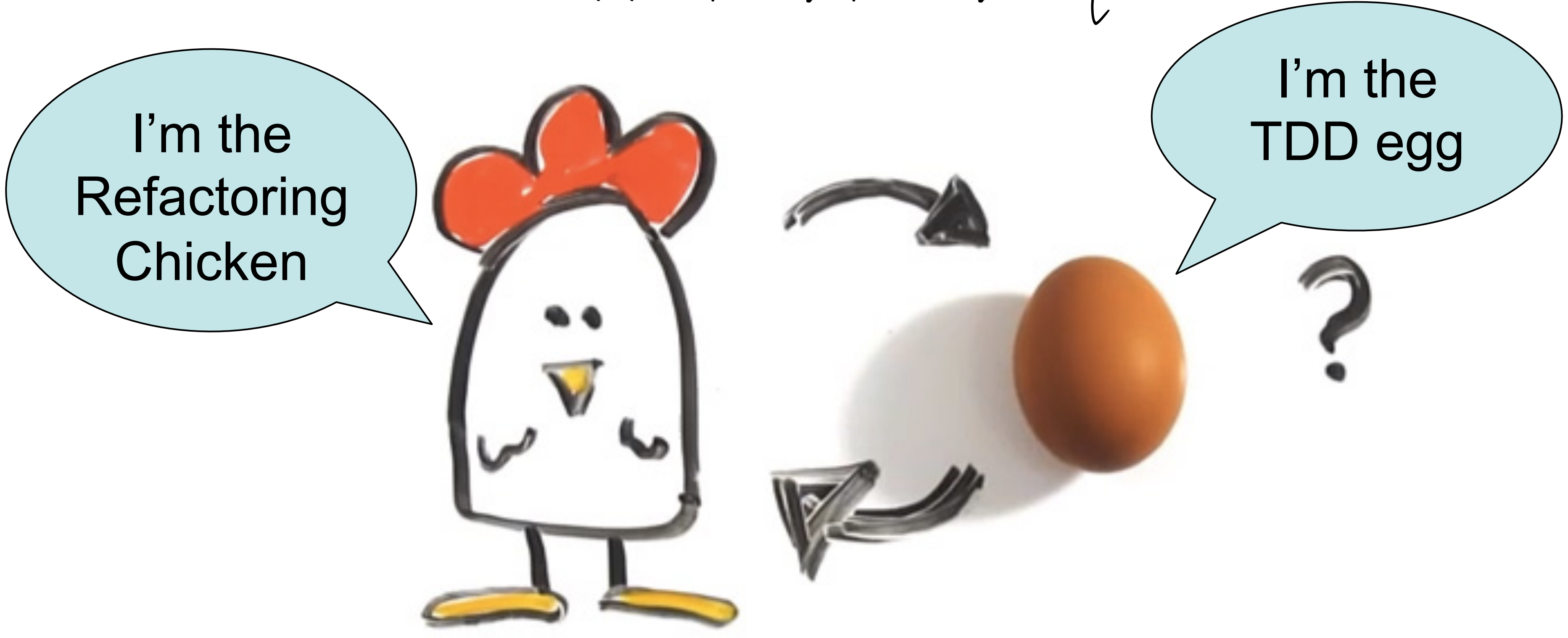
Refactoring legacy code driven by tests

Luca Minudel + Saleem Siddiqui

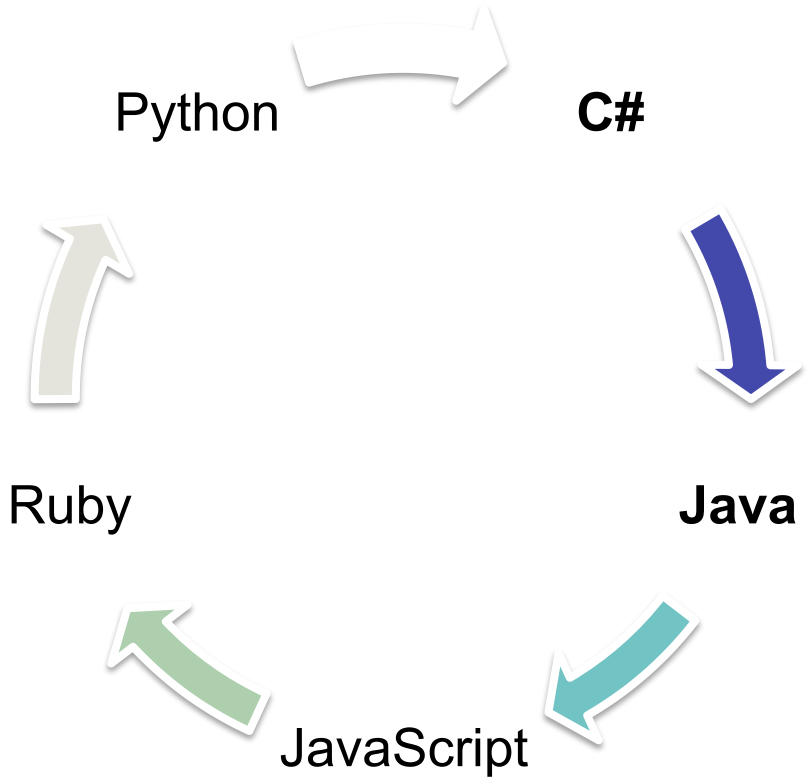


ThoughtWorks®

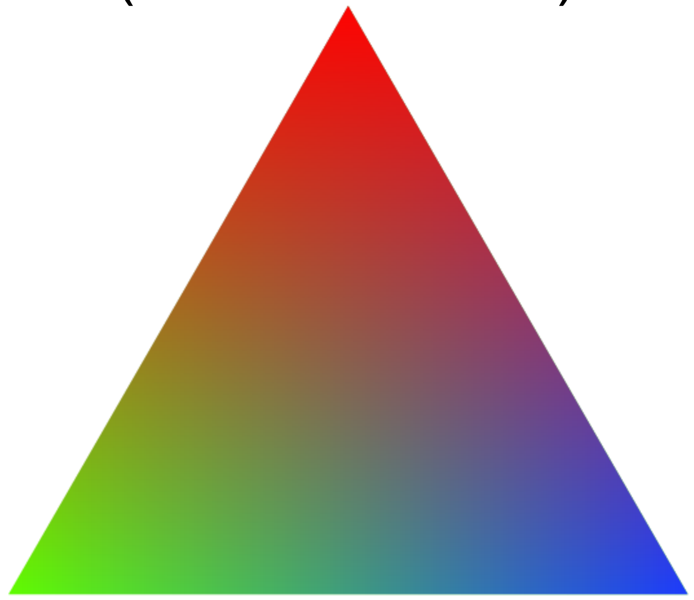
Let's clarify the scope of this Workshop

ThoughtWorks®

Languages supported in this Workshop



Automatic Testing Continuum



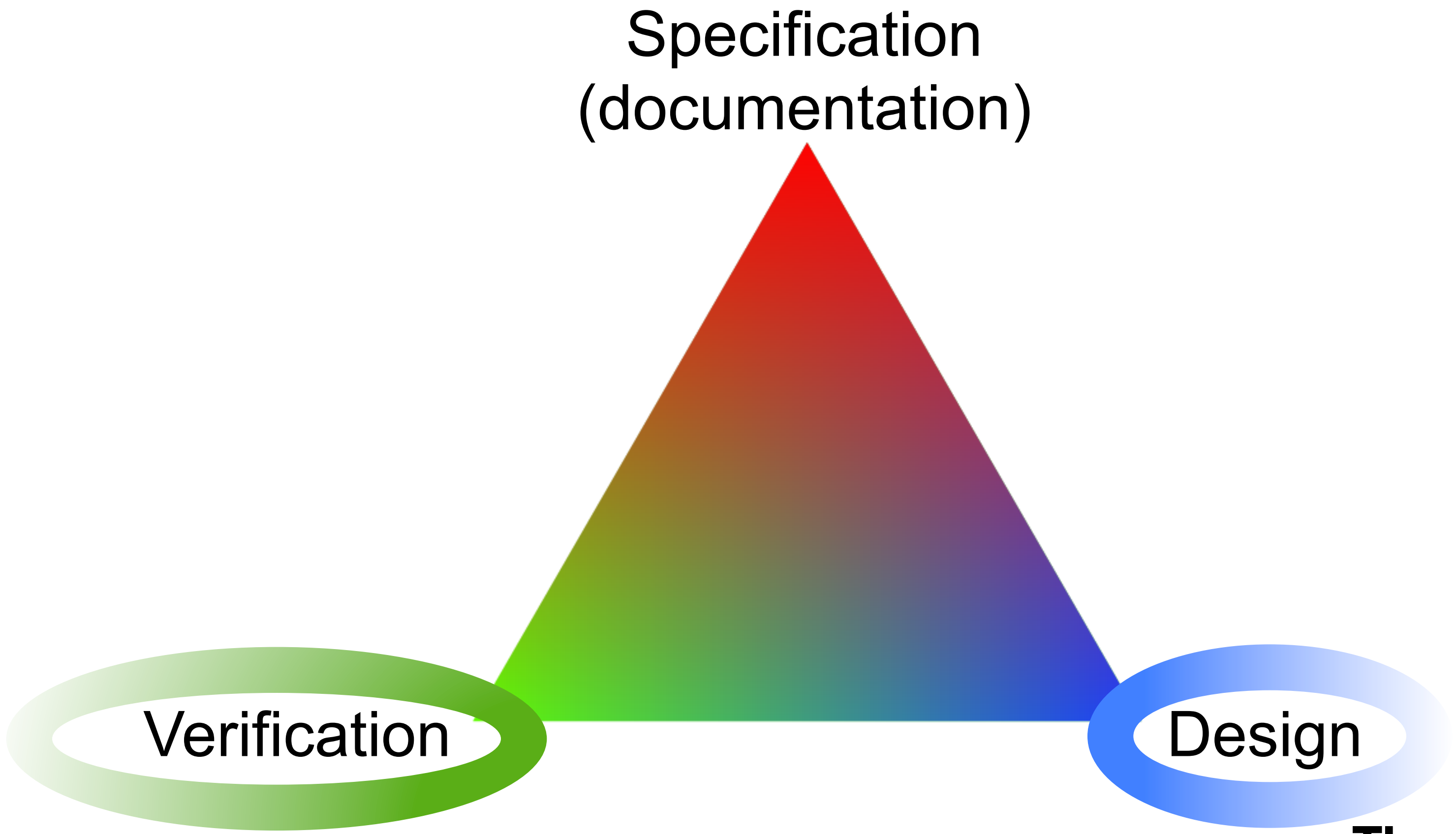
Specification
(documentation)

Verification

Design

ThoughtWorks®

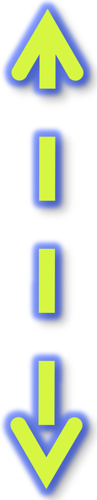
Scope of this workshop



ThoughtWorks®

Types of Automatic Tests

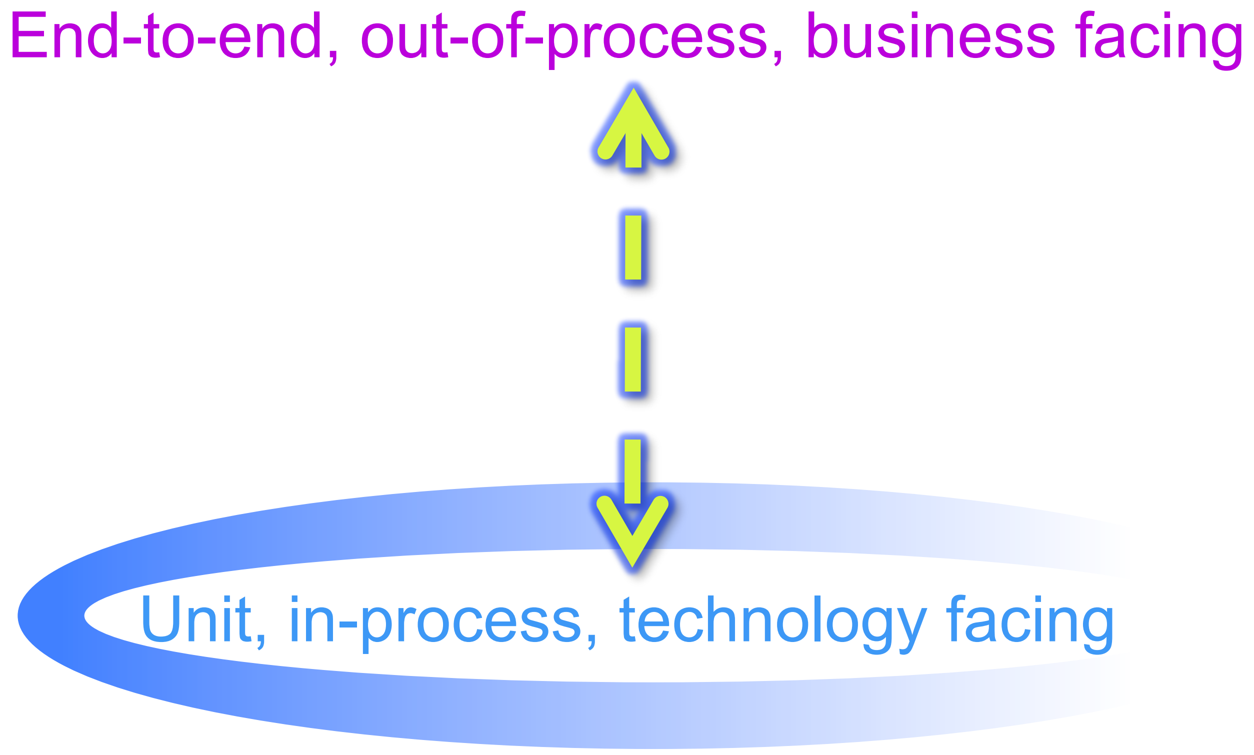
End-to-end, out-of-process, business facing



Unit, in-process, technology facing

ThoughtWorks®

Scope of this workshop


End-to-end, out-of-process, business facing
Unit, in-process, technology facing

Exercise 1: Tire Pressure Monitoring System

Alarm class:

monitors tire pressure and sets an alarm if the pressure falls outside of the expected range.

ThoughtWorks®

Exercise 1: Tire Pressure Monitoring System

Alarm class:

monitors tire pressure and sets an alarm if the pressure falls outside of the expected range.

Sensor class:

simulates the behavior of a real tire sensor, providing random but realistic values.

ThoughtWorks®

Exercise 1: Tire Pressure Monitoring System

Write the unit tests for the **Alarm class**.

Refactor the code as much as you need to **make the Alarm class testable**.

Exercise 1: Tire Pressure Monitoring System

Write the unit tests for the **Alarm class**.

Refactor the code as much as you need to **make the Alarm class testable**.

Minimize changes to the public API as much as you can.

ThoughtWorks®

Exercise 1: Tire Pressure Monitoring System

Write the unit tests for the **Alarm class**.

Refactor the code as much as you need to **make the Alarm class testable**.

Minimize changes to the public API as much as you can.

Extra credits:
Alarm class fails to follow one or more of the SOLID principles.
Write down the line number, the principle & the violation.

ThoughtWorks®

The SOLID acronym

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

ThoughtWorks®

Dependency Inversion Principle (DIP)

Martin Fowler's definition:

a) High level modules should not depend upon low level modules, both should depend upon abstractions.

b) Abstractions should not depend upon details, details should depend upon abstractions.

ThoughtWorks®

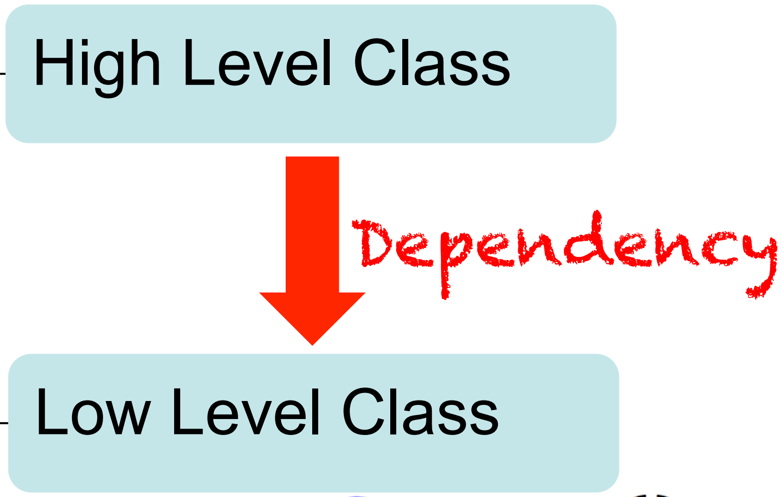
Dependency Inversion Principle (DIP)

Both low level classes and high level classes should depend on abstractions.

High level classes should not depend on low level classes.

DIP Violation In Example Code

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

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



Open Closed Principle (OCP)

Bertrand Meyer's definition:

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

ThoughtWorks®

Open Closed Principle (OCP)

Classes and methods should be
open for extensions
&
strategically closed for modification.

So that the behavior can be changed and extended adding new code instead of changing the class.

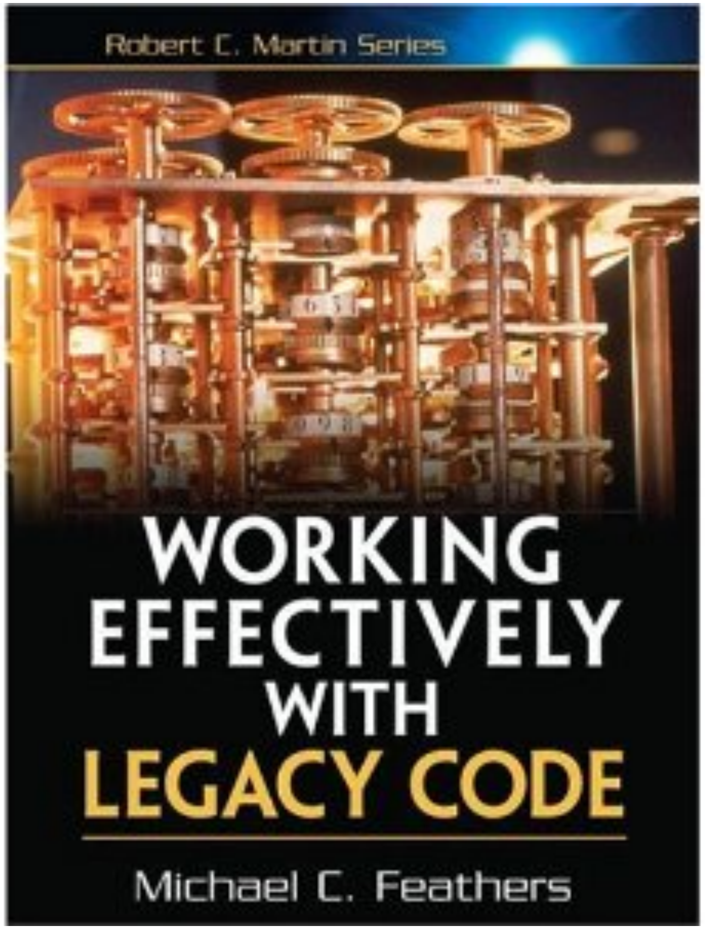
ThoughtWorks®

OCP Violation In Example Code

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

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

Want to use a new type of sensor?
Must modify code; cannot extend it

Reference: WELC



- Parametrize Constructor
- Extract Interface

ThoughtWorks®

Exercise 2: Unicode File To Htm Text Converter

UnicodeFileToHtmTextConverter class:

formats a plain text file for display in a browser.

Exercise 2: Unicode File To Htm Text Converter

Write the unit tests for the **UnicodeFileToHtmTextConverter class**.

Refactor the code as much as you need to **make the class testable**.

ThoughtWorks®

Exercise 2: Unicode File To Htm Text Converter

Write the unit tests for the **UnicodeFileToHtmTextConverter class**.

Refactor the code as much as you need to **make the class testable**.

Minimize changes to the public API as much as you can.

Exercise 2: Unicode File To Htm Text Converter

Write the unit tests for the **UnicodeFileToHtmTextConverter class**.

Refactor the code as much as you need to **make the class testable**.

Minimize changes to the public API as much as you can.

Extra credits:

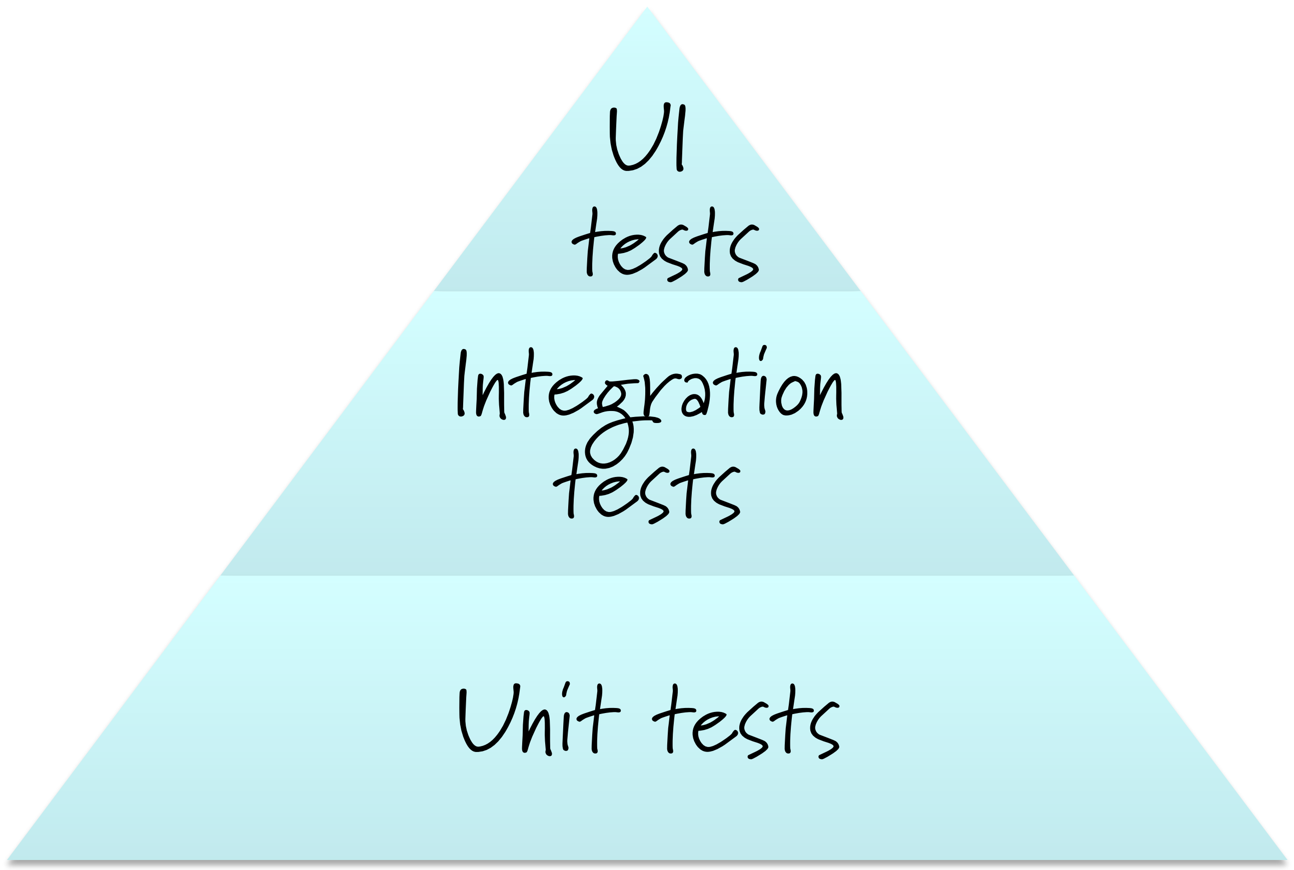
UnicodeFileToHtmTextConverter class fails to follow one or more of the SOLID principles. Write down the line number, the principle & the violation.

Feathers' rules of thumb. Extended !

A test is not a unit test when:

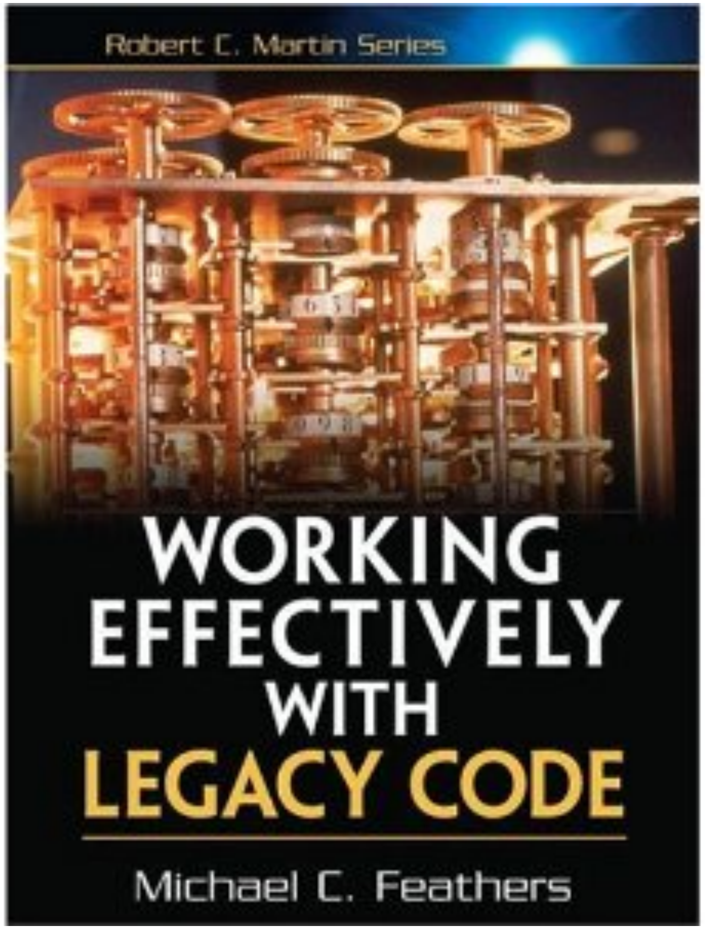
- It talks to the database
- It communicates across the network
- It touches the file system or reads config info
- It uses DateTime.now() or Random
- It depends on non-deterministic behavior
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it.

Mike Cohn's Test Pyramid. Explained !



ThoughtWorks®

Reference: WELC



- Parametrize Constructor
- Extract Interface
- Skin and Wrap the API

Refactoring and TDD

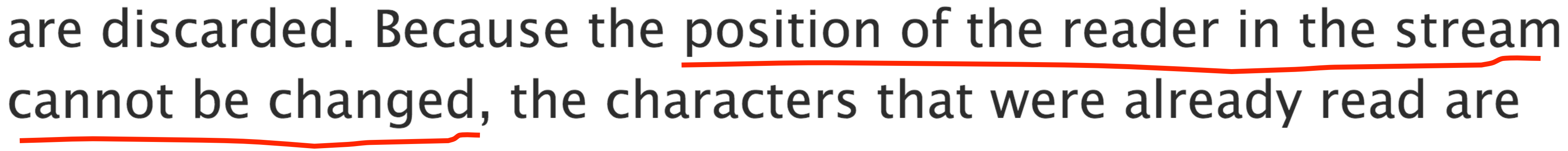
```
public string ConvertToHtml()
{
    using (TextReader unicodeFileStream = File.OpenText(_fullFilenameWithPath))
    {
        string html = string.Empty;

        string line = unicodeFileStream.ReadLine();

        // ... conversion details omitted

        return html;
    }
}
```

Should we inject this dependency?

ThoughtWorks®

Behavior of TextReader

TextReader documentation from MSDN


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

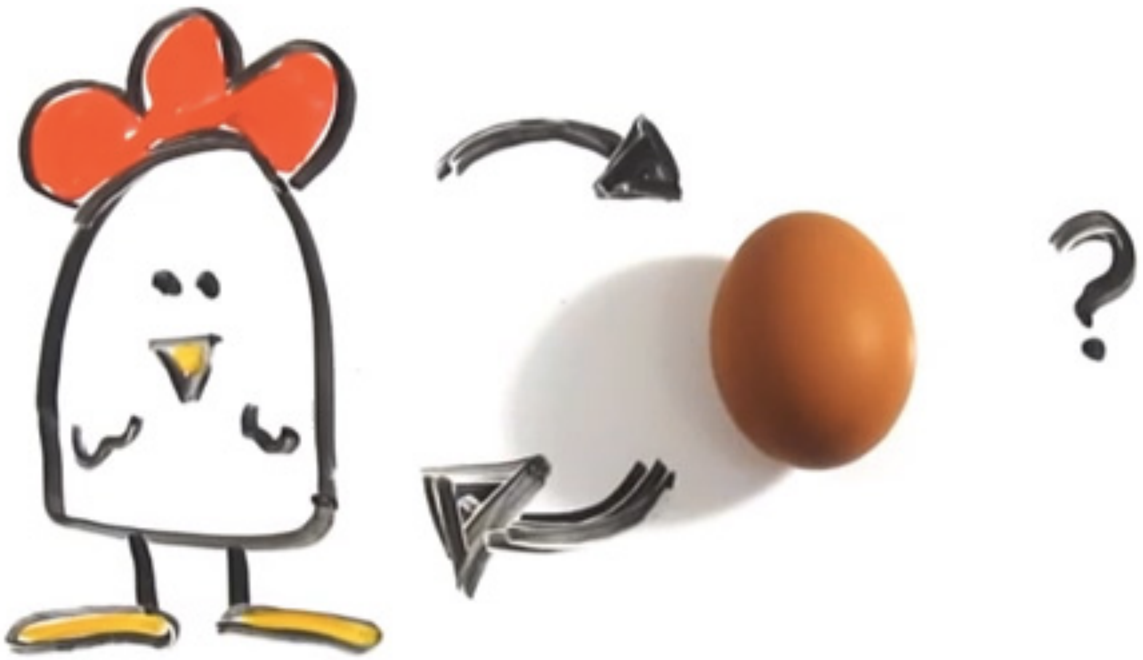
Non-idempotent behavior

ThoughtWorks®

```
public UnicodeFileToHtmTextConverter(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;
    }
}
```

Dependency injection and idempotent behavior

Refactoring and TDD



Exercise 3: Ticket Dispenser

TicketDispenser class:

manages a queuing system in a shop.

There may be more than one ticket dispenser but the same ticket should not be issued to two different customers.

Exercise 3: Ticket Dispenser

TurnTicket class:

represent the ticket with the turn number.

TurnNumberSequence class:

returns the sequence of turn numbers.

Exercise 3: Ticket Dispenser

Write the unit tests for the **TicketDispenser class**.

Refactor the code as much as you need to **make the TicketDispenser class testable**.

Exercise 3: Ticket Dispenser

Write the unit tests for the **TicketDispenser class**.

Refactor the code as much as you need to **make the TicketDispenser class testable**.

Minimize changes to the public API as much as you can.

ThoughtWorks®

Exercise 3: Ticket Dispenser

Write the unit tests for the **TicketDispenser class**.

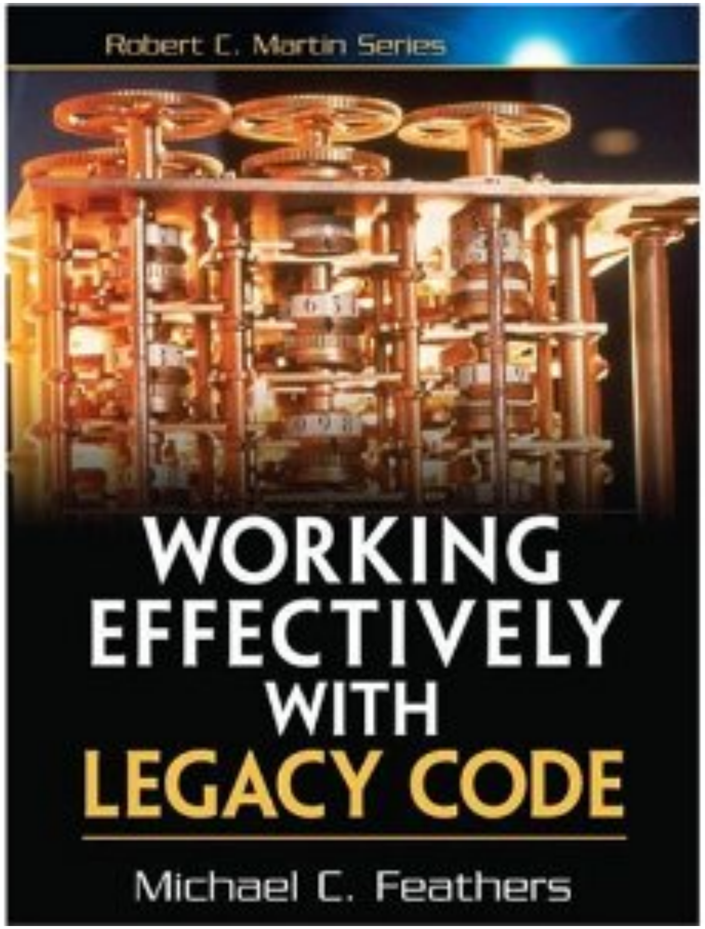
Refactor the code as much as you need to **make the TicketDispenser class testable**.

Minimize changes to the public API as much as you can.

Extra credits:
TicketDispenser class fails to follow one or more of the OO and SOLID principles. Write down the line number, the principle & the violation.

ThoughtWorks®

Reference: WELC



- Parametrize Constructor
- Extract Interface
- Skin and Wrap the API
- Introduce Instance Delegator
- …

Exercise 4: Telemetry System

TelemetryDiagnosticControl class:

establishes a connection to the telemetry server through the TelemetryClient,
sends a diagnostic request and receives the response with diagnostic info.

TelemetryClient class:

simulates the communication with the Telemetry Server, sends requests and then receives and returns the responses

Exercise 4: Telemetry System

Write the unit tests for the **TelemetryDiagnosticControl class**.

Refactor the code as much as you need to **make the class testable**.

ThoughtWorks®

Exercise 4: Telemetry System

Write the unit tests for the **TelemetryDiagnosticControl class**.

Refactor the code as much as you need to **make the class testable**.

Minimize changes to the public API as much as you can.

ThoughtWorks®

Exercise 4: Telemetry System

Write the unit tests for the **TelemetryDiagnosticControl class**.

Refactor the code as much as you need to **make the class testable**.

Minimize changes to the public API as much as you can.

Extra credits:
TelemetryClient class fails to follow one or more of the OO and SOLID principles. Write down the line number, the principle & the violation.

ThoughtWorks®

Single Responsibility Principle (SRP)

A class should have only one reason to change.

Single Responsibility Principle (SRP)

There should never be more than one reason for a class to change.

A class should have one and only one responsibility.

Interface Segregation Principle (IRP)

Clients should not be forced to depend upon interfaces that they do not use.

ThoughtWorks®

Interface Segregation Principle (IRP)

Clients should not be forced to depend upon interface members that they don't use.

Interfaces that serve only one scope should be preferred over fat interfaces.

ThoughtWorks®

Reference: SRP

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

Pag. 151/152

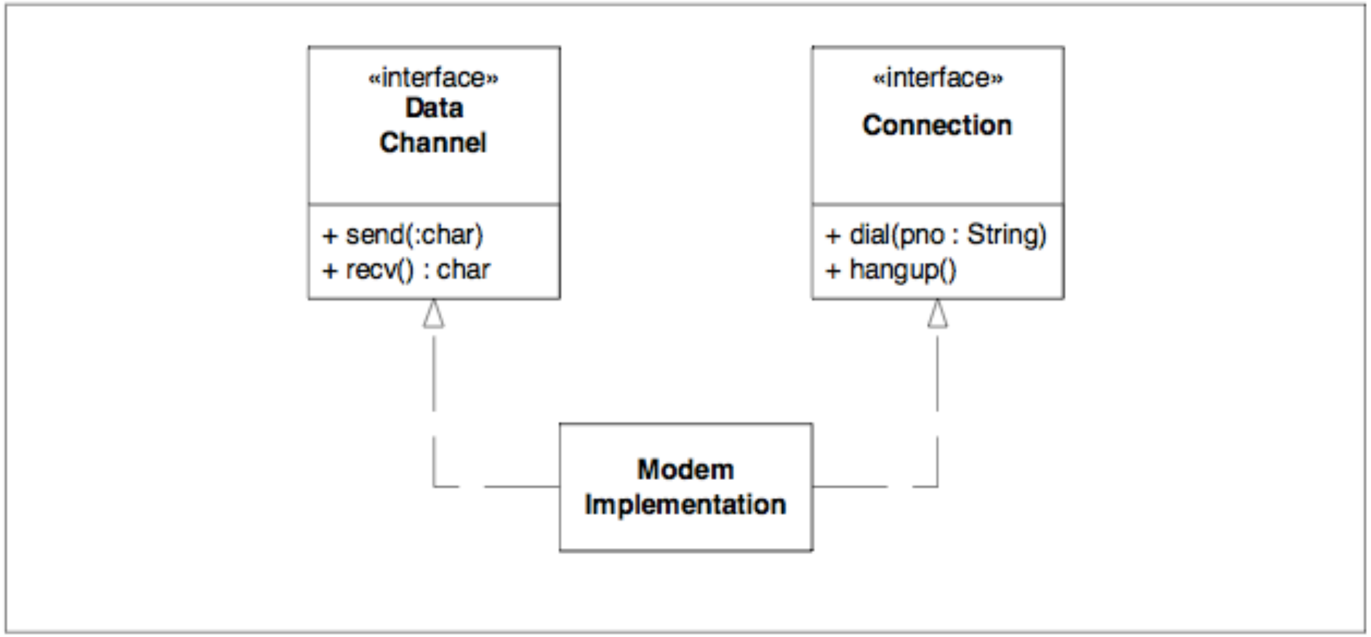


Figure 9-3
Separated Modem Interface

Synergy between testing and design

Michael Feathers:

writing tests is another way to look the code and
locally understand it and reuse it,
and that is the same goal of good OO design.

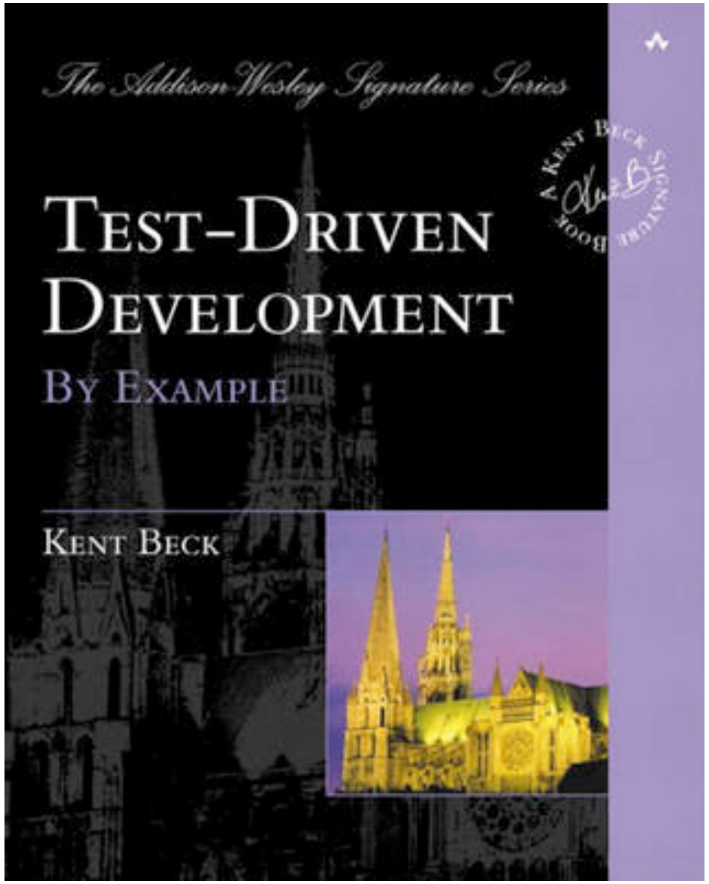
This is the reason for
the deep synergy
between testability and good design.

Other testing strategies for legacy code

- Start with other tests
- Use an automatic refactoring tool
- Strangler pattern
- DDD anti-corruption layer

ThoughtWorks®

More references


The Addison-Wesley Signature Series
A Kent Beck Signature Book
Test-Driven Development
By Example
Kent Beck

ThoughtWorks®

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 th
Software Engineering - XP200
be published in *XP eXamined*

Abstract

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

Keywords: Extreme Programm

1 Introduction

"Once," said the Moc
(A

Unit testing is a fundamental p
trivial code is difficult to test
time, and you want to be notif
because you are trying to test

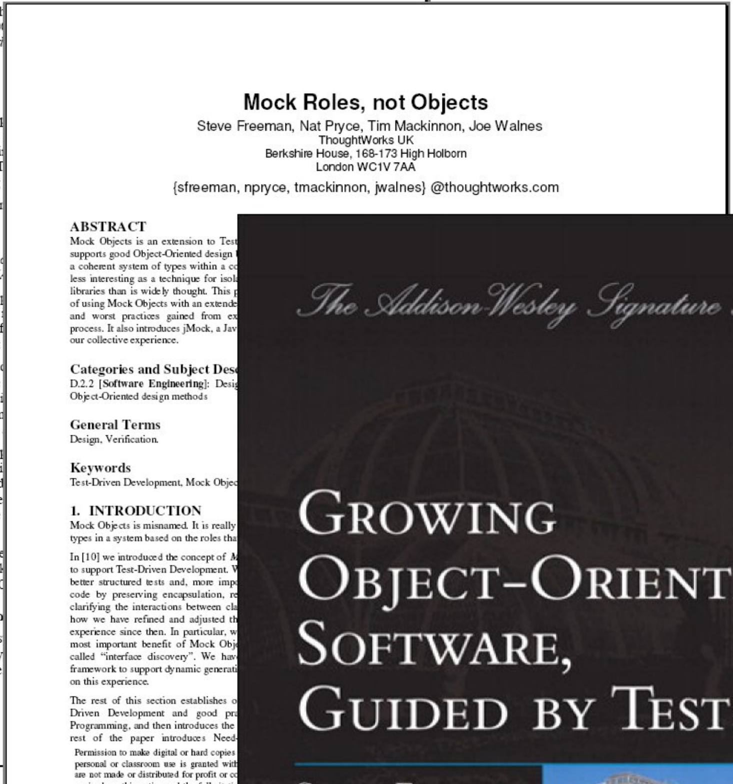
We propose a technique called
implementations that emulate
code which they test from insi
writing code stubs with two in
is usual, and we use our tests

Our experience is that develop
better structure of both domai
regular format that gives the d
should be written to make it e
technique to achieve this. We
cost of writing stub code.

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

2 Unit testing with Mo

An essential aspect of unit tes
you are testing and where any
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 Test
supports good Object-Oriented design
a coherent system of types within a co
less interesting as a technique for isola
libraries than is widely thought. This p
of using Mock Objects with an extende
and worst practices gained from ex
process. It also introduces jMock, a Jav
our collective experience.

Categories and Subject Desc
D.2.2 [Software Engineering]: Desig
Object-Oriented design methods

General Terms
Design, Verification.

Keywords
Test-Driven Development, Mock Objec

1. INTRODUCTION

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

In [10] we introduced the concept of *M*
to support Test-Driven Development. W
better structured tests and, more impo
code by preserving encapsulation, re
clarifying the interactions between cla
how we have refined and adjusted th
experience since then. In particular, w
most important benefit of Mock Obje
called "interface discovery". We hav
framework to support dynamic generati
on this experience.

The rest of this section establishes o
Driven Development and good pra
Programming, and then introduces the
rest of the paper introduces Need-

Permission to make digital or hard copies
personal or classroom use is granted with
are not made or distributed for profit or co

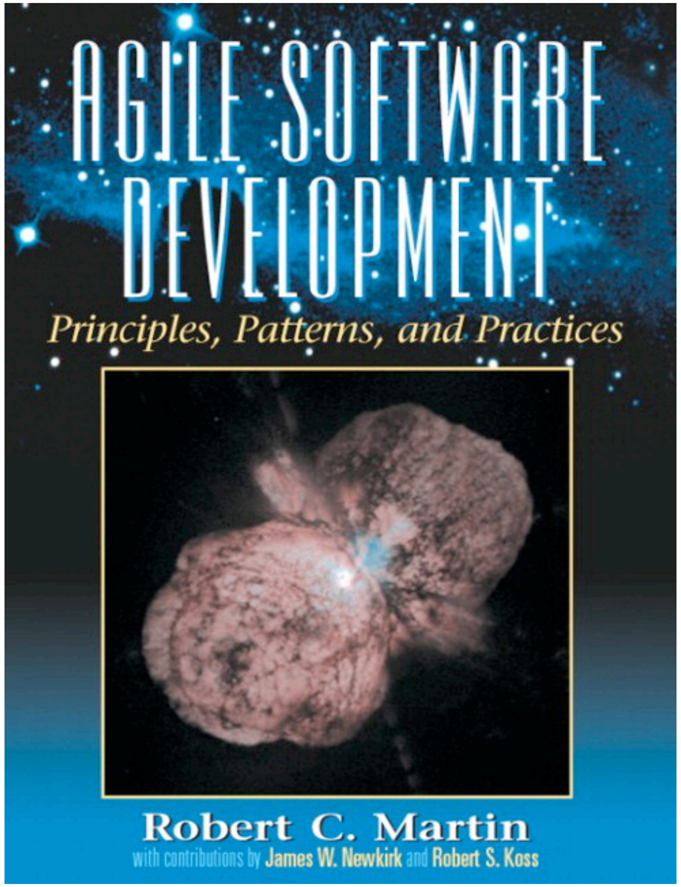
The Addison-Wesley Signature Series

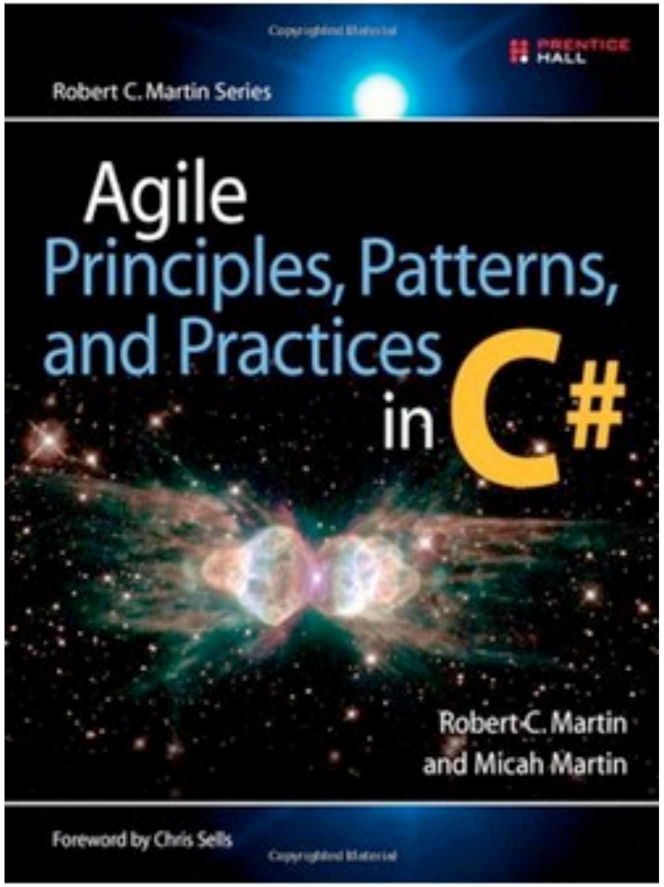
A Kent Beck Signature Book

GROWING OBJECT-ORIENTED SOFTWARE, GUIDED BY TESTS

ThoughtWorks®

More references


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss


Robert C. Martin Series
PRENTICE HALL
Agile Principles, Patterns, and Practices in C#
Robert C. Martin and Micah Martin
Foreword by Chris Sells

ThoughtWorks®

References

- http://www.youtube.com/watch?v=1a8pI65emDE
- 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

How can we help?

Emergent learning & Workshops

Innovative Software delivery

Tools

Contact us

www.thoughtworks.com

ThoughtWorks®

References / Links / Slides

On Twitter :

@S2IL

@LUKADOTNET

ThoughtWorks®