Agile Engineering

Fábio A Falavinha

All Practices

- Architecture vs Agile Design
- Design Patterns
- Coding Standards
- Test-First
- Behavior-Driven Development
  - Test Driven-Development
- Unit Testing
- Pair-Programming
- Refactoring
- Code Ownership
- Continuous Integration

Architecture

"… conveys a notion of the core elements of the system, the pieces that are difficult to change. A foundation on which the rest must be built"

by Martin Fowler

Architecture

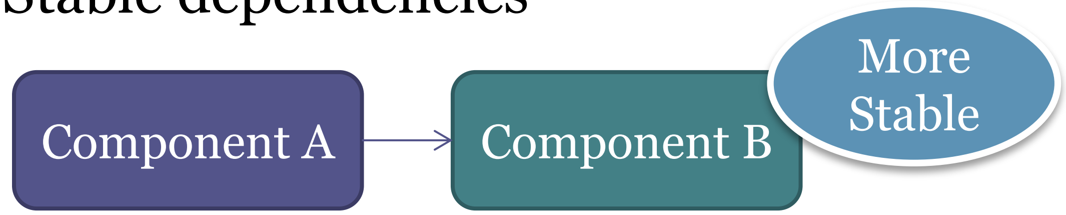
- Facebook "Yesterday"
  - 'Move Fast and Break Things'
    - Tolerates a few bugs to move faster
- Facebook "Today"
  - 'Move Fast with Stable Infra'
    - "It might not have the same ring to it and might not be as catchy… but it helps us build better experiences for everyone we serve and how we operate now." by Mark Zuckerberg

Agile Design Principles

- Acyclic dependencies
  - Say NO for cyclic dependencies!
- Common closure
  - Components should be cohesive!
  - Changes in a component CAN'T affect outside classes!
- Common reuse
- Dependency Inversion
  - Abstractions should not depend on details
  - Details should depend on Abstractions

Agile Design Principles

- Open-Closed
  - Open for extension
  - Closed for modification
- Release-reuse equivalency
  - YOU should NOT reuse <u>part of</u> a released software element
- Stable abstractions
  - Components should be <u>sufficiently</u> abstract so that they can be extended
- Stable dependencies



Agile Design vs Architecture

“focuses on adjusting the design and plan as more insight is gained into the domain”

“Architecture establishes the technology stack”

“agile techniques are leveraged to drive towards the desired architecture”

“agility often needs a backbone to give it some direction”

Design Smells

Rigidity

Difficult to change

- Avoid static
- Use interfaces instead of pure objects
- Everything in .NET C# is non-virtual

Fragility

Single change = many places break

- Use strong types
- Tiny types
- Avoid too many arguments

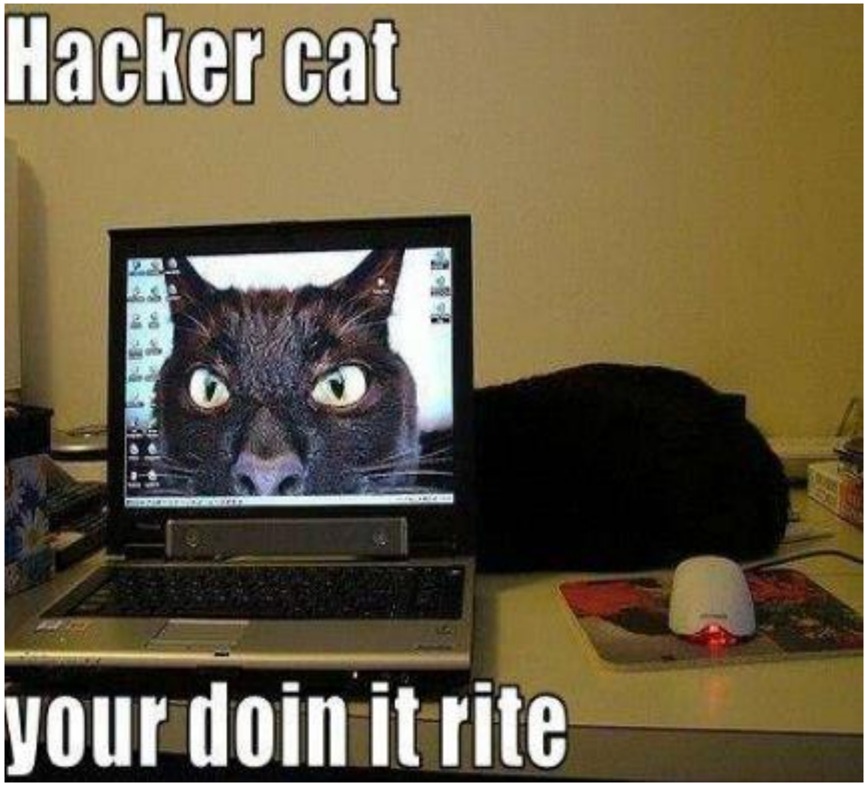
Immobility

Hard to move

- Avoid too many dependencies

Viscosity of Software

Easier to hack than preserve the design



- Don’t hack frameworks or APIs
- Avoid abstractions by pass
- Avoid public methods, prefer private

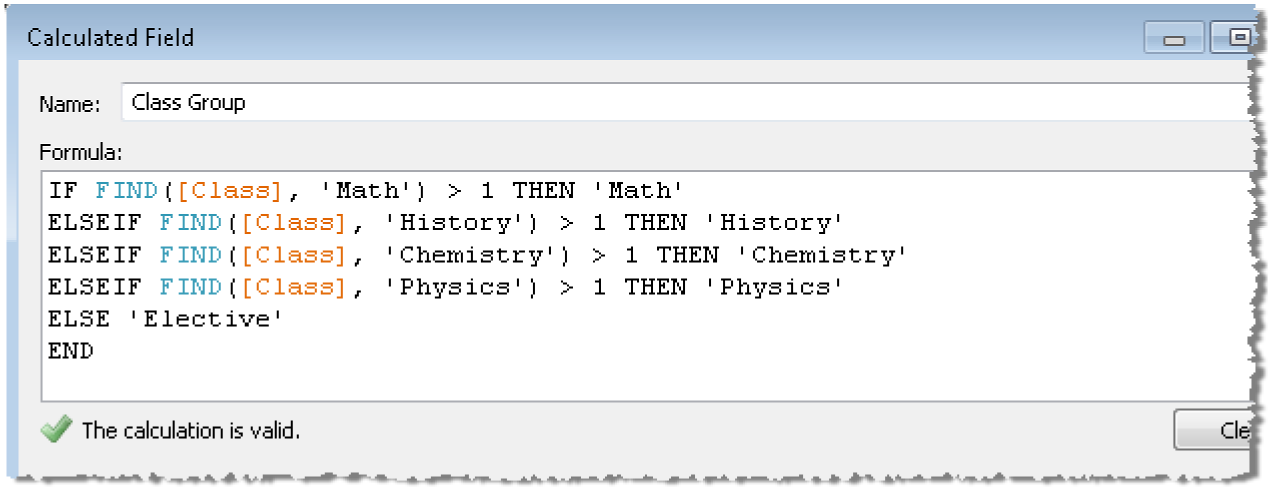
Viscosity of Environment

Not all environments are born equal (DEV, QA, PROD)

Needless Complexity

Software contains element that ain't useful at the time ("maybe" in the future)

- Java Vector was **synchronous** and didn't have interface
- Java StringBuffer is synchronous
- Java StringBuilder isn't synchronous


Calculated Field

Name: Class Group

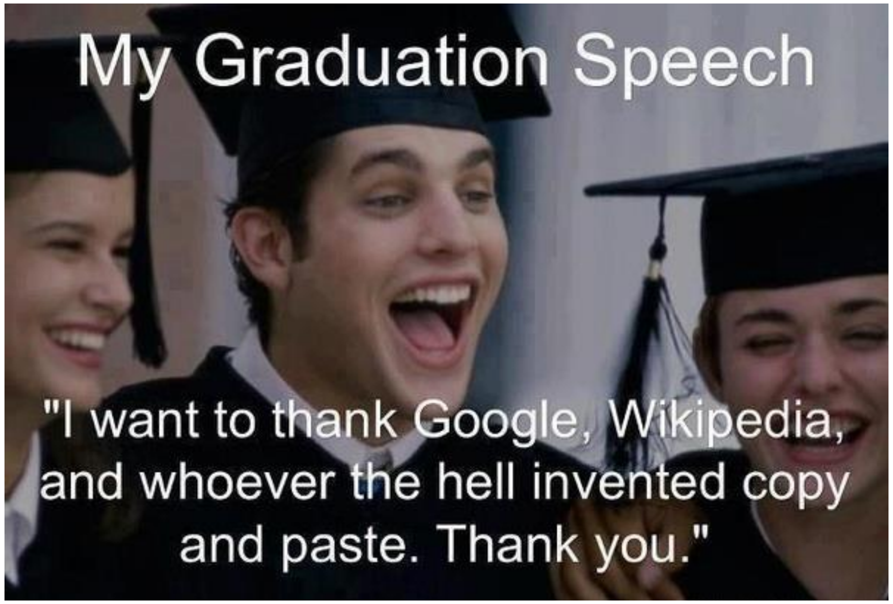
Formula:

```
IF FIND([Class], 'Math') > 1 THEN 'Math'
ELSEIF FIND([Class], 'History') > 1 THEN 'History'
ELSEIF FIND([Class], 'Chemistry') > 1 THEN 'Chemistry'
ELSEIF FIND([Class], 'Physics') > 1 THEN 'Physics'
ELSE 'Elective'
END
```

The calculation is valid.

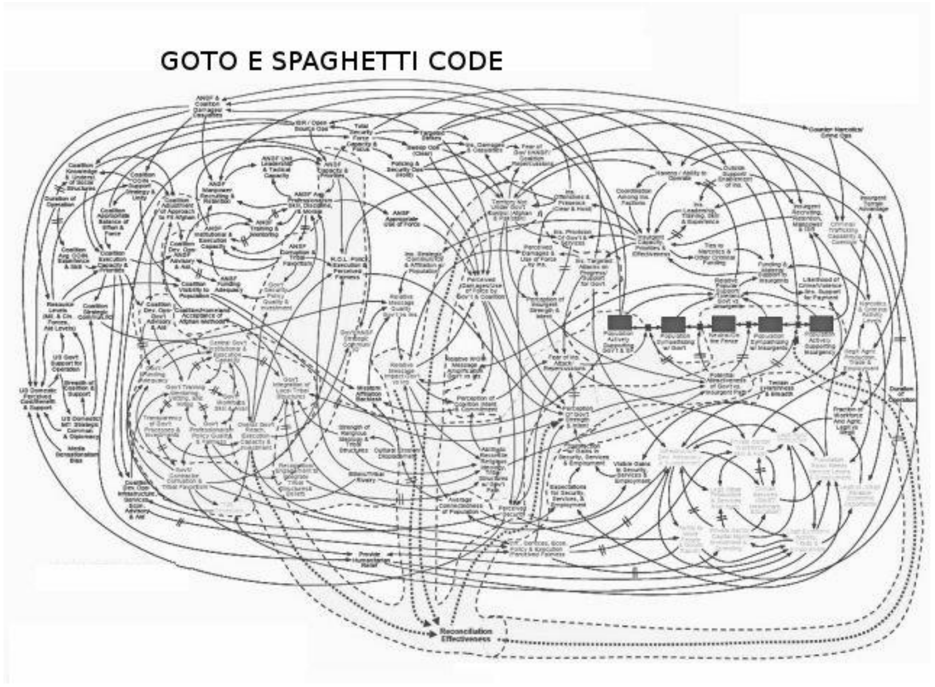
Needless Repetition

- "copy & paste" code without refactoring
- Dry Principle (Don’t Repeat Yourself)
  - Avoid code duplicates

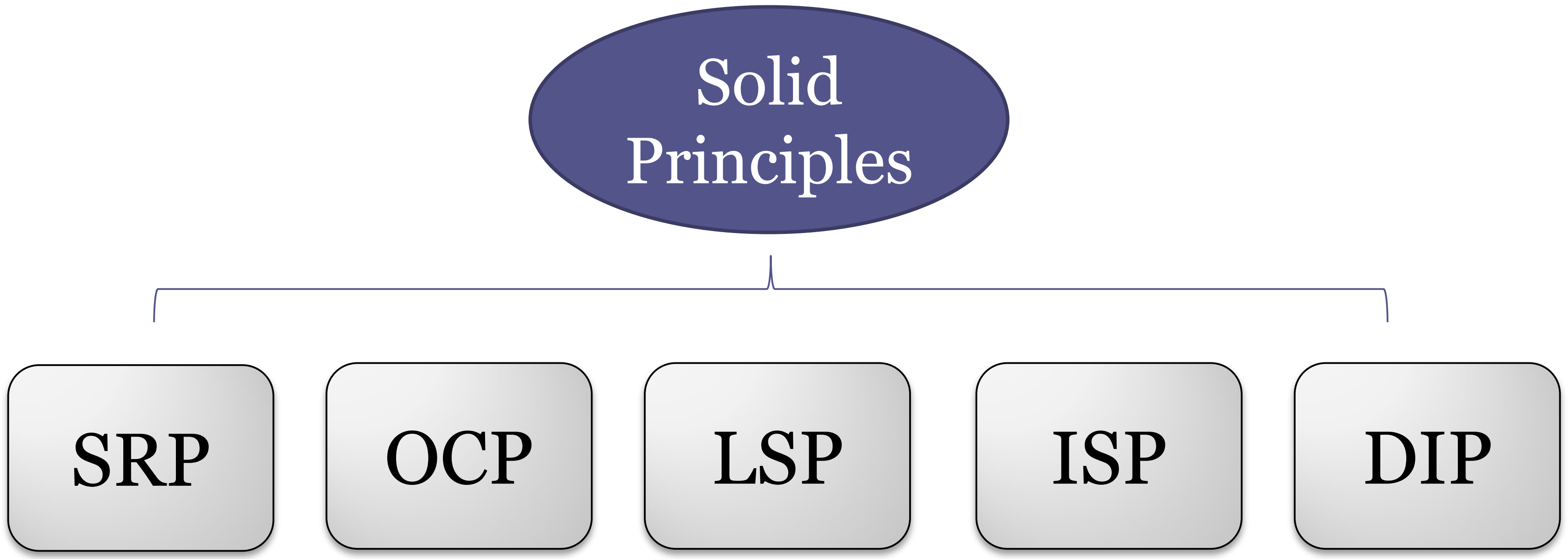

My Graduation Speech
"I want to thank Google, Wikipedia, and whoever the hell invented copy and paste. Thank you."

Opacity

- Difficulty to understand - code review?
- Avoid non-human readable variable names



How to avoid/remove those design smells?



Single Resposibility Principle

- Most simple principle but hardest to define
- What is the single responsibility of each unit?
- Single Responsibility = single reason for change
- Unit should have one and only one reason to change
- Example:
  - This class does too much. Help! [O famoso **manager** by Alexandre Cunha]

- Easy way to follow this principle:
  - Constantly ask yourself whether every method and operation of a unit is directly related to its abstraction

Open Closed Principle

- Open for extension, close for modification
- How to achieve OCP?
  - Force the system's design to be testable (TDD)
- We will have to build an abstraction to make the system testable

Liskov Substitution Principle

- Look for a derivative class:
  - Which removes functionality from the base class and does less than its base
  - Not substitutable from base class
  - Example: explicit vs implicit implementation using .NET C#

```
var array = new KeyValuePair<string, string>[0];
Dictionary<string, string>  dic = new Dictionary<string, string>();
dic.CopyTo(array, 0);
```

Compile?
No. WTF!

We can change public interface contract!!!!

Interface Segregation Principle

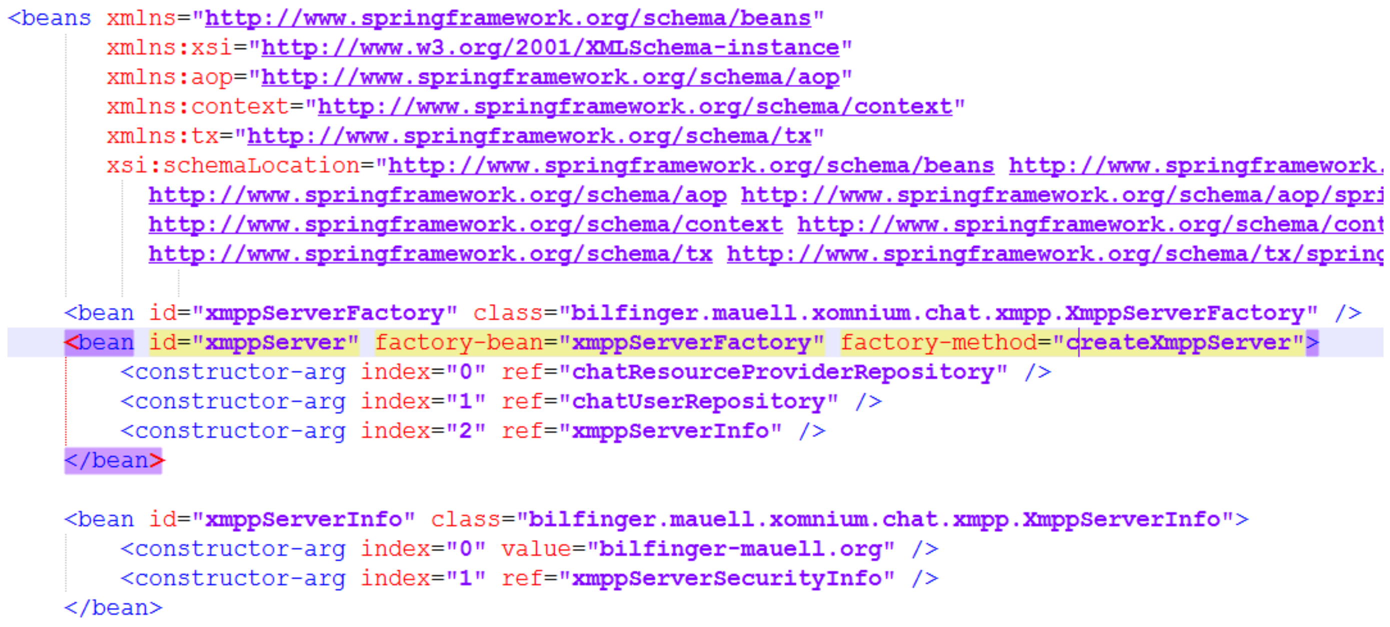
- Dealing with "fat" interface
  - which can be broken up in groups of methods and serves different clients
- Client should not be forced to depend on methods that do not use
- .NET C# examples:
  - IList vs IReadOnlyList
  - IEnumerable (GetEnumarator)

Dependency Inversion

- Avoid new operator for dependency
- Prefer constructor dependency (required)
  - NodeContext.GetContext() : INodeContext
    - This is a service locator
- Property dependency (non-required)
- Service Locator pattern (fetch)
- Factory pattern (substitution of new operator)

Dependency Injection

- Decouple application glue code from application logic
- Spring is an anti-pattern

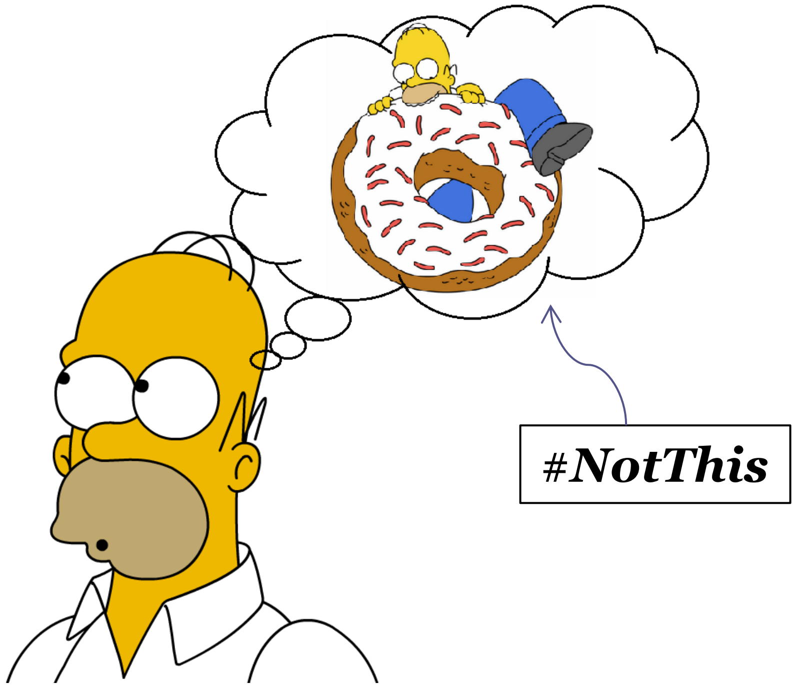


```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.
          http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spri
          http://www.springframework.org/schema/context http://www.springframework.org/schema/cont
          http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring

    <bean id="xmppServerFactory" class="bilfinger.mauell.xomnium.chat.xmpp.XmppServerFactory" />
    <bean id="xmppServer" factory-bean="xmppServerFactory" factory-method="createXmppServer">
        <constructor-arg index="0" ref="chatResourceProviderRepository" />
        <constructor-arg index="1" ref="chatUserRepository" />
        <constructor-arg index="2" ref="xmppServerInfo" />
    </bean>

    <bean id="xmppServerInfo" class="bilfinger.mauell.xomnium.chat.xmpp.XmppServerInfo">
        <constructor-arg index="0" value="bilfinger-mauell.org" />
        <constructor-arg index="1" ref="xmppServerSecurityInfo" />
    </bean>
```

Design Patterns

“General reusable solution to a commonly occurring problem within a given context in software design”

It's a paradigm!



Design Patterns

Design patterns are good

False assumption. Design patterns are neutral, some of them are less applicable than others and some are almost anti-patterns in some contexts (singleton).

Design Patterns

Design patterns are complex

<u>False assumption</u>. Some design patterns are actually very simple, such as the *template method pattern*. Some are very complex, such as the *visitor pattern*. On average the simple patterns have more uses than the complex ones.

Design Patterns

Should we use them in small projects?
How small is small?

A Project would have to be extremely small to be well-designed and yet not include any recognizable design pattern. It might not include many, and will almost certainly skip some of the more complex structural or behavioral patterns.

Design Patterns

Implementing design patterns needs more "*sophisticated*" developers

Partially true for the more complex patterns - but absolutely not for all patterns, and even more so for the types of patterns you are likely to put to good use in smaller projects

Design Patterns

Raises project costs

Categorically untrue if patterns are applied correctly. What may raise your project cost is <u>lack of design</u>, <u>bad design</u>, or <u>wrongly/overly applied design patterns</u>.

Design Patterns

They make code neat and clean

All the best code is as neat and clean as it could possibly be made, this is not exclusive to the intentional application of design patterns.

Design Patterns

Are they necessary for small projects

That depends on your project. If you can get away with code that fulfills its purpose but is very hard to maintain, that's fine. However if maintenance is likely then the cost you save up front by not considering patterns or design as a whole will be paid back quite quickly in extra pain and duration during maintenance.

Design Patterns

"Patterns provide a mechanism for rendering design advice in a reference format"

By Martin Fowler

Learn agile design principles first then design patterns

Coding Standards

In team software development, we create a collective work that is greater than any individual could create on his own. Arguing about style gets in the way.

by James Shore, The Art Of Agile

Coding Standards

Coding Standards

"It's easier to maintain and extend code, to refactor it, and to reconcile integration conflicts, if a common standard is applied consistently throughout. The standard itself matters much less than adherence to it. Fortunately, modern IDEs make it trivial to apply many kinds of formatting, even after the fact"

by VersionOne, Agile Made Easier

Test-first

- Product Owner define stories and acceptance criteria to be tested
- Coverage levels
  - 75% - 85%

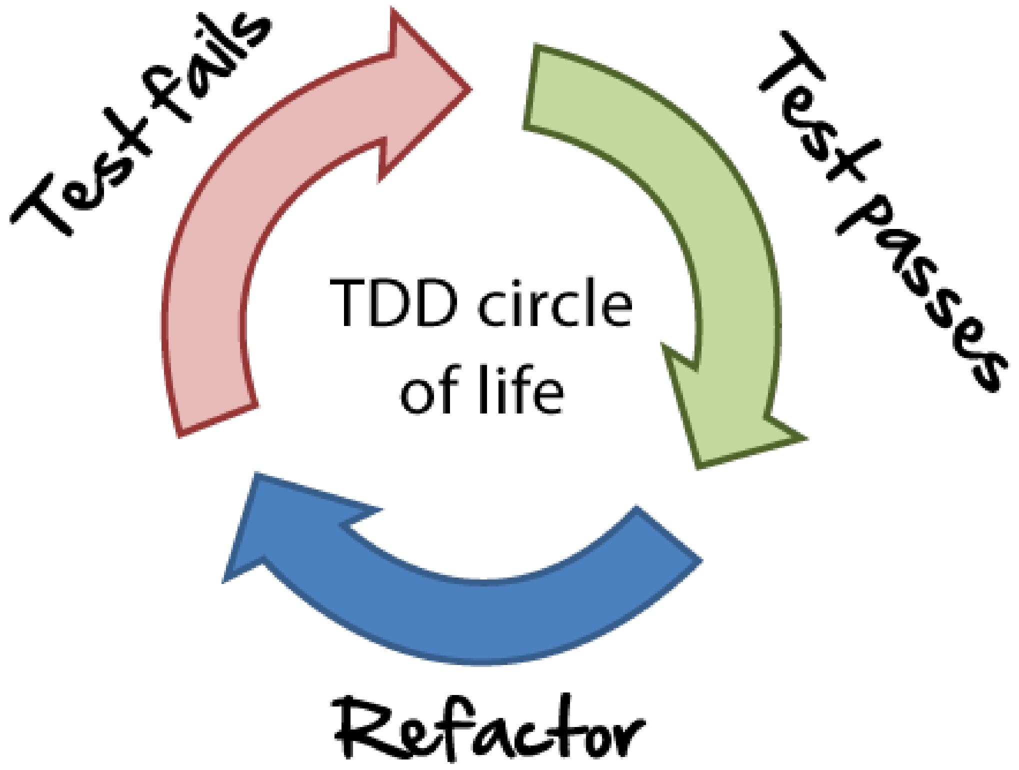
Behavior Driven Development

- Based on TDD (Test Driven Development, Introduction)
  - Combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented

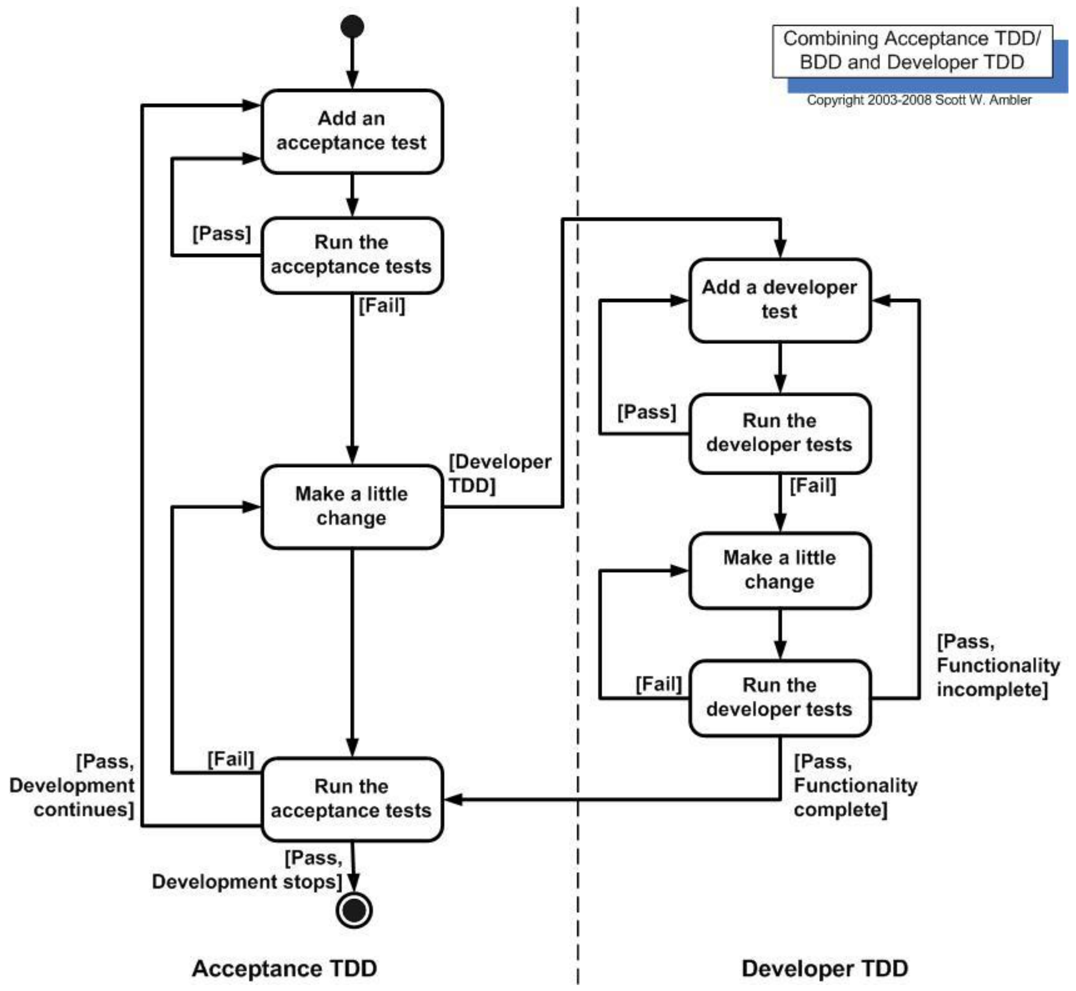
Behavior Driven Development

```
Story: Returns go to stock

In order to keep track of stock
As a store owner
I want to add items back to stock when they're returned

Scenario 1: Refunded items should be returned to stock
Given a customer previously bought a black sweater from me
And I currently have three black sweaters left in stock
When he returns the sweater for a refund
Then I should have four black sweaters in stock

Scenario 2: Replaced items should be returned to stock
Given that a customer buys a blue garment
And I have two blue garments in stock
And three black garments in stock.
When he returns the garment for a replacement in black,
Then I should have three blue garments in stock
And two black garments in stock
```

Test Driven Development



Combine(TDD, BDD)



Last from tests… Until the end :P

If it's worth building, it's worth testing.

If it's not worth testing, why are you wasting your time working on it?

Unit Testing

- Testing the unit of work (function, class, component, layer, system)
  - "Protect" API design from non-business acceptance
- Explore state space
- .NET C# Struct example
  - Fraction (avoid division by zero)

Pair-Programming

- One programmer focuses on the code at hand: the syntax, semantics, and algorithm

- The other programmer focuses on a higher level of abstraction:
  - Tests
  - Technical task
  - Time elapsed since all the tests were run
  - Time elapsed since the last repository commit
  - Quality of the overall design

Refactoring

- Restructuring an existing body of code
  - altering its internal structure without changing its external behavior
- Small transformations
  - Small refactorings

Refactoring

"refactoring is the technique we should use to introduce Gang of Four design patterns into our code.“

by Joshua Kerievsky

Why?

Because patterns are often over-used, and often introduced too early into systems.

Code Ownership

- Boy Scout Rule
  - When you see a non complying code you must correct it
- Strong code ownership
  - Component Lead
- Weak code ownership
- Collective code ownership
  - Team knows all design and architecture models

Continuous Integration

- Team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day

- Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible

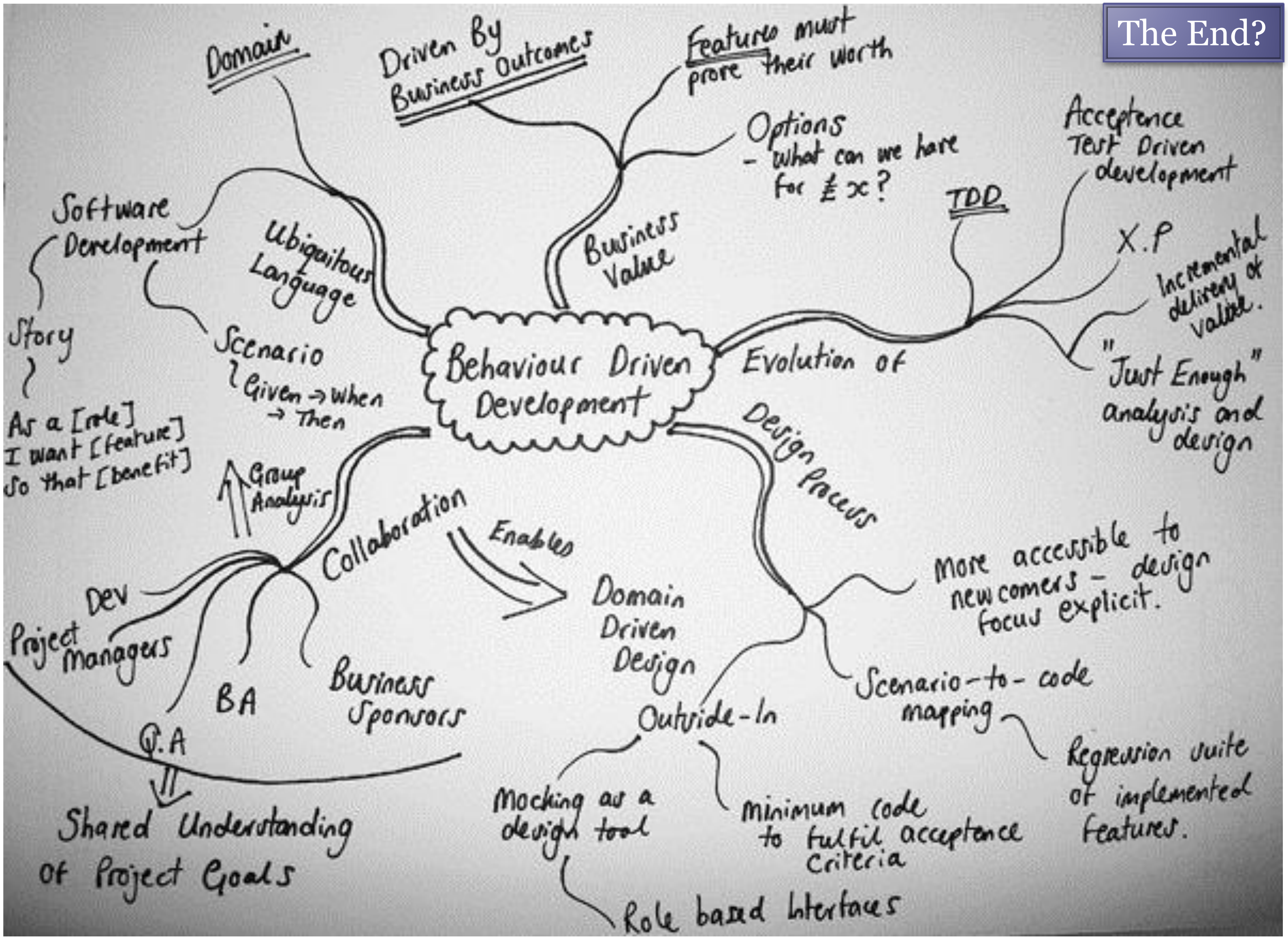
Continuous Integration

1. Maintain a code repository
2. Automate the build
3. Make the build self-testing
4. Everyone commits to the baseline every day
5. Every commit (to baseline) should be built
6. Keep the build fast
7. Test in a clone of the production environment
8. Make it easy to get the latest deliverables
9. Everyone can see the results of the latest build
10. Automate deployment

Continuous Integration

- Microsoft Team Foundation Server
- Cruise Control
- Hudson
- Jenkins

The End?


Behaviour Driven Development
Domain
Driven By Business Outcomes
Features must prove their worth
Options - what can we have for £x?
Business Value
Acceptance Test Driven development
TDD
X.P
Incremental delivery of value.
"Just Enough" analysis and design
Evolution of
Design Process
More accessible to new comers - design focus explicit.
Scenario-to-code mapping
Regression suite of implemented features.
Outside-In
Mocking as a design tool
Role based Interfaces
minimum code to fulfil acceptance criteria
Domain Driven Design
Enables
Collaboration
Group Analysis
Dev
Project managers
BA
Business Sponsors
Q.A
Shared Understanding of Project Goals
Software Development
Ubiquitous Language
Scenario
Given → When → Then
Story
As a [role]
I want [feature]
So that [benefit]

References

- The Object Primer (on Falavinha's Desk)
- martinfowler.com/ieeeSoftware/patterns.pdf
- en.wikipedia.org/wiki/Dependency_inversion_principle
- en.wikipedia.org/wiki/Behavior-driven_development
- en.wikipedia.org/wiki/Unit_testing
- martinfowler.com/articles/continuousIntegration.html
- www.infoq.com/news/2011/06/agile-architecture-conflict/
- www.rallydev.com/sites/default/files/principles_of_agile_architecture.pdf
- www.apolloseiko.com/cmssite/cms_pdf/phpz4.pdf
- www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

Agile Engineering

Fábio A Falavinha