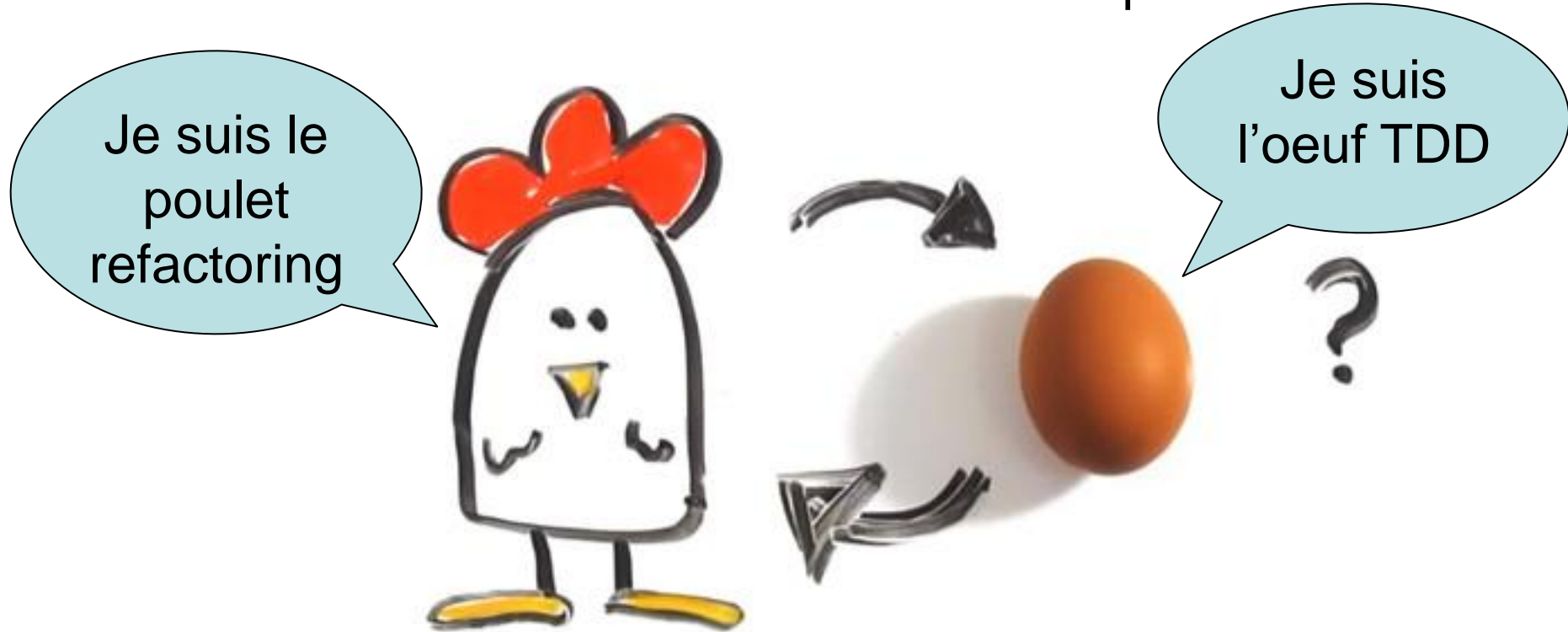


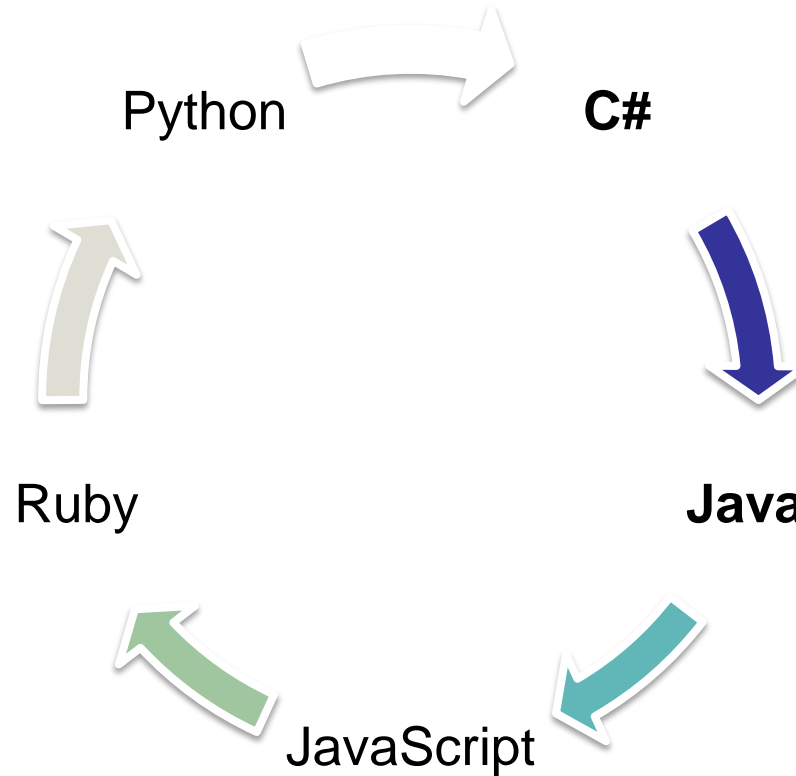
Refactoring legacy code driven by tests

Luca Minudel + Saleem Siddiqui

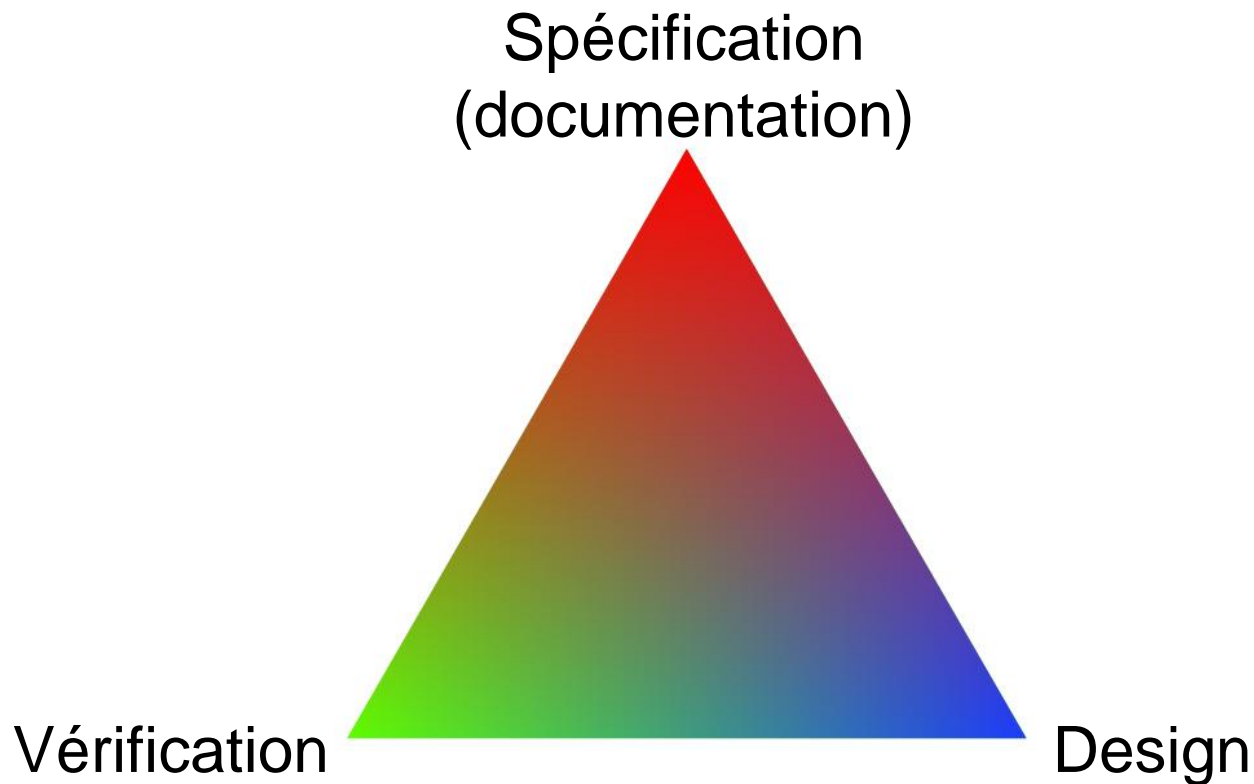


Clarifions le sujet

Langages supportés dans cet atelier

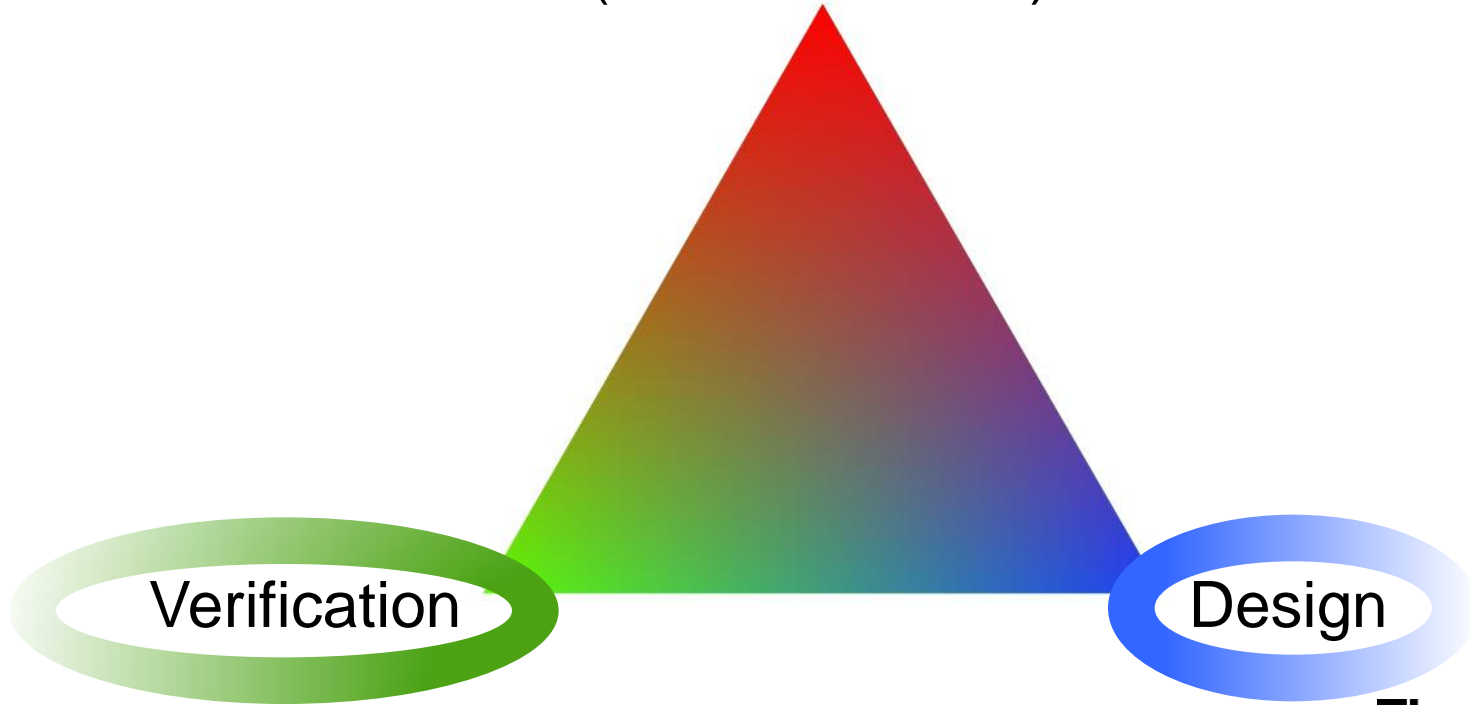


Continuum des tests automatisés



Scope de cet atelier

Specification
(documentation)



Types de tests automatisés

End-to-end, out-of-process, business facing



Unit, in-process, technology facing

Scope de cet atelier

End-to-end, out-of-process, business facing



Unit, in-process, technology facing

Exercice 1: Système de surveillance de pression des pneus

La classe Alarm :

Surveille la pression des pneus et déclenche une alarme si la pression sort de la plage prévue.

Exercice 1: Système de surveillance de pression des pneus

La classe Alarm :

Surveille la pression des pneus et déclenche une alarme si la pression sort de la plage prévue.

La classe Sensor:

Simule le comportement d'un capteur de pneu réel, en fournissant des valeurs aléatoires mais réalistes.

Exercice 1: Système de surveillance de pression des pneus

Ecrivez les tests unitaires pour la **classe Alarm**.

Refactorisez le code autant que nécessaire pour **rendre la classe Alarm testable**.

Exercice 1: Système de surveillance de pression des pneus

Ecrivez les tests unitaires pour la **classe Alarm**.

Refactorisez le code autant que nécessaire pour **rendre la classe Alarm testable**.

Minimisez les changements à l'API publique autant que possible.

Exercice 1: Système de surveillance de pression des pneus

Ecrivez les tests unitaires pour la **classe Alarm**.

Refactorisez le code autant que nécessaire pour **rendre la classe Alarm testable**.

Minimisez les changements à l'API publique autant que possible.

bonus:

La classe alarme ne suit pas un ou plusieurs principes SOLID. Donnez à chaque fois le numéro de ligne, le principe et la violation.

The SOLID acronym

S	single responsibility	principle
O	open closed	principle
L	Liskov substitution	principle
I	interface segregation	principle
D	dependency inversion	principle

Dependency Inversion Principle (DIP)

Définition de Martin Fowler:

a) Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau, les deux doivent dépendre des abstractions.

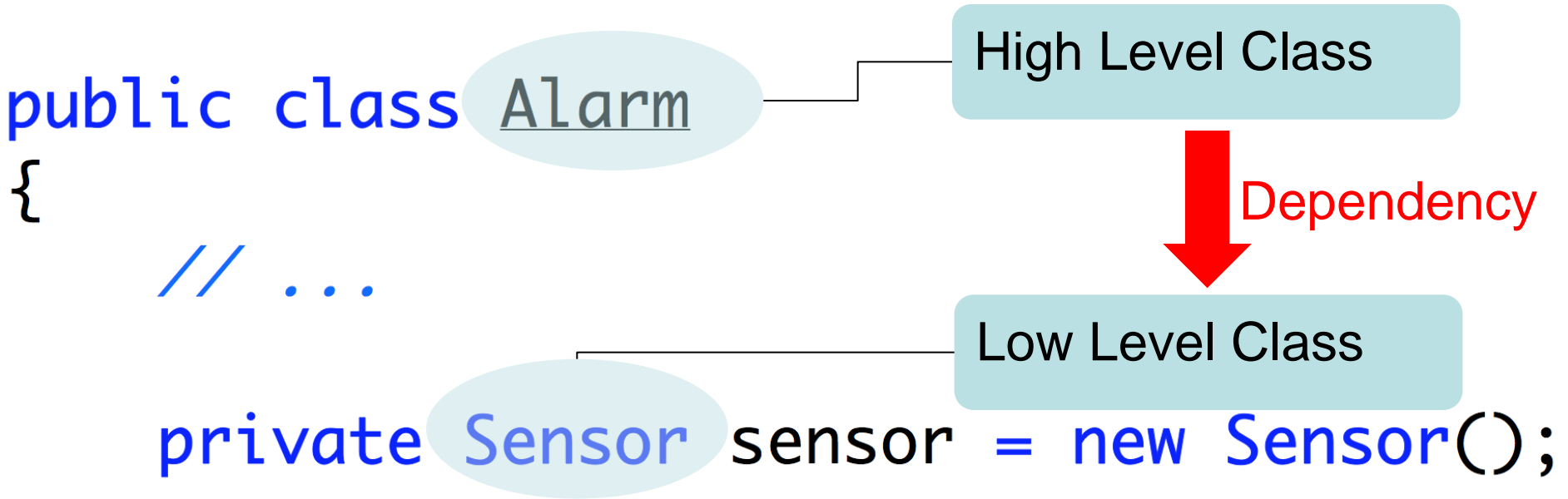
B) Les abstractions ne doivent pas dépendre des détails, les détails doivent dépendre des abstractions.

Dependency Inversion Principle (DIP)

Les classes de bas niveau et les classes de haut niveau doivent dépendre des abstractions.

Les classes de haut niveau ne doivent pas dépendre de classes de bas niveau.

Exemple de violation du DIP



Open Closed Principle (OCP)

Définition de Bertrand Meyer:

Les entités logicielles (classes, modules, fonctions, etc.) doivent être ouvertes pour extension, mais fermées pour modification.

Open Closed Principle (OCP)

**Les classes et méthodes doivent être
ouvertes pour les extensions
&
Stratégiquement fermées pour la modification.**

**Ainsi le comportement peut être changé et
étendu en ajoutant le nouveau code au lieu de
modifier la classe.**

Exemple de violation du principe OCP

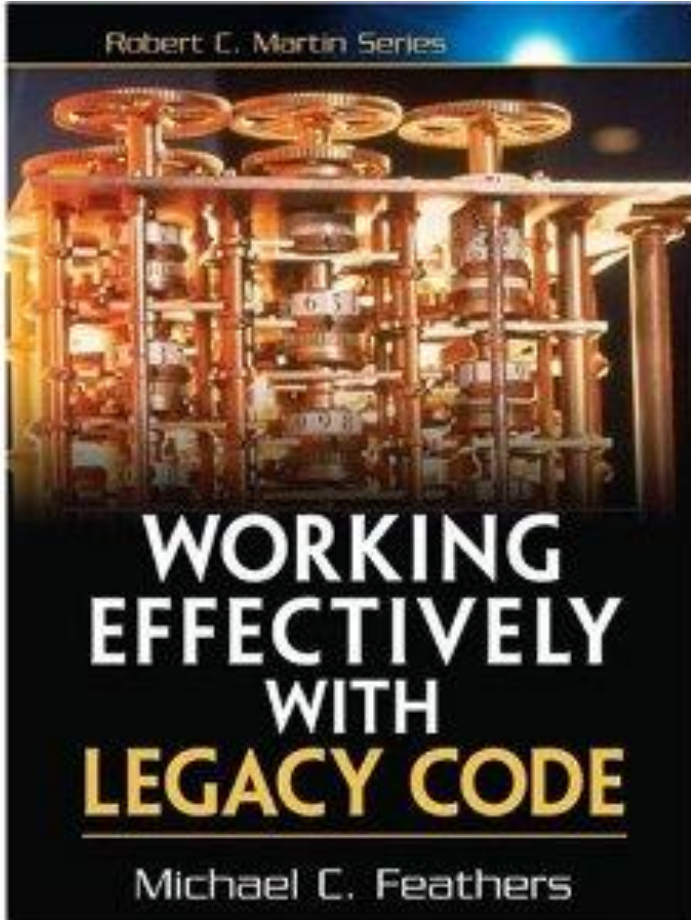
```
public class Alarm  
{  
    // ...
```

```
    private Sensor sensor = new Sensor();
```

Vous voulez utiliser un nouveau type de senseur ?

Vous devez modifier le code, vous ne pouvez pas l'étendre

Reference: WELC



- ✂ Paramétrez les constructeurs
- ✂ Extrayez les interfaces

Exercice 2: Conversion d'unicode vers HTML

La classe `UnicodeFileToHtmTextConverter` :

Convertit un fichier simple pour l'afficher dans un navigateur.

Exercice 2: Conversion d'unicode vers HTML

Ecrivez les tests unitaires pour la **classe** **UnicodeFileToHtmTextConverter** .

Refactorez le code autant que nécessaire pour **la classe** **testable**.

Exercise 2: Unicode File To Htm Text Converter

Ecrivez les tests unitaires pour la **classe UnicodeFileToHtmTextConverter** .

Refactorisez le code autant que nécessaire pour **la classe testable**.

Minimisez les changements à l'API publique autant que possible.

Exercise 2: Unicode File To Htm Text Converter

Ecrivez les tests unitaires pour la **classe UnicodeFileToHtmTextConverter** .

Refactorisez le code autant que nécessaire pour **la classe testable**.

Minimisez les changements à l'API publique autant que possible.

bonus:

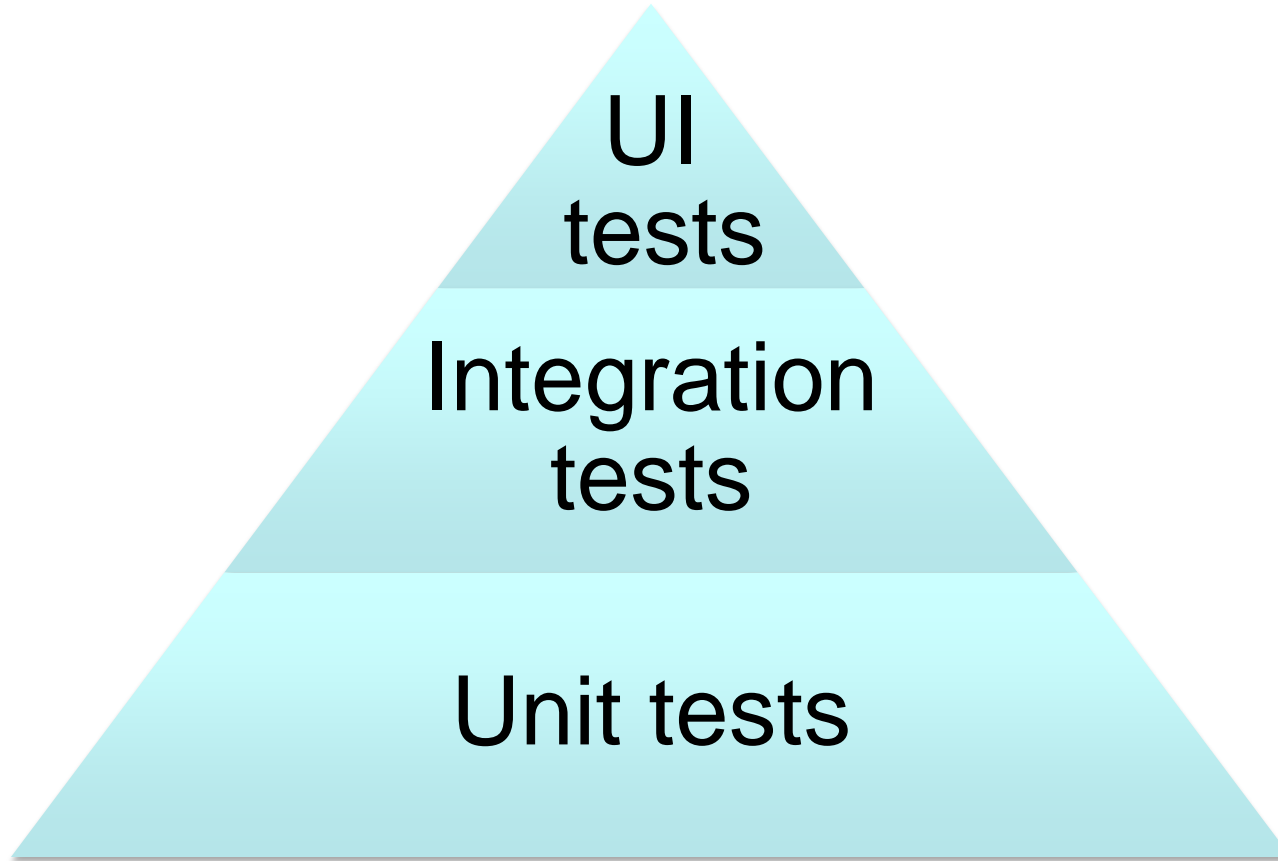
La classe UnicodeFileToHtmTextConverter ne suit pas un ou plusieurs principes SOLID. Donnez à chaque fois le numéro de ligne, le principe et la violation.

Règles générales de Feathers. Extended !

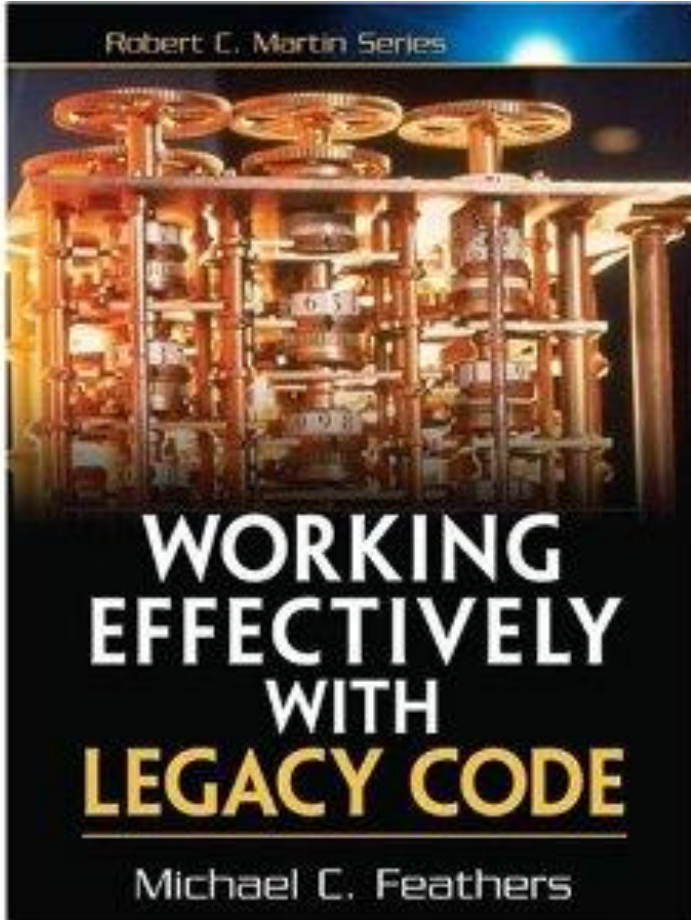
Un test n'est pas un test unitaire quand

- ★ Il parle à la base de données
- ★ Il communique à travers le réseau
- ★ Il touche le système de fichiers ou lit les informations de configuration
- ★ Il utilise `DateTime.now ()` ou aléatoire
- ★ Il dépend d'un comportement non déterministe
- ★ Il ne peut pas être exécuté en même temps que vos autres tests unitaires
- ★ Vous devez faire des choses spécifiques à votre environnement (comme l'édition de fichiers de configuration) pour l'exécuter.

Pyramide des tests de Mike Cohn's. Expliquée !



Reference: WELC



- ✂ Paramétrez les constructeurs
- ✂ Extrayez les interfaces
- ✂ Skin and Wrap the API

Refactoring et TDD

```
public string ConvertToHtml()  
{  
    using (TextReader unicodeFileStream = File.OpenText(_fullFilenameWithPath))  
    {  
        string html = string.Empty;  
  
        string line = unicodeFileStream.ReadLine();  
  
        // ... conversion details omitted  
  
        return html;  
    }  
}
```

Devrions-nous injecter cette dépendance ?

Comportement de TextReader

Documentation de TextReader tirée du MSDN

are discarded. Because the position of the reader in the stream cannot be changed, the characters that were already read are

Comportement non-idempotent

```
public UnicodeFileToHtmlTextConverter(IUnicodeTextSource textSource)
{
    _textSource = textSource;
}

public string ConvertToHtml()
{
    using (TextReader unicodeFileStream = _textSource.GetTextReader())
    {
        string html = string.Empty;

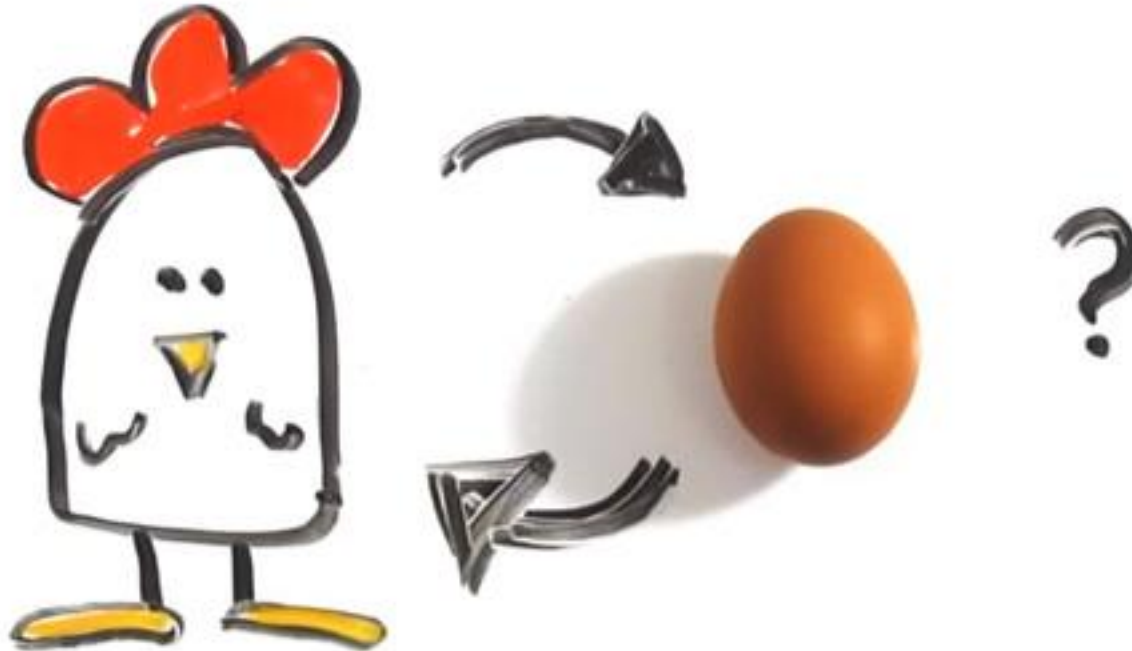
        string line = unicodeFileStream.ReadLine();

        // ... conversion details omitted

        return html;
    }
}
```

Injection de dépendance et
Comportement idempotent

Refactoring et TDD



Exercice 3: Ticket Dispenser

La classe TicketDispenser :

Gère un système de file d'attente dans un magasin.

Il peut y avoir plus d'un distributeur de billets, mais le même billet ne devrait pas être délivré à deux clients différents.

Exercice 3: Ticket Dispenser

La classe TurnTicket:

Représente le billet avec le numéro de tour.

La classe TurnNumberSequence:

Retourne la séquence des numéros de tour.

Exercice 3: Ticket Dispenser

Ecrivez les tests unitaires pour la **classe TicketDispenser**.

Refactorisez le code autant que nécessaire pour **la classe testable**.

Exercice 3: Ticket Dispenser

Ecrivez les tests unitaires pour la **classe TicketDispenser**.

Refactorisez le code autant que nécessaire pour **la classe testable**.

Minimisez les changements à l'API publique autant que possible.

Exercice 3: Ticket Dispenser

Ecrivez les tests unitaires pour la **classe TicketDispenser**.

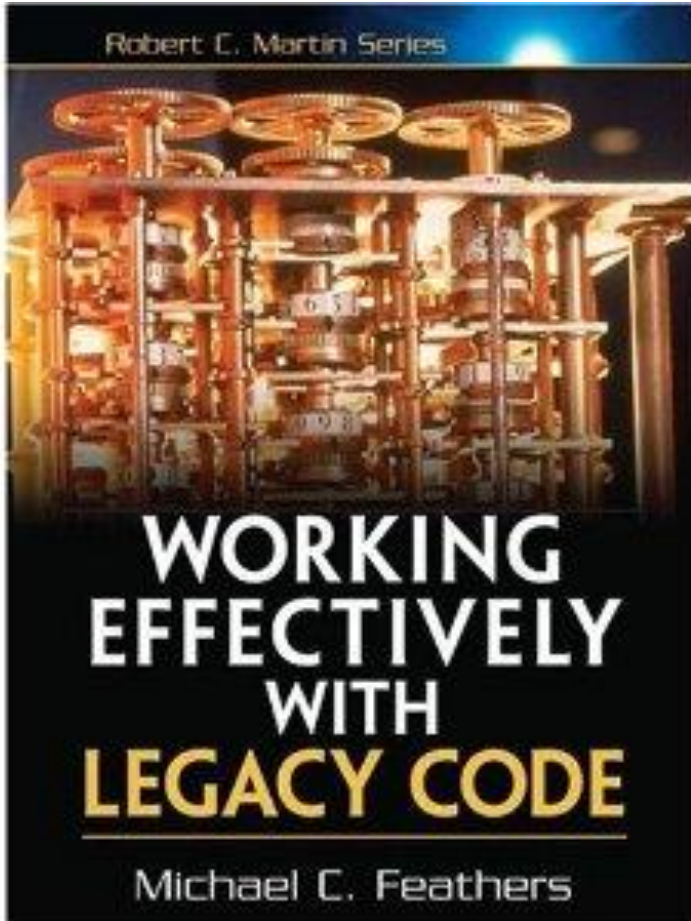
Refactorez le code autant que nécessaire pour **la classe testable**.

Minimisez les changements à l'API publique autant que possible.

bonus:

La classe TicketDispenser ne suit pas un ou plusieurs principes SOLID ou objet. Donnez à chaque fois le numéro de ligne, le principe et la violation.

Reference: WELC



- ✂ Paramétrez les constructeurs
- ✂ Extrayez les interfaces
- ✂ Skin and Wrap the API
- ✂ Introduire un délégué dans l'instance
- ✂ ...

Exercice 4: Système de télémétrie

La classe **TelemetryDiagnosticControl**:

Établit une connexion au serveur de télémétrie via TelemetryClient,
Envoie une requête de diagnostic et reçoit la réponse avec des informations de diagnostic.

La classe **TelemetryClient** :

Simule la communication avec le serveur de télémétrie, envoie des demandes, puis reçoit et retourne les réponses

Exercice 4: Système de télémétrie

Ecrivez les tests unitaires pour la **classe TelemetryDiagnosticControl** .

Refactorez le code autant que nécessaire pour **la classe testable** .

Exercice 4: Système de télémétrie

Ecrivez les tests unitaires pour la **classe TelemetryDiagnosticControl** .

Refactorisez le code autant que nécessaire pour **la classe testable**.

Minimisez les changements à l'API publique autant que possible.

Exercice 4: Système de télémétrie

Ecrivez les tests unitaires pour la **classe TelemetryDiagnosticControl** .

Refactorisez le code autant que nécessaire pour **la classe testable**.

Minimisez les changements à l'API publique autant que possible.

bonus:

La classe TelemetryDiagnosticControl ne suit pas un ou plusieurs principes SOLID ou objet. Donnez à chaque fois le numéro de ligne, le principe et la violation.

Single Responsibility Principle (SRP)

Une classe ne devrait avoir qu'une seule raison de changer.

Single Responsibility Principle (SRP)

Il ne devrait jamais y avoir plus d'une raison pour qu'une classe change.

Une classe ne devrait avoir qu'une et une seule responsabilité.

Interface Segregation Principle (IRP)

Un client ne devrait pas être forcé de dépendre d'une interface qu'il n'utilise pas.

Interface Segregation Principle (IRP)

Un client ne devrait pas être forcé de dépendre de membres d'interface qu'il n'utilise pas.

Des interfaces qui n'ont qu'un rôle devraient être préférées à des interfaces plus grosses.

Reference: SRP

<http://www.objectmentor.com/resources/articles/srp.pdf>

Pag. 151/152

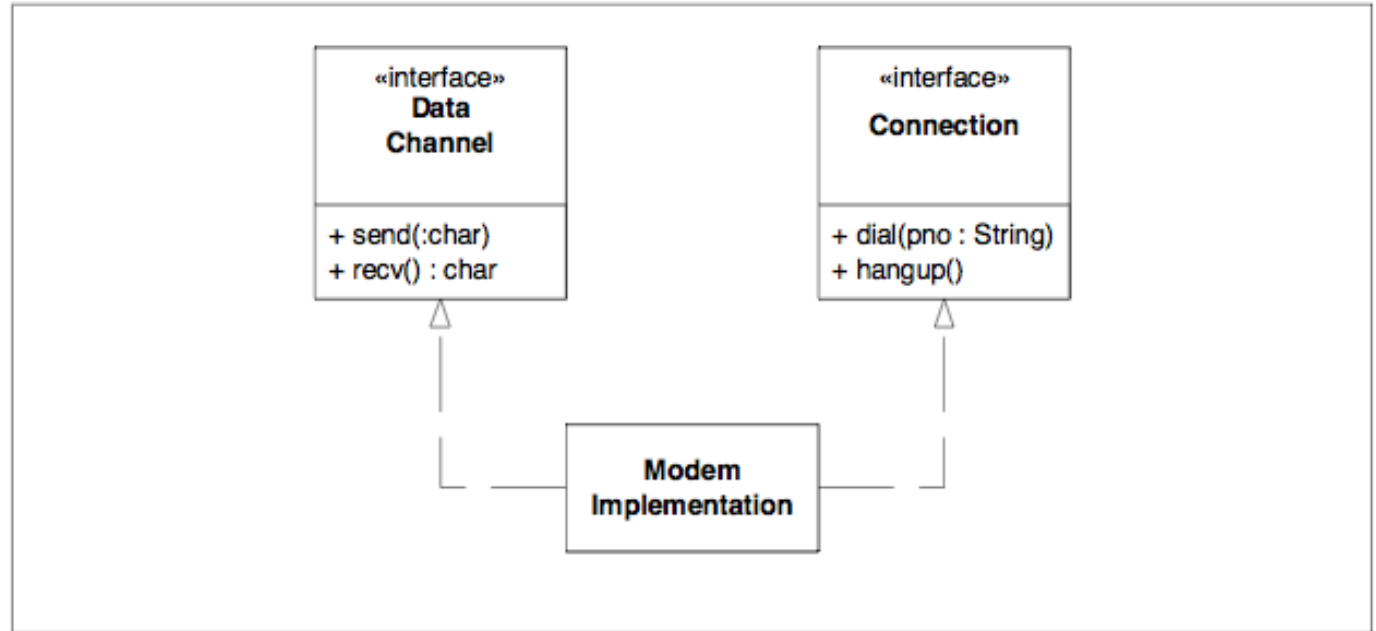


Figure 9-3
Separated Modem Interface

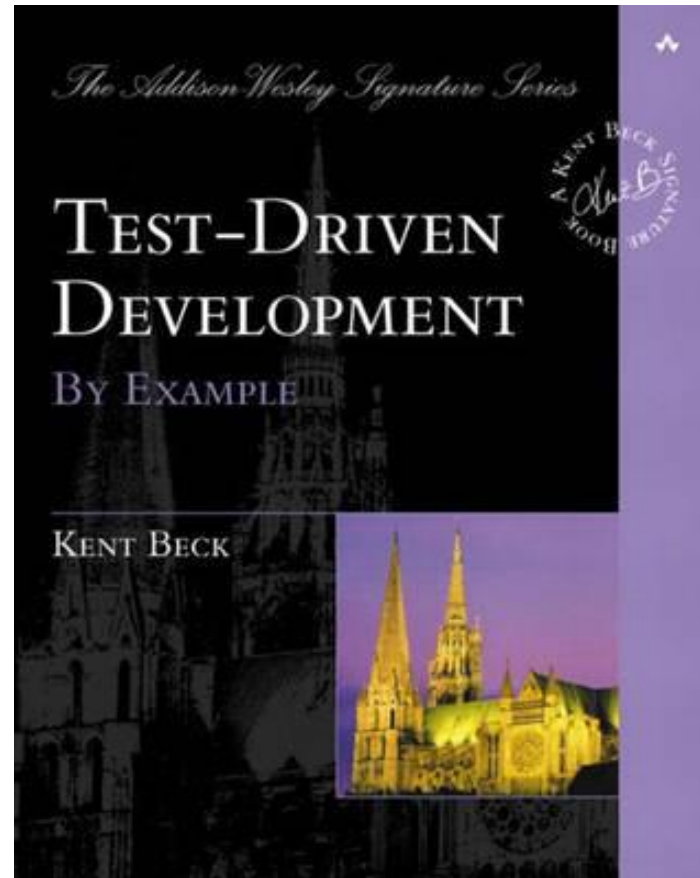
Synergie entre le test et le design

Michael Feathers:

Ecrire des tests est une autre façon de regarder le code et de le comprendre et le réutiliser, et c'est le but d'un bon design OO.

C'est la raison de la synergie profonde entre la testabilité et le bon design.

More references



More references

Endo-Testing: Unit Testing with Mock Objects

Tim Mackinnon (Connextra), Steve Freeman (BBST), Philip Craig (Independent)
(tim.mackinnon@pobox.com, steve@m3p.co.uk, philip@pobox.com)

This paper was presented at the Software Engineering - XP2004 conference, and will be published in *XP eXamined*.

Abstract

Unit testing is a fundamental 1
difficult to test in isolation. It
and difficult to maintain and i
domain code and test suites. T
structure, and avoid polluting

Keywords: Extreme Programs

1 Introduction

"Once," said the Moe

Unit testing is a fundamental principle of software development. Trivial code is difficult to test, and you want to be notified because you are trying to test.

We propose a technique called *emulated stubs* that emulates the code which they test from inside the code stubs. In other words, we write code stubs with two interfaces: one as usual, and we use our tests

Our experience is that developing a better structure of both domain and regular format that gives the domain should be written to make it easier technique to achieve this. We pay the cost of writing stub code.

In this paper, we first describe the benefits and costs of Mock brief pattern for using Mock C

2 Unit testing with Mo

An essential aspect of unit tests is that you are testing what you are testing and where any errors occur, simply and clearly as possible.

Mock Roles, not Objects

Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes
ThoughtWorks UK
Berkshire House, 168-173 High Holborn
London WC1V 7AA

{sfreeman, npryce, tmackinnon, jwalnes} @thoughtworks.com

ABSTRACT

Mock Objects is an extension to TestNG that supports good Object-Oriented design and a coherent system of types within a codebase. It is at least as interesting as a technique for isolating code under test from external dependencies as libraries than is widely thought. This paper discusses the pros and cons of using Mock Objects with an extended TestNG, and worst practices gained from experience. It also introduces jMock, a Java mocking framework, and shares our collective experience.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design; Object-Oriented design methods

General Terms

Design, Verification.

Keywords

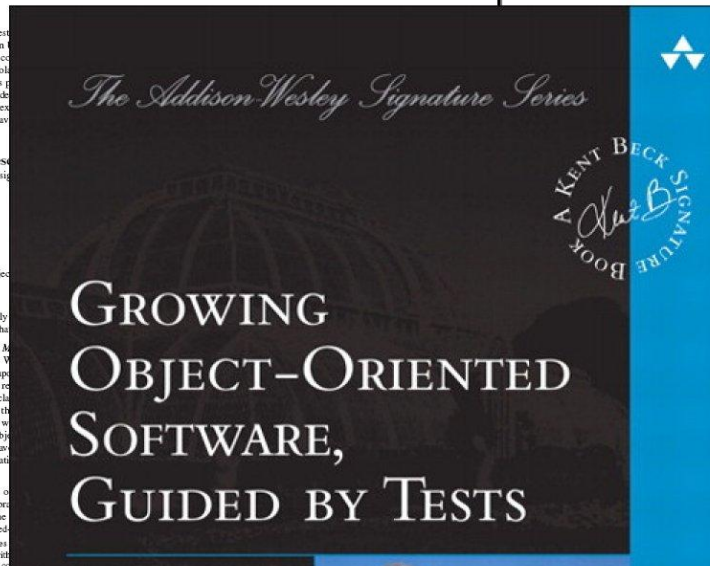
Test-Driven Development, Mock Objects

1. INTRODUCTION

Mock Objects is misnamed. It is really types in a system based on the roles that

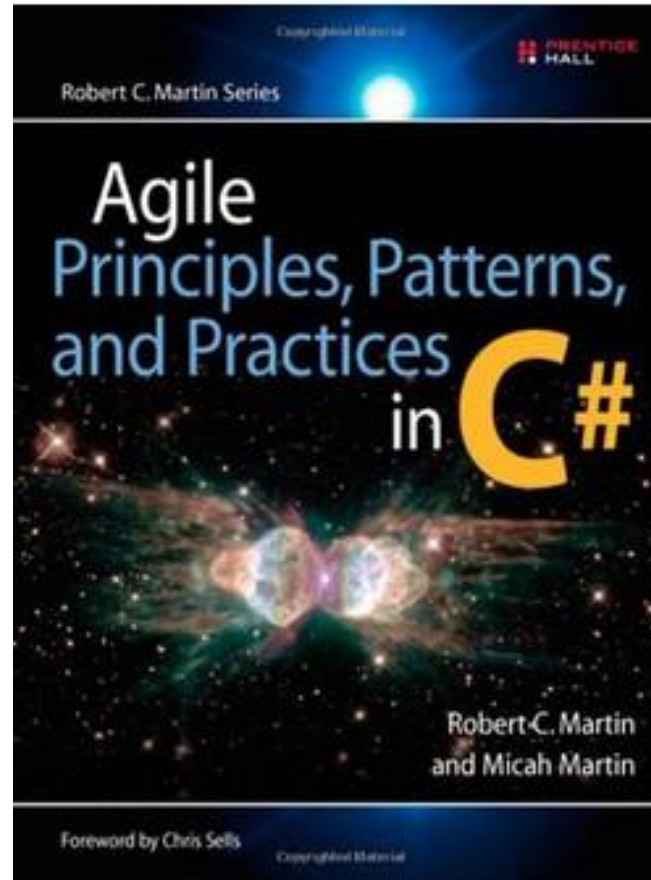
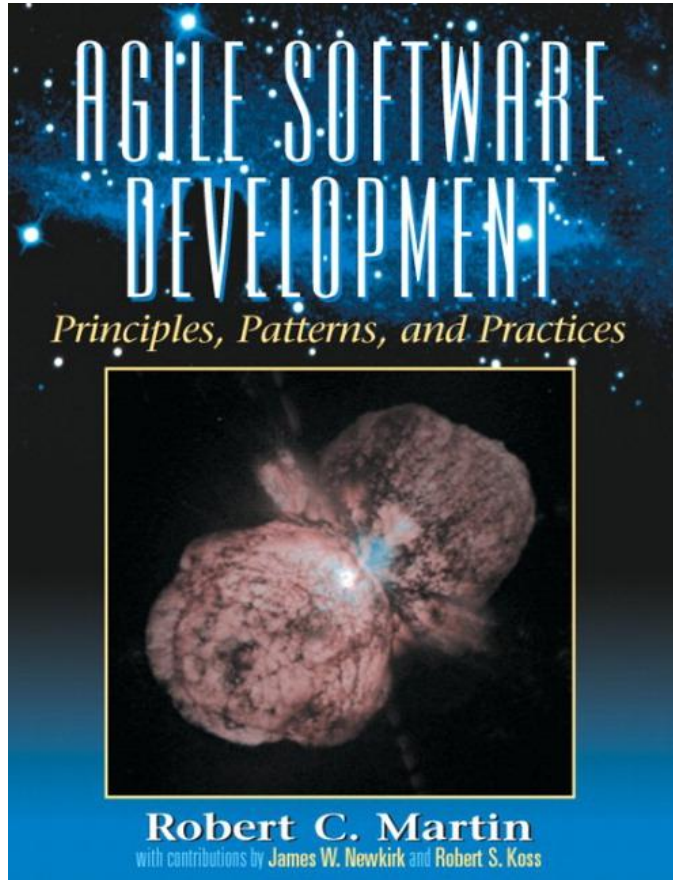
In [10] we introduced the concept of *Mock Objects* to support Test-Driven Development. We wrote better structured tests and, more importantly, we wrote code by preserving encapsulation, reusing code, clarifying the interactions between classes, and so on. Now, how we have refined and adjusted the concept of *Mock Objects* with the experience since then. In particular, we will discuss the most important benefit of *Mock Objects*, called “interface discovery”. We have developed a framework to support dynamic generation of *Mock Objects* from this experience.

The rest of this section establishes our Driven Development and good programming, and then introduces the rest of the paper introduces Needs-Driven Development.



ThoughtWorks®

More references



References

- ★ <http://scratch.mit.edu/projects/13134082/>
- ★ <http://vimeo.com/15007792>
- ★ <http://martinfowler.com/bliki/TestPyramid.html>
- ★ <http://martinfowler.com/bliki/StranglerApplication.html>
- ★ <http://www.markhneedham.com/blog/2009/07/07/domain-driven-design-anti-corruption-layer/>
- ★ <http://www.objectmentor.com/resources/articles/srp.pdf>
- ★ <http://www.objectmentor.com/resources/articles/ocp.pdf>
- ★ <http://www.objectmentor.com/resources/articles/lsp.pdf>
- ★ <http://www.objectmentor.com/resources/articles/isp.pdf>
- ★ <http://www.objectmentor.com/resources/articles/dip.pdf>

References / Links / Slides

On Twitter :

@S2IL

@LUKADOTNET