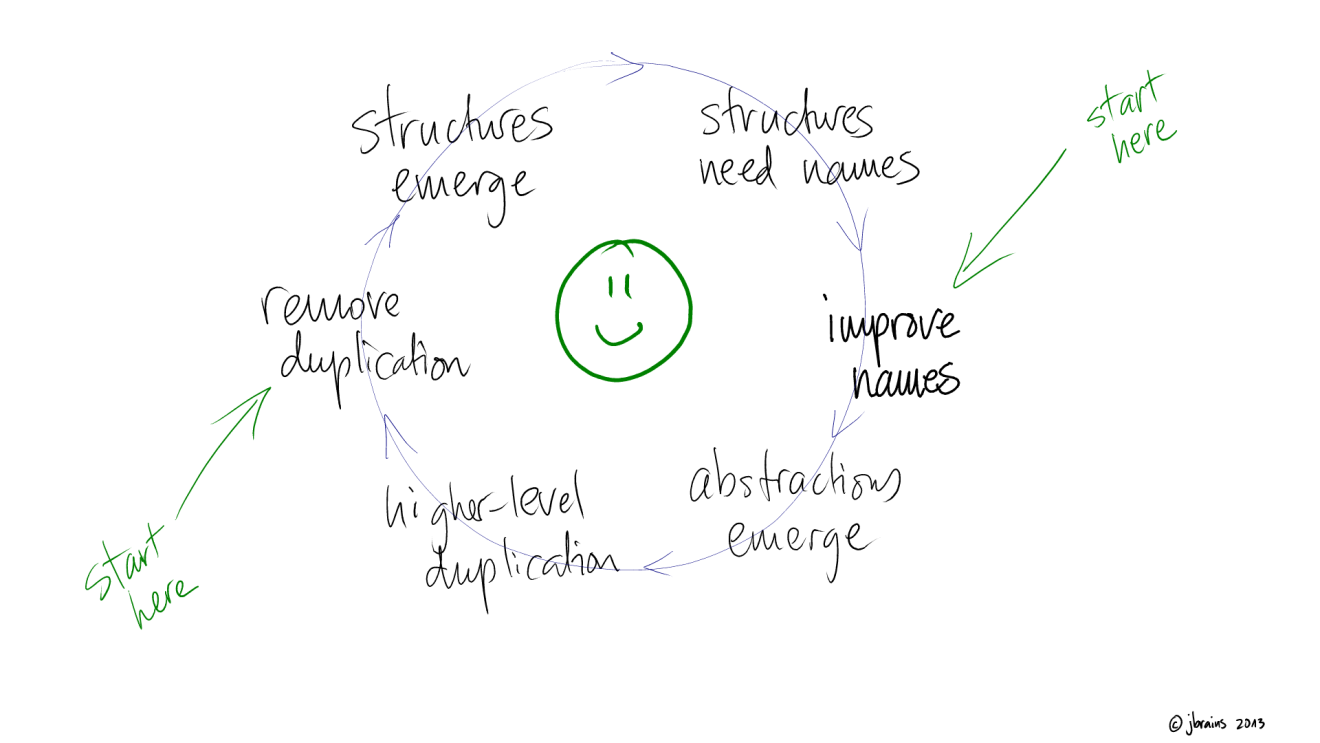
## Reading list

<http://www.amazon.co.uk/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882/ref=sr_1_1?ie=UTF8&qid=1428489389&sr=8-1&keywords=clean+code>

# Design VI

## The four elements of simple design

1. Passes its tests
2. Minimizes duplication
3. Maximizes clarity
4. Has fewer elements



### What does it mean?

“I claim that developing strong skills of detecting duplication, removing duplication, identifying naming problems, and fixing naming problems equates to learning everything ever written about object-oriented design.” J. B. Rainsberger.

Tests help limit the amount of code written. Removing duplication and improving names helps reduce the liability (cost) of the code. Together, they help reduce both the total cost and the volatility of the cost of the features.

* Running all the tests makes refactoring safe. But thinking about having *all* the tests gives us their other benefit: they help make the code clear; they help to communicate the design.
* Removing duplication tends to create new, and smaller, classes and methods. We find a little patch of code that appears again and again. It has some meaning: we extract it, giving life to whatever idea that code patch implements
* Expressing ideas gives meaningful names to the classes and methods we have and to those that are created by the duplication rule. As well, we are often moved to create new classes and methods just for improved expression. For example, it is common to replace a switch statement with a few classes with a polymorphic method. When we see the essential idea that underlies the switch, the classes, and the name of the method, tend to pop right out.
* The fourth and final rule, minimizing the number of classes and methods, helps us make sure that we don't proliferate unnecessary entities.

Applying these rules, we wind up with compact, modular code, expressing all the important ideas of the system. We wind up with “good design”.

### Alternative definitions

|  |  |
| --- | --- |
| **Definition I** | **Definition II** |
| Passes its tests  Minimizes duplication  Maximizes clarity  Has fewer elements | Pass all tests  Duplicates no behaviour or configuration  Clear, expressive and consistent  Minimal methods, classes and modules |
| **Definition III** | **Definition IV** |
| Runs all the tests  Maximizes Cohesion  Minimizes Coupling  Says everything once and only once | Passes all the tests  Expresses every idea that we need to express  Says everything once and only once  Has fewer elements |

## Katas

### Mars Rover

The problem below requires some kind of input. You are free to implement any mechanism for feeding input into your solution (for example, using hard coded data within a unit test). You should provide sufficient evidence that your solution is complete by, as a minimum, indicating that it works correctly against the supplied test data.

The code you write should be of production quality, and most importantly, it should be code you are proud of. MARS ROVERS

A squad of robotic rovers are to be landed by NASA on a plateau on Mars.

This plateau, which is curiously rectangular, must be navigated by the rovers so that their on board cameras can get a complete view of the surrounding terrain to send back to Earth.

A rover's position is represented by a combination of an x and y co-ordinates and a letter representing one of the four cardinal compass points. The plateau is divided up into a grid to simplify navigation. An example position might be 0, 0, N, which means the rover is in the bottom left corner and facing North.

In order to control a rover, NASA sends a simple string of letters. The possible letters are 'L', 'R' and 'M'. 'L' and 'R' makes the rover spin 90 degrees left or right respectively, without moving from its current spot.

'M' means move forward one grid point, and 7maintains the same heading.

Assume that the square directly North from (x, y) is (x, y+1).

Input:

The first line of input is the upper-right coordinates of the plateau, the lower-left coordinates are assumed to be 0,0.

The rest of the input is information pertaining to the rovers that have been deployed. Each rover has two lines of input. The first line gives the rover's position, and the second line is a series of instructions telling the rover how to explore the plateau.

The position is made up of two integers and a letter separated by spaces, corresponding to the x and y co-ordinates and the rover's orientation.

Output:

The output for each rover should be its final co-ordinates and heading.

Test Input:

5 5

1 2 N

LMLMLMLMM

Expected Output:

1 3 N

#### GIT

## Links

<http://www.jbrains.ca/permalink/the-four-elements-of-simple-design>

<http://blog.thecodewhisperer.com/2013/12/07/putting-an-age-old-battle-to-rest/>

<http://martinfowler.com/bliki/BeckDesignRules.html>

<http://c2.com/cgi/wiki?XpSimplicityRules>

## Reading List

<https://leanpub.com/4rulesofsimpledesign/>

# Design VII

## Design Patterns

We refactor into design patterns.

### Purpose

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | **Creational** | **Structural** | **Behavioural** |
| **Scope** | Class | Factory Method | Adapter | ~~Interpreter~~  Template method |
| Object | Abstract Factory  Builder  Prototype  Singleton | Adapter  ~~Bridge~~  ~~Composite~~  Decorator  Façade  Proxy | ~~Chain of responsibility~~  Command  Iterator  Mediator  ~~Memento~~  ~~Flyweight~~  Observer  State  Strategy  ~~Visitor~~ |

### Catalogue

|  |  |  |  |
| --- | --- | --- | --- |
| **Design pattern** | **Problem** | **Design pattern** | **Problem** |
| **Strategy**  Defines a family of algorithms, encapsulates each one, and make them interchangeable. | Do we have a varying rule? | **Observer**  Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. | Do various entities need to know about events that have occurred? |
| **Template**  Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. | Do we have an algorithm that needs to redefine certain behaviour in sub classes? | **Chain of Responsibility**  Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. | Do we have different objects that can do the job but we don’t want the client object know which is actually going to do it? |
| **Visitor**  Represent an operation to be performed on the elements of an object structure. | Do we have new tasks that we will need to apply to our existing classes? | **Iterator**  Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. | Do we want to separate the collection from the client that is using it so we don’t have to worry about having the right collection implementation? |
| **State**  Allow an object to alter its behaviour when it’s internal state changes. The object will appear to change its class. | Do we have a system with lots of states where keeping track of code for the different states is difficult? | **Mediator**  Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently. | Do we have a lot of coupling in who must talk to who? |
| **Proxy**  Provides a surrogate or placeholder for another object to control access to it. | Do we need to optionally add some new functionality to something that already exists? | **Decorator**  Attaches additional responsibilities to an object dynamically. | Do we have multiple additional functions we may need to apply, but which and how many we add varies? Do we need to capture an order dependency for multiple additional functions? |
| **Adapter**  Convert the interface of a class into another interface clients expect. | Do we have the right stuff but the wrong interface? (Used to fit classes into patterns as well) | **Abstract Factory**  Provide an interface for creating families of related or dependent objects without specifying their concrete classes. | Do we need to create families (or sets) of objects? |
| **Composite**  Compose objects into tree structures to represent part-whole hierarchies. | Do we have units and groups and want to treat them the same way? | **Builder**  Separate the construction of a complex object from its representation so that the same construction processes can create different representations. | Do we need to create our objects with several steps? |
| **Façade**  Provide a unified interface to a set of interfaces in a system. | Do we want to simplify, beautify, existing system or sub-system? | **Factory Method**  Define an interface for creating an object, but let the subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. | Do we need to have derived classes figure out what to instantiate? |

## Links

<http://www.dofactory.com/net/design-patterns>

<http://www.amazon.co.uk/Head-First-Design-Patterns-Freeman/dp/0596007124/ref=sr_1_1?ie=UTF8&qid=1428498256&sr=8-1&keywords=head+first+design+patterns>

http://www.amazon.co.uk/s/ref=nb\_sb\_noss\_2?url=search-alias%3Daps&field-keywords=Refactoring%20to%20Patterns

http://blog.markturansky.com/wp-content/uploads/2008/01/designpatterns1\_sm.jpg

<http://blog.markturansky.com/wp-content/uploads/2008/01/designpatterns2_sm.jpg>

<http://campus.murraystate.edu/academic/faculty/wlyle/430/rc008-designpatterns_online.pdf>

# Design VIII

## CRC Cards or Responsibility-based modelling

Use Class, Responsibilities, and Collaboration (CRC) Cards to design the system as a team. The biggest value of CRC cards is to allow people to break away from the procedural mode of thought and more fully appreciate object technology. CRC Cards allow entire project teams to contribute to the design. The more people who can help design the system the greater the number of good ideas incorporated.

Individual CRC Cards are used to represent objects. The class of the object can be written at the top of the card, responsibilities listed down the left side, collaborating classes are listed to the right of each responsibility. We say "can be written" because once a CRC session is in full swing participants usually only need a few cards with the class name and virtually no cards written out in full. A short example is shown as part of the coffee maker problem.

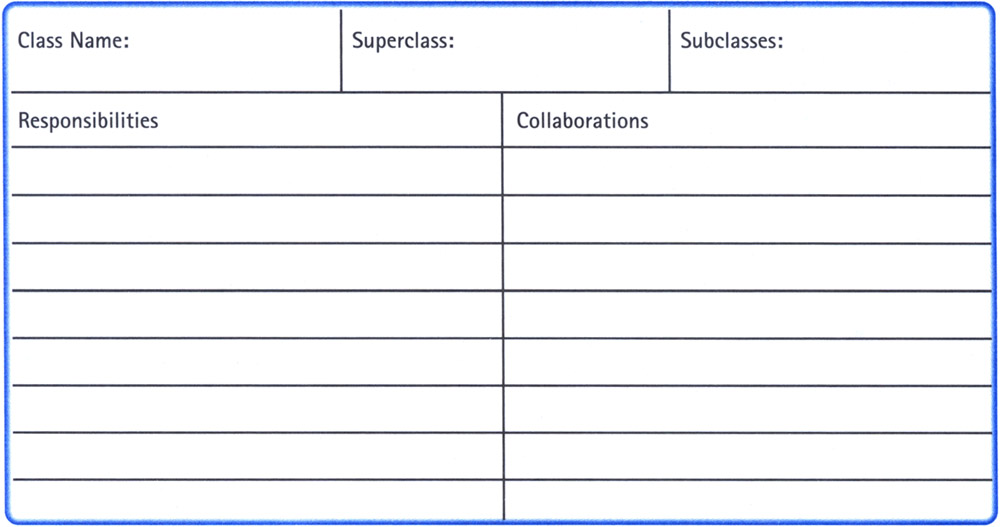
A CRC session proceeds with someone simulating the system by talking about which objects send messages to other objects. By stepping through the process weaknesses and problems are easily uncovered. Design alternatives can be explored quickly by simulating the design being proposed.

If you find too many people speaking and moving cards at once then simply limit the number of people standing and moving cards to two. When one person sits down another may stand up. This works for sessions that get out of hand, which often happens as teams become rowdy when a tough problem is finally solved.

One of the biggest criticisms of CRC Cards is the lack of written design. This is usually not needed as CRC Cards make the design seem obvious. Should a more permanent record be required, one card for each class can be written out in full and retained as documentation. A design, once envisioned as if it were already built and running, stays with a person for some time.

CRC Cards explicitly represent multiple objects simultaneously

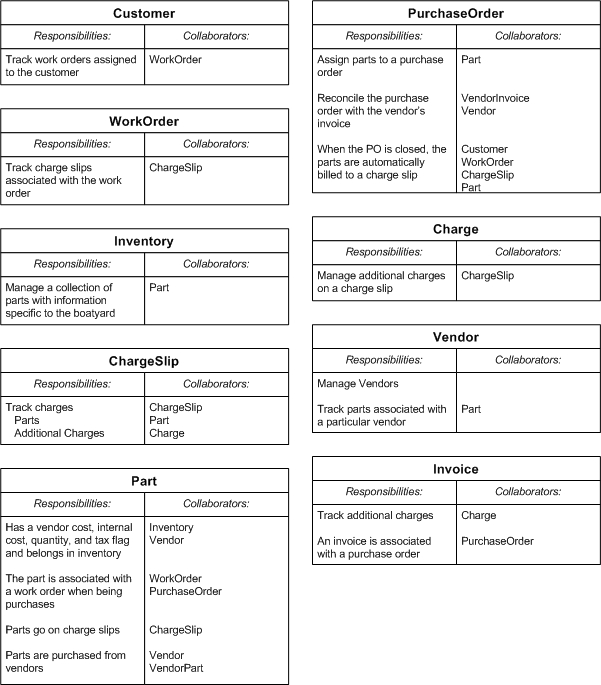
1. The Name of the class it refers to
2. The Responsibilities of the class. These should be high level, not at the level of individual methods
3. The Collaborators that help discharge a responsibility



## CRC Steps

* Step 1. Decide to use CRC cards and choose an coherent set of use cases
  + Decide to work with CRC cards as opposed to or in conjunction with object interaction diagrams and object type diagrams.
  + Select a set of use cases which look as though they will touch a related set of object types. These provide the scenarios for the group to walk through.
* Step 2. Put a card on the table
  + Put a first card onto the table to represent the external actor who triggers the use case.
  + Put a second card on the table. This is the card to whom the first card will send its initial message.
  + Alternatively, put cards onto the table for all the known, relevant object types, and label the cards with their main responsibilities.
* Step 3. Walk through the scenario, naming cards and responsibilities
  + Walk through the handling of a scenario of the use case, pointing to or picking up the cards, naming their responsibilities and how they handle and delegate each request.
  + In a brainstorming session, add new cards as new functions are needed, or reallocate the responsibilities of the cards already on the table. It is not always necessary to name both the object type and the responsibility at the moment the card is put onto the table, as long as they are both written before the end.
* Step 4. Vary the situations, to stress test the cards
  + At any time during the walkthrough, you may vary the assumptions on the use case, to see if that causes a shift in the handling. With a good design, the handling is the same, but with the addition of a future object, or the change to at most one card.
  + If it is decided a new object is needed to create a more stable design, add a new card, with the needed responsibility put onto it.
  + Not all the cards on the table need be used; some may drift out to the sides if they are not used much. The cards that are needed at the end are those that get put into the design.
* Step 5. Add cards, push cards to the side, to let the design evolve
  + CRC cards permit several design alternatives to sit on the table at the same time. An unpopular initial design may turn out to be a popular later design, or perhaps the final design is a small alteration of an initially rejected design.
  + Do not throw cards away, but push them to the side, in case it turns out later they are useful.
* Step 6. Write down the key responsibility decisions and interactions
  + Often, the design turns around a few key decisions about the allocation of responsibilities. Write these down.
  + It is not useful to draw interaction diagrams for all the scenarios considered, but it is very useful to draw a representative few.

## Example



## Kata

In this session we will do an OO design for a known kata, using paper and pencil only.

* The objective is to define a design for the first story of a new product we will be launching soon.
* We will be creating a design using CRC cards, following OO design principles such as SOLID, DRY, YAGNI, Tell Don’t Ask, etc.
* The product is a personal savings ATM machine similar to this:
  + 
* For the first release
  + We will disregard the card reader and user authentication.
  + We will implement the Deposit, Withdraw and Print Statement operations.

We have the following gherkin definition for our first user story:

Given a client makes a deposit of 10 on 10-01-2012

And a deposit of 20 on 13-01-2012

And a withdrawal of 5 on 14-01-2012

When they print their bank statement

Then they would see

date || credit || debit || balance

14/01/2012 || || 5.00 || 25.00

13/01/2012 || 20.00 || || 30.00

10/01/2012 || 10.00 || || 10.00

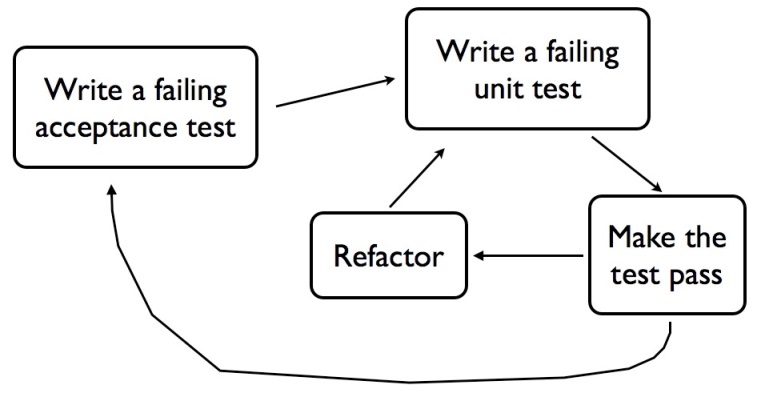
## Links

<http://alistair.cockburn.us/Responsibility-based+modeling>

<http://alistair.cockburn.us/Using+CRC+cards>

**Outside-in TDD (AKA London School of TDD, AKA ATDD, AKA Mockist TDD)**

Outside-In TDD, also known as London School, ATDD or mockist, is a TDD style developed and adopted by some of the first XP practitioners in London. It later inspired the creation of BDD.



**Main characteristics**

* Different from the classicist, Outside-In TDD prescribes a direction in which we start test-driving our code: from outside (first class to receive an external request) to the inside (classes that will contain single pieces of behaviour that satisfy the feature being implemented).
* We normally start with an acceptance test which verifies if the feature as a whole works. The acceptance test also serves as a guide for the implementation.
* With a failing acceptance test informing why the feature is not yet complete (no data returned, no message sent to a queue, no data stored in a database, etc.), we start writing unit tests. The first class to be tested is the class handling an external request (a controller, queue listener, event handler, the entry point for a component, etc.)
* As we already know that we won’t build the entire application in a single class, we make some assumptions of which type of collaborators the class under test will need. We then write tests that verify the collaboration between the class under test and its collaborators.
* Collaborators are identified according to all the things the class under test needs to do when its public method is invoked. Collaborators names and methods should come from the domain language (nouns and verbs).
* Once a class is tested, we pick the first collaborator (which was created with no implementation) and test-drive its behaviour, following the same approach we used for the previous class. This is why we call outside-in: we start from classes that are closer to the input of the system (outside) and move towards the inside of our application as more collaborators are identified.
* Design starts in the *red* phase, while writing the tests.
* Tests are about collaboration and behaviour, not state.
* Design is refined during the *refactoring* phase.
* Each collaborator and its public methods are always created to *serve* an existing client class, making the code read very well.
* Refactoring phases are much smaller, when compared to the classicist approach.
* Promotes a better encapsulation since no state is exposed for test purposes only,
* More aligned to the “*tell, don’t ask”* approach.
* More aligned to the original ideas of Object Oriented Programming: tests are about objects sending messages to other objects instead of checking their state.
* Suitable for business applications, where names and verbs can be extracted from user stories and acceptance criteria.

**Problems**

* Much harder for novices to adopt since a higher level of design skill is necessary.
* Developers don’t get feedback from code in order to create collaborators. They need to *visualise* collaborators while writing the test.
* May lead to over-engineering due to premature type (collaborators) creation.
* Not suitable for exploratory work or behaviour that is not specified in a user story (data transformation, algorithms, etc.).
* Bad design skills may lead to an explosion of mocks.
* Behavioural tests are harder to write than state tests.
* Knowledge of Domain Driven Design and other design techniques, including 4 Rules of Simple Design, are required while writing tests.

**Which TDD style should we use?**

Both, all, they are just tools and as such, they should be used according to your needs. Experienced TDD practitioners jump from one style to another without ever worrying which style they are using.

**Principles**

### Allow Queries; Expect Commands

* Use Mocks for Commands
  + Commands are all about side effects, and Mocks are all about [Behaviour Verification](http://xunitpatterns.com/Behavior%20Verification.html): that is, that side effects occurred
* Use Stubs for Queries
  + Stubs mainly exist to 'make happy noises', and one of the ways they have to do that, is to return data from dependencies, when return data is required.

**Tell don’t ask**

“Tell-Don't-Ask is a principle that helps people remember that object-orientation is about bundling data with the functions that operate on that data. It reminds us that rather than asking an object for data and acting on that data, we should instead tell an object what to do. This encourages moving behaviour into an object to go with the data.” Martin Fowler.

“Different design styles have different techniques that are most applicable for test-driving code written in those styles, and there are different tools that help you with those techniques… That’s what we… designed JMock to do … Tell, Don’t Ask object-oriented design“ Nat Pryce, in an email to a discussion forum.

“Tell, Don’t Ask” Object Oriented Design is about having Cohesive objects that hide their internal workings. If your objects obey the Law of Demeter, that’s a good start, it means they hide their inner workings and don’t talk to objects far away on the object graph. It reduces Coupling in your system, which should make for better maintainability.

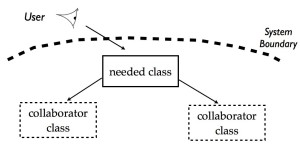
In their book “Growing Object Oriented Software, Guided by Tests”, Freeman & Pryce actually define “Tell, Don’t Ask” as the same as following the Law of Demeter (p17). Then they go on with several chapters about their design style, expanding far beyond simply “following the Law of Demeter”. It’s well worth a read, here’s a sample:

“… we focus our design effort on how the objects collaborate … obviously, we want to achieve a well-designed class structure, but we think the communication patterns between objects are more important.” Freeman & Pryce, GOOS, p58

## Designing Outside-In

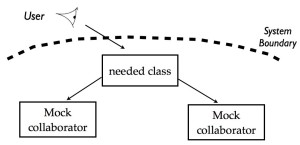
If you’re doing double loop TDD, you’ll begin with a Guiding Test that expresses something about how a user wants to interact with your system. That test helps you identify the top level function or class that is the entry point to the desired functionality that will be called first. Often it’s a widget in a GUI, a link on a webpage, or a command line flag.

With London School TDD, you’ll often start your inner loop TDD by designing the class or method that gets called by that widget in the GUI, that link on the webpage, or that command line flag. You should quickly discover that this new piece of code can’t implement the whole function by itself, but will need collaborating classes to get stuff done.



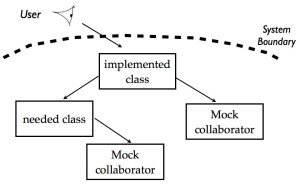
The user looks at the system, and wants some functionality. This implies a new class is needed at the boundary of the system. This class in turn needs collaborating classes that don’t yet exist.

The collaborating classes don’t exist yet, or at least don’t provide all the functionality you need. Instead of going away and developing these collaborating classes straight away, you can just replace them with mocks in your test. It’s very cheap to change mocks and experiment until you get the the interface and the protocol just the way you want it. While you’re designing a test case, you’re also designing the production code.



You replace collaborating objects with mocks so you can design the interface and protocol between them.

When you’re happy with your design, and your test passes, you can move down the stack and start working on developing the implementation of one of the collaborating classes. Of course, if this class in turn has other collaborators, you can replace them with mocks and design these interactions too. This approach continues all the way through the system, moving through architectural layers and levels of abstraction.



You’ve designed the class at the boundary of the system, and now you design one of the collaborating classes, replacing its collaborators with mocks.

This way of working lets you break a problem down into manageable pieces, and get each part specified and tested before you move onto the next part. You start with a focus on what the user needs, and build the system from the “outside-in”, following the user interaction through all the parts of the system until the guiding test passes. The Guiding Test will not usually replace parts of the system with mocks, so when it passes you should be confident you’ve remembered to actually implement all the needed collaborating classes.

## *Key Ideas*

* In general, we want low coupling and high cohesion.
* Object-oriented design can be thought of as the network of communications among the objects in your software system.
* Objects are mutable; values are immutable.
* Interfaces help define an object's roles.
  + Use interfaces to name roles played by objects. Keep interfaces narrow in scope.
* "Tell, Don't Ask" or "Law of Demeter"
* Mock objects are used to test interactions between objects.
* Begin with a "walking skeleton".
  + Minimun project tha is:
    - Automatically buildable
    - Automatically deployable
    - Automaticaly testable
    - No functionality required
    - Initial architecture
  + Why?
    - Infrastructure to build, deploy and test expensive
    - Need it so why not start with it
    - You may not have time later on
    - Early visibility
    - Infrastructure
    - Architecture
* Makes Outside-In testing easier
* Start each new feature with an acceptance test to determine how the new feature will function.
* Separate acceptance tests for completed features to catch bugs vs acceptance tests for new features in progress.
* Write unit tests for object behaviour rather than the object's methods.
* Unit tests check the internal quality of the code; acceptance tests check the external quality.
* Something that is difficult to test is probably badly designed.
* Single Responsibility Principle:
  + Our heuristic is that we should be able to describe what an object does without using any conjunctions ("and," "or").
* Interacting with the composite object should be simpler than interacting with the components that compose it.
* "Mock an object's peers [...] but not its internals."
* Techniques for introducing new objects:
  + "Breaking out": when code for an object becomes too complex, separate it into smaller units
  + "Budding off": placeholder for a new object, to be filled in with more implementation details later
  + "Bundling up": creating a new object for a group of objects that are always used together
* When to break out:
  + Break up an object if it becomes too large to test easily, or if its test failures become difficult to interpret. Then unit-test the new parts separately.
* When to bud off:
  + When writing a test, we ask ourselves, "If this worked, who would know?" If the right answer to that question is not in the target object, it's probably time to introduce a new collaborator.
* When to bundle up:
  + When the test for an object becomes too complicated to set up [...] consider bundling up some of the collaborating objects.
* Goal is to move to "higher-order" programming: "composing programs from smaller programs".
* Don't use mocks for third-party code, since it is usually not changeable. Use an adapter layer to implement interactions with third-party code.

## Walking skeleton

### What?

* Automatically buildable
* Automatically deployable
* Automaticaly testable
* No functionality required
* Initial architecture

### Why?

* Infrastructure to build, deploy and test expensive
  + Need it so why not start with it
  + You may not have time later on
* Early visibility
  + Infrastructure
  + Architecture
* Makes Outside-In testing easier

**Katas**

**Bank**

**Requirements**

Deposit and Withdrawal

Account statement (date, amount, balance)

Statement printing

Statement filters (just deposits, withdrawal, date)

Starting Point

Given a client makes a deposit of 1000 on 10-01-2012

And a deposit of 2000 on 13-01-2012

And a withdrawal of 500 on 14-01-2012

When she prints her bank statement

Then she would see

date || credit || debit || balance

14/01/2012 || || 500.00 || 2500.00

13/01/2012 || 2000.00 || || 3000.00

10/01/2012 || 1000.00 || || 1000.00

The Rules

• One level of indentation per method

• Don’t use the ELSE keyword

• Wrap all primitives and Strings

• First class collections

• One dot per line

• Don’t abbreviate

• Keep all entities small (50 lines)

• No classes with more than two instance variables

• No getters/setters/properties

**Links**

<http://coding-is-like-cooking.info/2013/04/outside-in-development-with-double-loop-tdd/>

<http://coding-is-like-cooking.info/2013/04/the-london-school-of-test-driven-development/>

<http://martinfowler.com/bliki/TellDontAsk.html>

<http://c2.com/cgi/wiki?TellDontAsk>

<http://c2.com/cgi/wiki?LawOfDemeter>

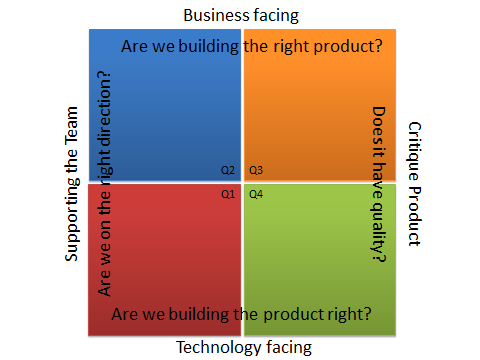
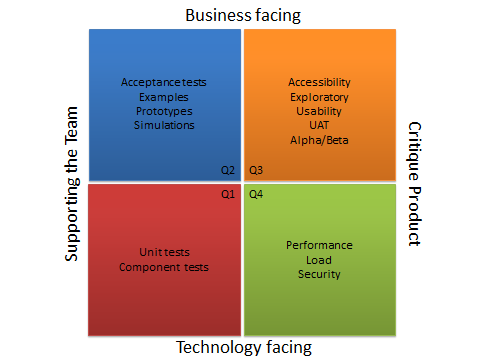
http://blog.ploeh.dk/2013/10/23/mocks-for-commands-stubs-for-queries/

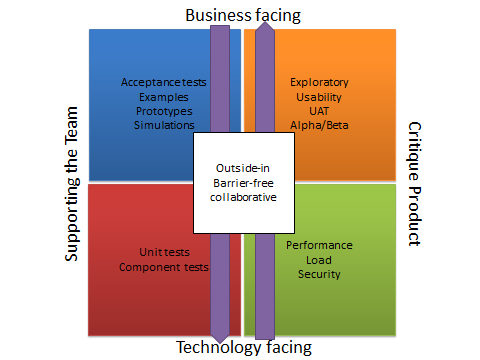
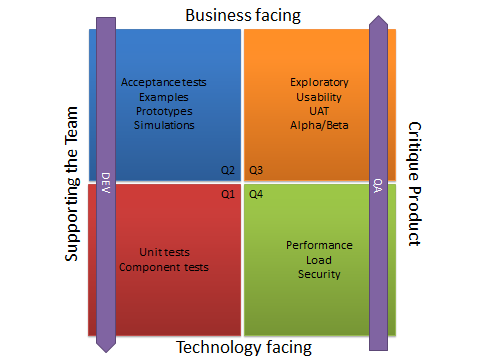
**Reading list**

<http://www.amazon.co.uk/Growing-Object-Oriented-Software-Guided-Signature/dp/0321503627/ref=sr_1_1?ie=UTF8&qid=1428489511&sr=8-1&keywords=guided+by+tests>

# Agile testing

# 



# Design IX

## Domain Driven Design

### Definitions

#### Domain

A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.

### Model

A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.

### Ubiquitous language

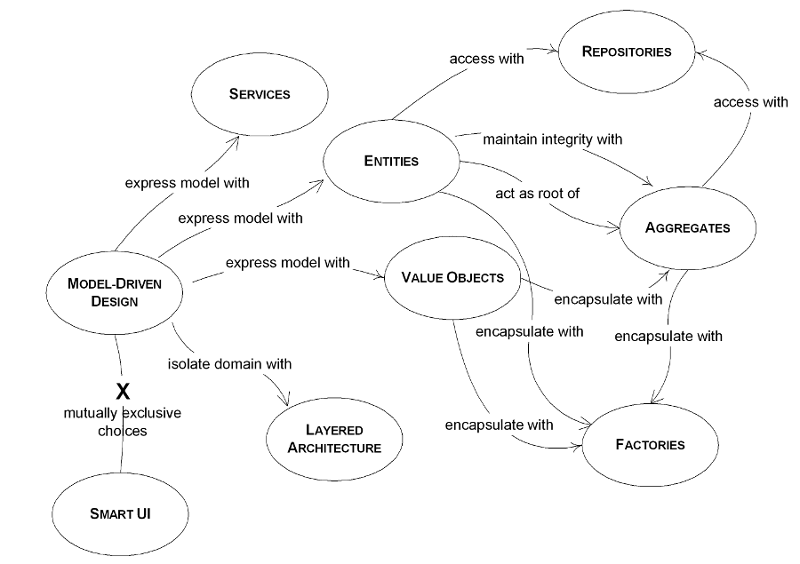
A language structured around the domain model and used by all team members within a bounded context to connect all the activities of the team with the software.

#### Context

The setting in which a word or statement appears that determines its meaning. Statements about a model can only be understood in a context.

#### Bounded context

A description of a boundary (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable.



## Putting the Model to Work

Domain-­‐Driven Design is an approach to the development of complex software in which we:

1. Focus on the core domain.
2. Explore models in a creative collaboration of domain practitioners and software practitioners.
3. Speak a ubiquitous language within an explicitly bounded context.

This three-point summary of DDD depends on the definition of the terms, which are defined in this booklet.

Many projects do modeling work without getting much real benefit in the end. The patterns of DDD distill successful practices from projects where dramatic benefits have come from modeling. Taken together, they lay out a quite different approach to modeling and software development that runs from fine details to high-level vision. Rigorous modeling conventions must be balanced with free exploration of models in collaboration with non-technical people. Tactics and strategy must be combined to succeed, and DDD addresses both tactical and strategic design.

### Bounded Context

Multiple models are in play on any large project. They emerge for many reasons. Two subsystems commonly serve very different user communities, with different jobs, where different models may be useful. Teams working independently may solve the same problem in different ways through lack of communication. The tool set may also be different, meaning that program code cannot be shared.

Multiple models are inevitable, yet when code based on distinct models is combined, software becomes buggy, unreliable, and difficult to understand. Communication among team members becomes confused. It is often unclear in what context a model should not be applied.

Model expressions, like any other phrase, only have meaning in context.

Therefore:

Explicitly define the context within which a model applies. Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas. Apply Continuous Integration to keep model concepts and terms strictly consistent within these bounds, but don’t be distracted or confused by issues outside. Standardize a single development process within the context, which need not be used elsewhere.

### Ubiquitous Language

For first you write a sentence,

And then you chop it small;

Then mix the bits, and sort them out

Just as they chance to fall:

The order of the phrases makes

No difference at all.

—Lewis Carroll, “Poeta Fit, Non Nascitur”

To create a supple, knowledge-rich design calls for a versatile, shared team language, and a lively experimentation with language that seldom happens on software projects.

Within a single bounded context, language can be fractured in ways that undermine efforts to apply sophisticated modeling. If the model is only used to draw UML diagrams for the technical members of the team, then it is not contributing to the creative collaboration at the heart of DDD.

Domain experts use their jargon while technical team members have their own language tuned for discussing the domain in terms of design. The terminology of day-to-day discussions is disconnected from the terminology embedded in the code (ultimately the most important product of a software project). And even the same person uses different language in speech and in writing, so that the most incisive expressions of the domain often emerge in a transient form that is never captured in the code or even in writing.

Translation blunts communication and makes knowledge crunching anemic.

Yet none of these dialects can be a common language because none serves all needs.

Domain experts should object to terms or structures that are awkward or inadequate to convey domain understanding; developers should watch for ambiguity or inconsistency that will trip up design.

Play with the model as you talk about the system. Describe scenarios out loud using the elements and interactions of the model, combining concepts in ways allowed by the model. Find easier ways to say what you need to say, and then take those new ideas back down to the diagrams and code.

With a ubiquitous language, the model is not just a design artifact. It becomes integral to everything the developers and domain experts do together.

Therefore:

Use the model as the backbone of a language. Commit the team to exercising that language relentlessly in all communication within the team and in the code. Within a bounded context, use the same language in diagrams, writing, and especially speech.

Recognize that a change in the language is a change to the model.

Iron out difficulties by experimenting with alternative expressions, which reflect alternative models. Then refactor the code, renaming classes, methods, and modules to conform to the new model. Resolve confusion over terms in conversation, in just the way we come to agree on the meaning of ordinary words.

### Model-Driven Design

Tightly relating the code to an underlying model gives the code meaning and makes the model relevant.

If the design, or some central part of it, does not map to the domain model, that model is of little value, and the correctness of the software is suspect. At the same time, complex mappings between models and design functions are difficult to understand and, in practice, impossible to maintain as the design changes. A deadly divide opens between analysis and design so that insight gained in each of those activities does not feed into the other.

Draw from the model the terminology used in the design and the basic assignment of responsibilities. The code becomes an expression of the model, so a change to the code may be a change to the model. Its effect must ripple through the rest of the project’s activities accordingly.

To tie the implementation slavishly to a model usually requires software development tools and languages that support a modeling paradigm, such as object-oriented programming.

Therefore:

Design a portion of the software system to reflect the domain model in a very literal way, so that mapping is obvious. Revisit the model and modify it to be implemented more naturally in software, even as you seek to make it reflect deeper insight into the domain. Demand a single model that serves both purposes well, in addition to supporting a fluent ubiquitous language.

## Building Blocks of a Model-Driven Design

These patterns cast widely held best practices of object-oriented design in the light of domain-driven design. They guide decisions to clarify the model and to keep the model and implementation aligned with each other, each reinforcing the other’s effectiveness. Careful crafting the details of individual model elements gives developers a steady platform from which to explore models and to keep them in close correspondence with the implementation.

### Entities

Many objects represent a thread of continuity and identity, going through a lifecycle, though their attributes may change.

Some objects are not defined primarily by their attributes. They represent a thread of identity that runs through time and often across distinct representations. Sometimes such an object must be matched with another object even though attributes differ. An object must be distinguished from other objects even though they might have the same attributes. Mistaken identity can lead to data corruption.

Therefore:

When an object is distinguished by its identity, rather than its attributes, make this primary to its definition in the model. Keep the class definition simple and focused on life cycle continuity and identity.

Define a means of distinguishing each object regardless of its form or history. Be alert to requirements that call for matching objects by attributes. Define an operation that is guaranteed to produce a unique result for each object, possibly by attaching a symbol that is guaranteed unique. This means of identification may come from the outside, or it may be an arbitrary identifier created by and for the system, but it must correspond to the identity distinctions in the model.

The model must define what it means to be the same thing.

(aka Reference Objects)

### Value Objects

Some objects describe or compute some characteristic of a thing.

Many objects have no conceptual identity.

Tracking the identity of entities is essential, but attaching identity to other objects can hurt system performance, add analytical work, and muddle the model by making all objects look the same. Software design is a constant battle with complexity. We must make distinctions so that special handling is applied only where necessary.

However, if we think of this category of object as just the absence of identity, we haven’t added much to our toolbox or vocabulary. In fact, these objects have characteristics of their own, and their own significance to the model. These are the objects that describe things.

Therefore:

When you care only about the attributes and logic of an element of the model, classify it as a value object. Make it express the meaning of the attributes it conveys and give it related functionality. Treat the value object as immutable. Make all operations Side-effect-free Functions that don’t depend on any mutable state. Don’t give a value object any identity and avoid the design complexities necessary to maintain entities.

### Domain Events

Something happened that domain experts care about.

An entity is responsible for tracking its state and the rules regulating its lifecycle. But if you need to know the actual causes of the state changes, this is typically not explicit, and it may be difficult to explain how the system got the way it is. Audit trails can allow tracing, but are not usually suited to being used for the logic of the program itself. Change histories of entities can allow access to previous states, but ignores the meaning of those changes, so that any manipulation of the information is procedural, and often pushed out of the domain layer.

A distinct, though related set of issues arises in distributed systems. The state of a distributed system cannot be kept completely consistent at all times. We keep the aggregates internally consistent at all times, while making other changes asynchronously. As changes propagate across nodes of a network, it can be difficult to resolve multiple updates arriving out of order or from distinct sources.

Therefore:

Model information about activity in the domain as a series of discrete events. Represent each event as a domain object. These are distinct from system events that reflect activity within the software itself, although often a system event is associated with a domain event, either as part of a response to the domain event or as a way of carrying information about the domain event into the system.

A domain event is a full-fledged part of the domain model, a representation of something that happened in the domain. Ignore irrelevant domain activity while making explicit the events that the domain experts want to track or be notified of, or which are associated with state change in the other model objects.

In a distributed system, the state of an entity can be inferred from the domain events currently known to a particular node, allowing a coherent model in the absence of full information about the system as a whole.

Domain events are ordinarily immutable, as they are a record of something in the past. In addition to a description of the event, a domain event typically contains a timestamp for the time the event occurred and the identity of entities involved in the event. Also, a domain event often has a separate timestamp indicating when the event was entered into the system and the identity of the person who entered it. When useful, an identity for the domain event can be based on some set of these properties. So, for example, if two instances of the same event arrive at a node they can be recognized as the same.

### Services

Sometimes, it just isn’t a thing.

Some concepts from the domain aren’t natural to model as objects. Forcing the required domain functionality to be the responsibility of an entity or value either distorts the definition of a model-based object or adds meaningless artificial objects.

Therefore:

When a significant process or transformation in the domain is not a natural responsibility of an entity or value object, add an operation to the model as a standalone interface declared as a service. Define a service contract, a set of assertions about interactions with the service. (See assertions.) State these assertions in the ubiquitous language of a specific bounded context. Give the service a name, which also becomes part of the ubiquitous language.

### Modules

Everyone uses modules, but few treat them as a full-fledged part of the model. Code gets broken down into all sorts of categories, from aspects of the technical architecture to developers’ work assignments. Even developers who refactor a lot tend to content themselves with modules conceived early in the project.

Explanations of coupling and cohesion tend to make them sound like technical metrics, to be judged mechanically based on the distributions of associations and interactions. Yet it isn’t just code being divided into modules, but also concepts. There is a limit to how many things a person can think about at once (hence low coupling). Incoherent fragments of ideas are as hard to understand as an undifferentiated soup of ideas (hence high cohesion).

Therefore:

Choose modules that tell the story of the system and contain a cohesive set of concepts. Give the modules names that become part of the ubiquitous language. Modules are part of the model and their names should reflect insight into the domain. This often yields low coupling between modules, but if it doesn’t look for a way to change the model to disentangle the concepts, or an overlooked concept that might be the basis of a module that would bring the elements together in a meaningful way. Seek low coupling in the sense of concepts that can be understood and reasoned about independently. Refine the model until it partitions according to high-level domain concepts and the corresponding code is decoupled as well.

(aka Packages)

### Aggregates

It is difficult to guarantee the consistency of changes to objects in a model with complex associations. Objects are supposed to maintain their own internal consistent state, but they can be blindsided by changes in other objects that are conceptually constituent parts. Cautious database locking schemes cause multiple users to interfere pointlessly with each other and can make a system unusable. Similar issues arise when distributing objects among multiple servers, or designing asynchronous transactions.

Therefore:

Cluster the entities and value objects into aggregates and define boundaries around each. Choose one entity to be the root of each aggregate, and allow external objects to hold references to the root only (references to internal members passed out for use within a single operation only). Define properties and invariants for the aggregate as a whole and give enforcement responsibility to the root or some designated framework mechanism.

Use the same aggregate boundaries to govern transactions and distribution.

Within an aggregate boundary, apply consistency rules synchronously. Across boundaries, handle updates asynchronously.

Keep an aggregate together on one server. Allow different aggregates to be distributed among nodes.

When these design decisions are not being guided well by the aggregate boundaries, reconsider the model. Is the domain scenario hinting at an important new insight? Such changes often improve the model’s expressiveness and flexibility as well as resolving the transactional and distributional issues.

# Books

## Behavioural and professional books

* [Software Craftsmanship: Professionalism, Pragmatism, Pride](https://leanpub.com/socra), by Sandro Mancuso.
* [The Clean Coder: A Code of Conduct for Professional Programmers](http://goo.gl/WGav4Y), by Robert C. Martin
* [Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman](http://goo.gl/OnbNlC), by Dave Hoover and Adewale Oshineye
* [The Pragmatic Programmer: from Journeyman to Master](http://goo.gl/4e30MZ), by Andrew Hunt and David Thomas
* [Software Craftsmanship: The New Imperative](http://goo.gl/QgysUa), by Pete McBreen
* [The Mythical Man-Month: Essays on Software Engineering](http://goo.gl/XcxVxI), by Frederick P. Brooks Jr.
* [The Passionate Programmer: Creating a Remarkable Career in Software Development](http://goo.gl/QgysUa), by Chad Fowler
* [Drive: The Surprising Truth About What Motivates Us](http://goo.gl/kZhyDF), by Daniel Pink
* [Agile Coaching](http://pragprog.com/book/sdcoach/agile-coaching), by Rachel Davies and Liz Sedley
* [Pragmatic Thinking and Learning](http://pragprog.com/book/ahptl/pragmatic-thinking-and-learning), by Andy Hunt
* [Agile Retrospectives](http://pragprog.com/book/dlret/agile-retrospectives), by Esther Derby and Diana Larsen
* [Impact Mapping: Making a Big Impact with Software Products and Projects](http://goo.gl/aPgcGi), by Gojko Adzic

## Technical books

* [Clean Code: A Handbook of Agile Software Craftsmanship](http://goo.gl/cEFlrK), by Robert C. Martin
* [Growing Object-Oriented Software, Guided by Tests](http://goo.gl/KX4FWD), by Steve Freeman and Nat Pryce
* [Working Effectively With Legacy Code](http://goo.gl/9wz6gz), by Michael Feathers
* [Refactoring: Improving the Design of Existing Code](http://goo.gl/vfOlP0), by Martin Fowler
* [Refactoring Workbook](http://www.amazon.co.uk/Refactoring-Workbook-Addison-Wesley-Object-Technology/dp/0321109295), by Bill Wake
* [Refactoring to Patterns](http://goo.gl/Q6A03w), by Joshua Kerievsky
* [Design Patterns: Elements of Reusable Object-Oriented Software](http://goo.gl/ZJCmlc), by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
* [Head First Design Patterns](http://goo.gl/5LyyJS), by Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra
* [Test-Driven Development by Example](http://goo.gl/u0mtDR), by Kent Beck
* [Domain-Driven Design: Tackling Complexity in the Heart of Software](http://goo.gl/SoR7xg), by Eric Evans
* [Implementing Domain Driven Design](http://goo.gl/GV63Cm), by Vaughn Vernon
* [Patterns of Enterprise Application Architecture](http://goo.gl/V7JYLJ), by Martin Fowler
* [Extreme Programming Explained: Embrace Change](http://goo.gl/ERIikp), by Kent Beck
* [Specification by Example: How Successful Teams Deliver the Right Software](http://goo.gl/Hox6XC), by Gojko Adzic

# Cheat sheets

<http://www.planetgeek.ch/2013/06/05/clean-code-cheat-sheet/>

<http://www.industriallogic.com/blog/smells-to-refactorings-cheatsheet/>

<https://www.jetbrains.com/resharper/docs/ReSharper80DefaultKeymap_VS_scheme.pdf>

<https://www.jetbrains.com/resharper/docs/ReSharper70DefaultKeymap_IDEA_scheme.pdf>

Make smaller things, smaller libraries, smaller classes, and smaller methods. Let them know as little about each other as possible.

## Definitions

* **Acceptance test**: Tests the functionality of a feature and how the whole system operates
* **Integration test**: Tests how code interacts with external/invariant code
* **Unit test**: Tests the functionality of a single object
* **End to end test**: Tests the system as if it were a black box and only interacts with it through the UI
* **Edge to edge test**: Tests every step from build to deployment to release
* **Coupling**: Change in one component forces change in another (i.e. the modularity, or lack thereof, of the system)
* **Cohesion**: Responsibilities form a meaningful unit
* **Role**: Related group of responsibilities
* **Responsibility**: Obligation to either perform a task or know information
* **Collaboration**: Interaction between objects or roles
* **Mockery**: jMock term that refers to the context of the object being tested
* **Mock object**: A test object that substitutes for objects that interact with the object under test
* **Expectations**: Rules that define how mock objects should be invoked
* **"Walking skeleton"**: Most minimal implementation necessary to have an end-to-end test
* **Encapsulation**: Behavior of an object can only be affected through methods for interaction with other objects
* **Information hiding**: How object functions remains internal and invisible to other objects
* **Aliasing**: Sharing references to mutable objects, breaks encapsulation
* **Peers**: Objects with which a given object communicates
* **Dependencies**: Services from peers without which the object cannot function
* **Notifications**: Peers that need to be updated with the object's behavior or status
* **Adjustments**: Peers that adjust the object's behavior to work with the rest of the system
* **Context independence**: Object has no internal knowledge about its environment
* **Interface**: "Whether two components will fit together"
* **Protocol**: "Whether two components will work together"
* **Spike**: Initial code written to figure out what to do, later rolled back and rewritten more cleanly
* **Implementation layer**: Describes how the code will do something
* **Declarative layer**: Describes what the code will do

# Assessment

* Level zero – Ignorance
  + Does not know XP Practices
  + Does not write any tests
* Level 1 - Few Simple Tests
  + Knows what Red, green, refactor means but does not apply consistently
  + Writes a few tests to prove production code behaves correctly
  + Writes tests after production code
* Level 2 – Basic classic TDD
  + Uses tests to drive algorithms
  + Simple refactoring’s
  + Baby steps
  + How to test
    - Simple katas using TDD rules
      * <http://osherove.com/tdd-kata-1/>
* Level 3 - Test doubles
  + Uses Test doubles to isolate code under test.
  + Separates state tests and collaboration tests.
  + How to test
    - More complex kata using TDD rules
      * <http://osherove.com/tdd-kata-2/>
* Level 4 - Design for Testability
  + SOLID Principles
  + Code Smells are avoided
  + Refactoring
  + DRY, YAGNY, TDA, principles
  + How to test
    - More complex kata using TDD rules, plus Object Calisthenics
      * Tik Tak Toe or Game of life Katas
* Level 5 – Outside in TDD
  + Drives code from outside
  + How to test
    - More complex kata using ATDD rules
      * Bank Kata
* Level 6 - Test Driven Development
  + Tests drive design and code.
  + Switches between London and classic TDD styles.
  + How to test
    - Social networking Kata
      * <http://monospacedmonologues.com/post/49250842364/the-social-networking-kata>

# Action plan

* Level zero – Ignorance
  + XP intro
  + TDD intro
  + Pair programming techniques
  + Fizz buzz kata
  + Leap years kata
  + http://codingkata.net/Katas
* Level 1 - Few Simple Tests
  + TDD habits
  + Fibonacci sequence kata
  + Transformation Priority Premise
  + Roman numbers kata
* Level 2 – Basic classic TDD
  + Object Calisthenics
  + The four elements of simple design
  + Tik-Tak-toe kata using Object Calisthenics
  + Test doubles
* Level 3 - Test doubles
  + Code smells
  + Refactoring
  + SOLID principles
  + DRY, YAGNY, TDA, principles
  + Resharper Workshops
    - <https://github.com/JetBrains/resharper-workshop>
    - <https://github.com/rasmuskl/ReSharperCourse/blob/master/Source/ReSharper%20Exercises.pdf>
  + Gilded Rose
    - <https://github.com/emilybache/GildedRose-Refactoring-Kata>
* Level 4 - Design for Testability
  + Outside in TDD
  + Bank Kata
  + Mars Rover
* Level 5
  + Graduation project

1. By combining the direction information from two or more suitably spaced receivers (or a single mobile receiver), the source of a transmission may be located via triangulation. Only generalize code when we have two examples or more… When the second example demands a more general solution, then and only then do we generalize. [↑](#footnote-ref-1)