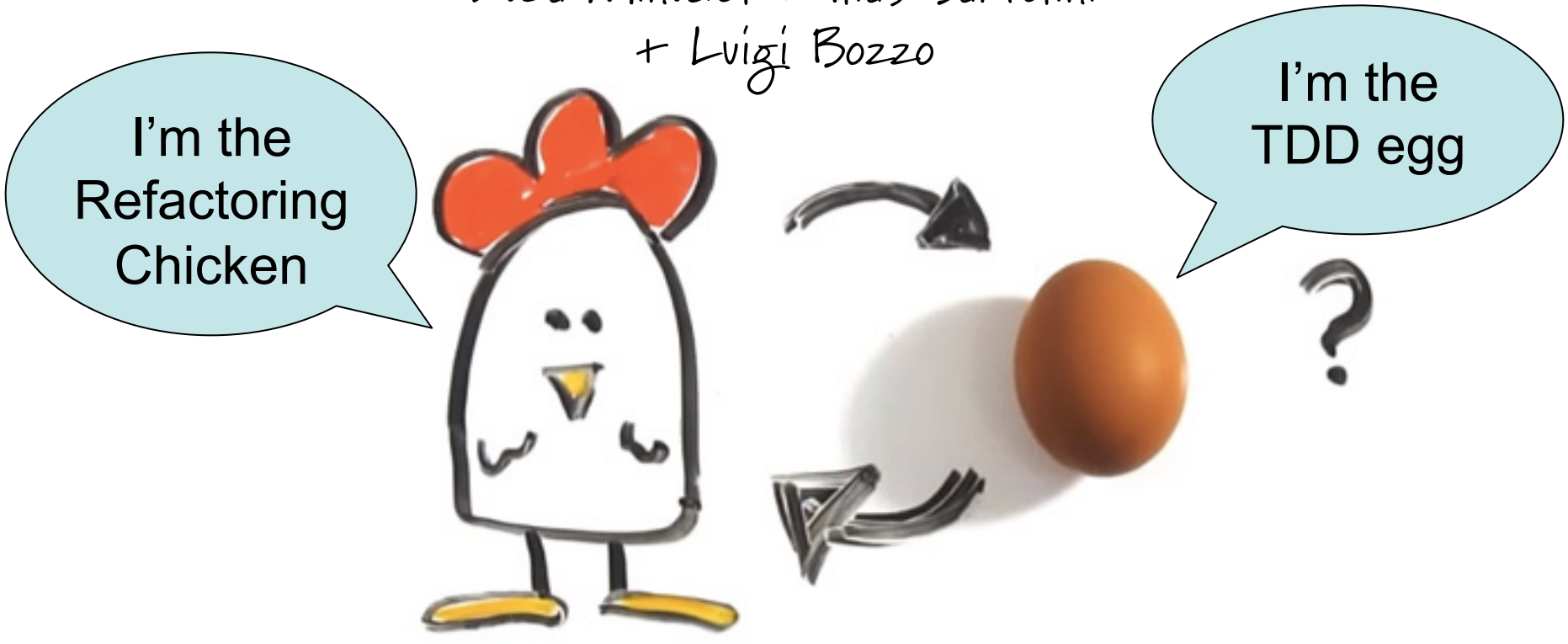
Refactoring legacy code driven by tests

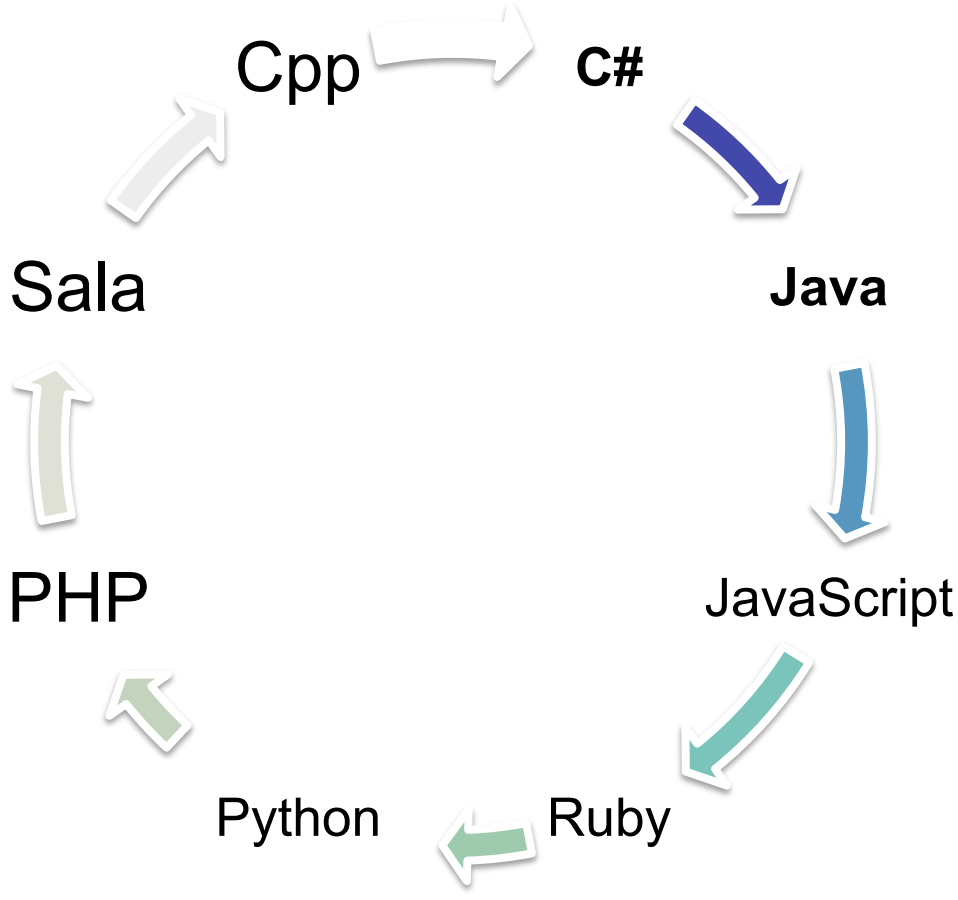
Luca Minudel + Ilias Bartolini
+ Luigi Bozzo



ThoughtWorks®

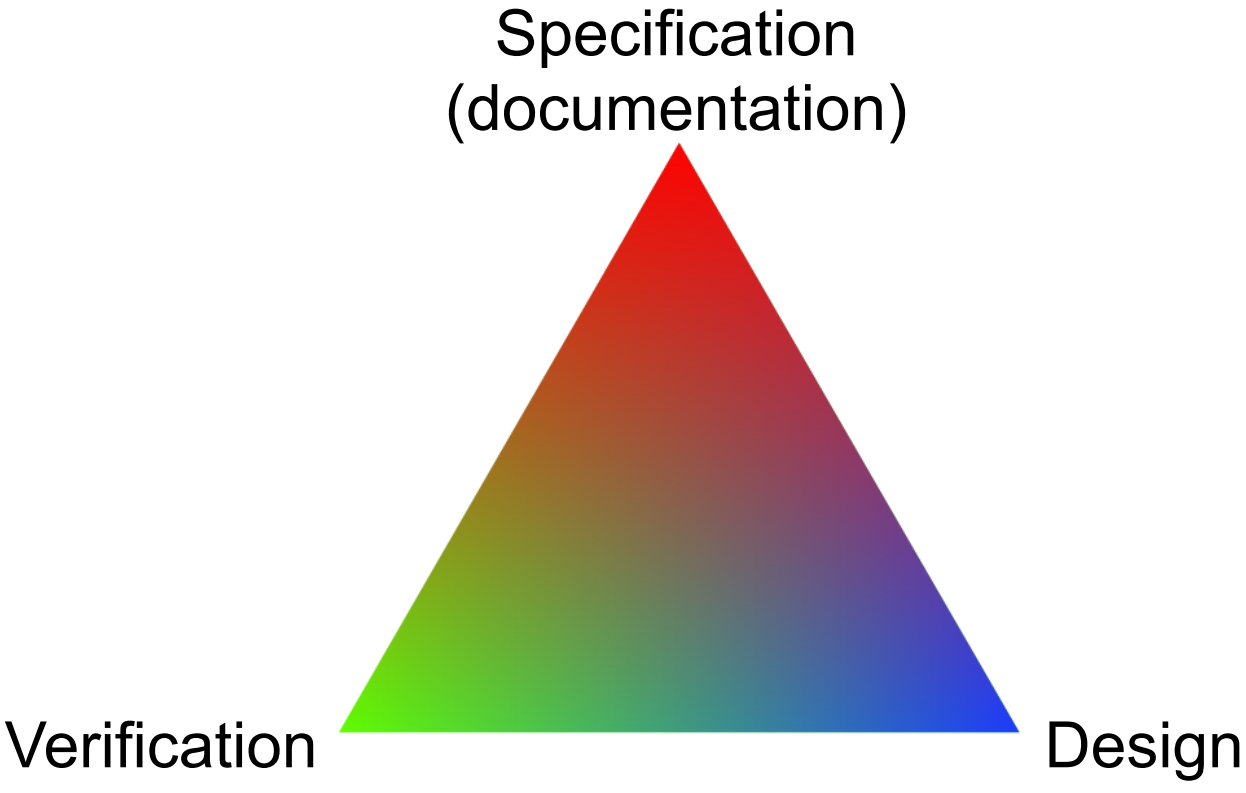
Let's clarify the scope of this Workshop

Languages supported in this Workshop

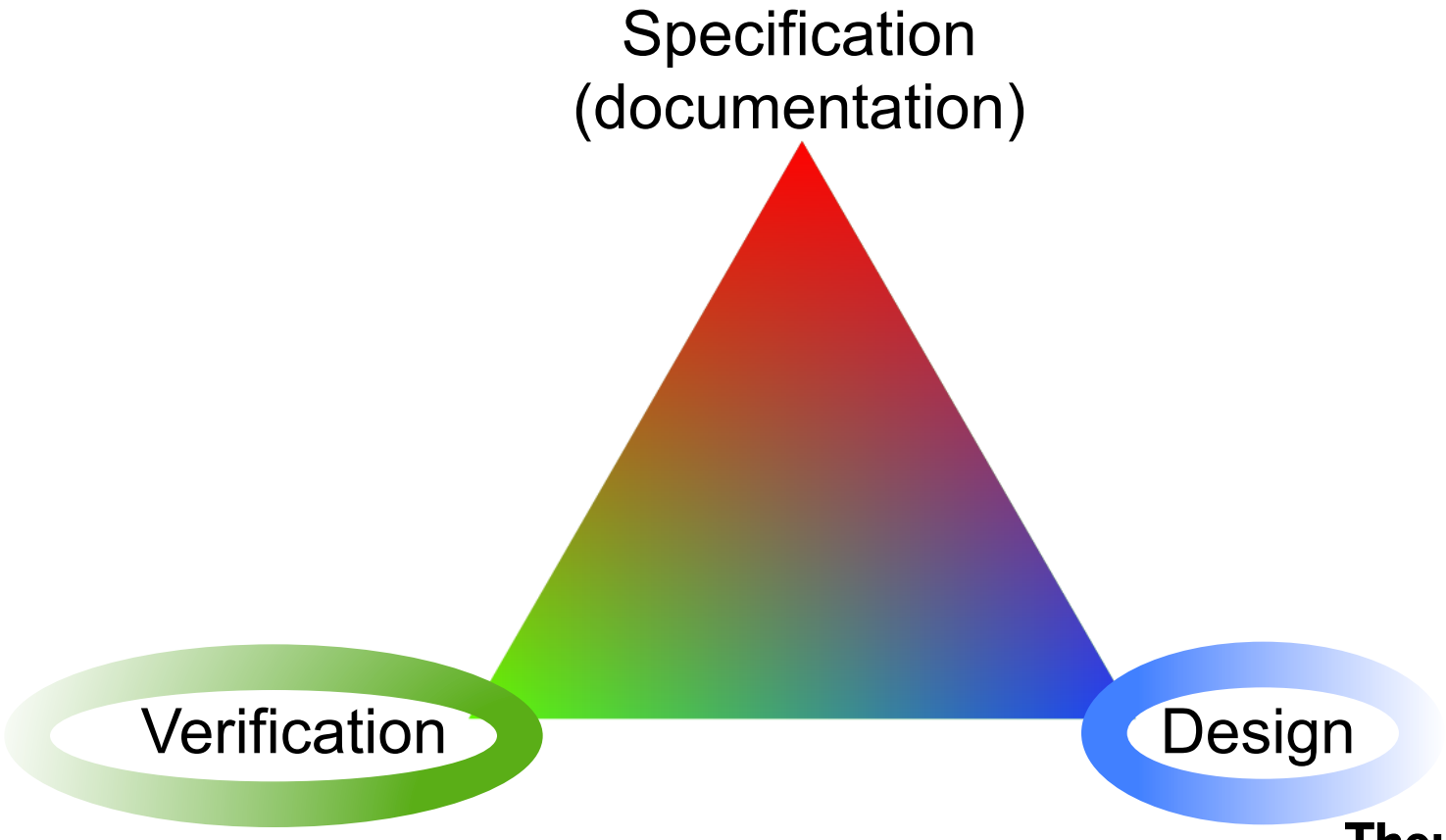


ThoughtWorks®

Automatic Testing Continuum



Scope of this workshop

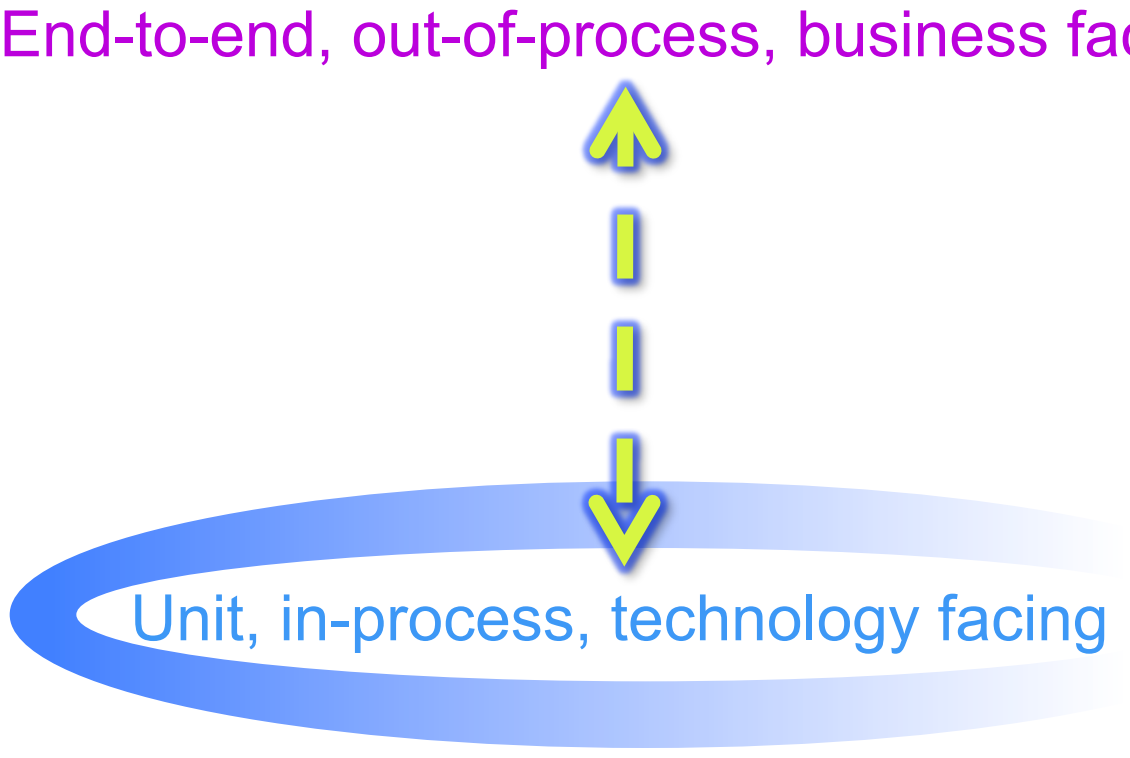


Types of Automatic Tests

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

Unit, in-process, technology facing

Scope of this workshop



Exercise 1: Tire Pressure Monitoring System

Alarm class:

controlla la pressione di un pneumatico e lancia un allarme se la pressione esce dall'intervallo di valori attesi.

Exercise 1: Tire Pressure Monitoring System

Alarm class:

controlla la pressione di un pneumatico e lancia un allarme se la pressione esce dall'intervallo di valori attesi.

Sensor class:

simula un vero sensore di pressione in un pneumatico, generando valori casuali ma realistici di pressione.

Exercise 1: Tire Pressure Monitoring System

Scrivi gli unit test per la **Alarm class**.

Fai il Refactoring del codice sino a rendere **la Alarm class testabile**.

Exercise 1: Tire Pressure Monitoring System

Scrivi gli unit test per la **Alarm class**.

Fai il Refactoring del codice sino a rendere **la Alarm class testabile**.

Minimizza i cambiamenti alla API pubblica più che puoi.

Exercise 1: Tire Pressure Monitoring System

Scrivi gli unit test per la **Alarm class**.

Fai il Refactoring del codice sino a rendere **la Alarm class testabile**.

Minimizza i cambiamenti alla API pubblica piu che puoi.

Extra credits:
Alarm class viola alcuni dei principi SOLID. Prendi nota della linea di codice e il principio violato.

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)

Martin Fowler's definition:

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

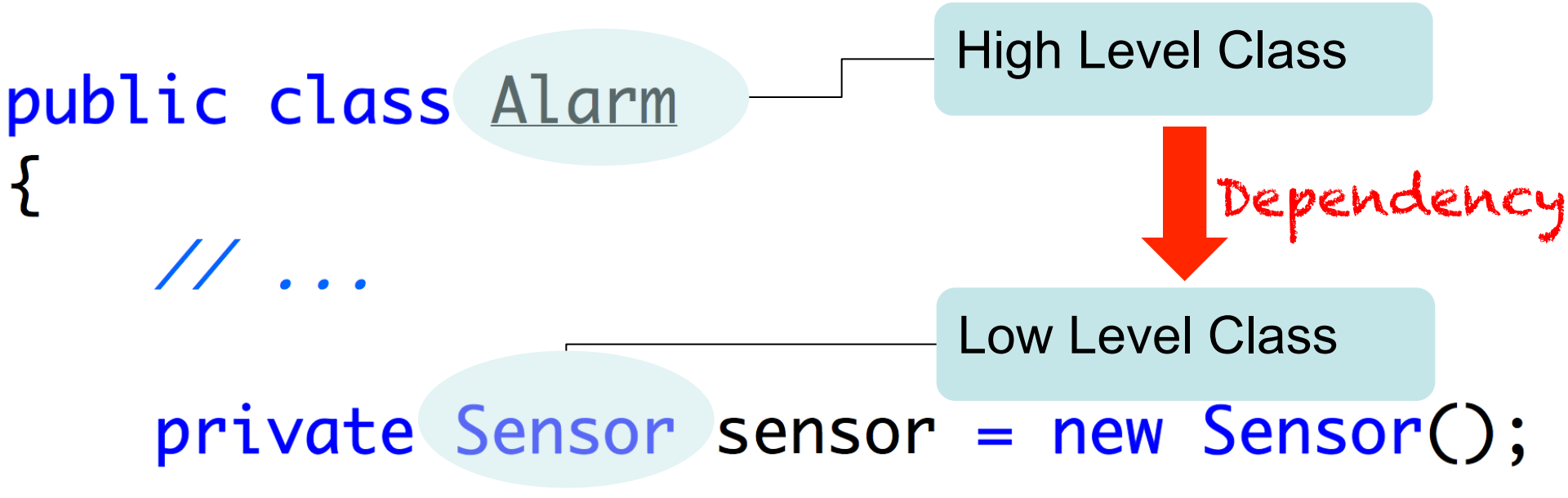
b) Abstractions should not depend upon details, details should depend upon abstractions.

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



Open Closed Principle (OCP)

Bertrand Meyer's definition:

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

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.

OCP Violation In Example Code

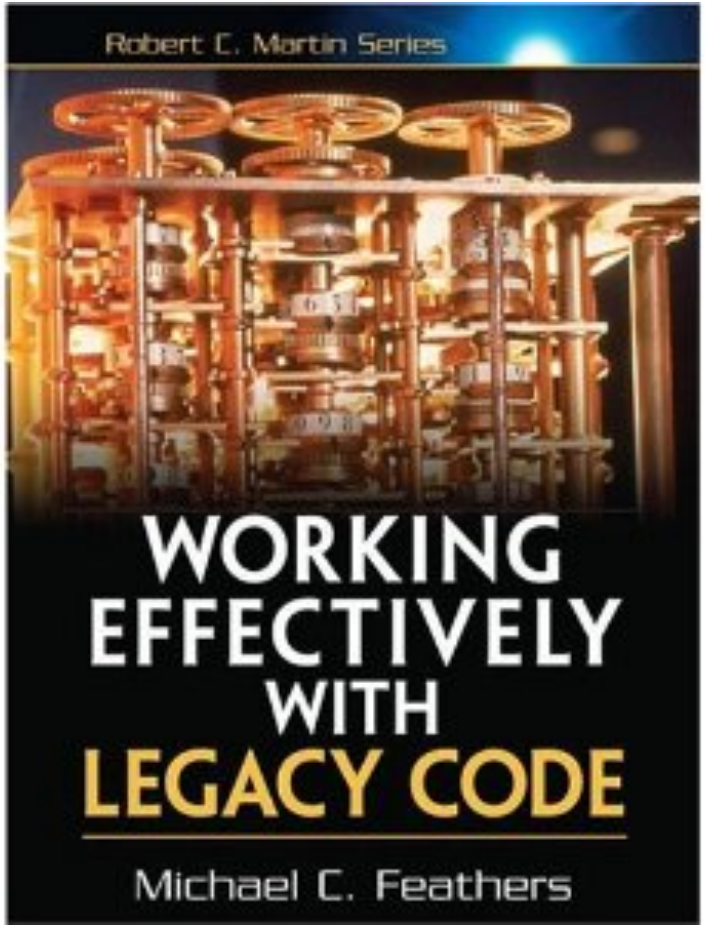
```
public class Alarm
{
    // ...

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

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

```
public Alarm() : this(new Sensor())
{
}

public Alarm(ISensor sensor)
{
    _sensor = sensor;
}

public void Check()
{
    double psiPressureValue = _sensor.PopNextPressurePsiValue();
```

Dependency injection

Reference: WELC



- Parametrize Constructor
- Extract Interface

ThoughtWorks®

Exercise 2: Unicode File To Htm Text Converter

UnicodeFileToHtmTextConverter class:

trasforma del testo semplice in html per essere visualizzato da un browser.

Exercise 2: Unicode File To Htm Text Converter

Scrivi gli unit test per la **UnicodeFileToHtmTextConverter**.

Fai il Refactoring del codice sino a rendere **la classe testabile**.

Exercise 2: Unicode File To Htm Text Converter

Scrivi gli unit test per la **UnicodeFileToHtmTextConverter**.

Fai il Refactoring del codice sino a rendere **la classe testabile**.

Minimizza i cambiamenti alla API pubblica più che puoi.

Exercise 2: Unicode File To Htm Text Converter

Scrivi gli unit test per la **UnicodeFileToHtmTextConverter**.

Fai il Refactoring del codice sino a rendere **la classe testabile**.

Minimizza i cambiamenti alla API pubblica più che puoi.

Extra credits:
UnicodeFileToHtmTextConverter class viola alcuni dei principi SOLID. Prendi nota della linea di codice e il principio violato.

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.

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?

Behavior of TextReader

TextReader documentation from MSDN

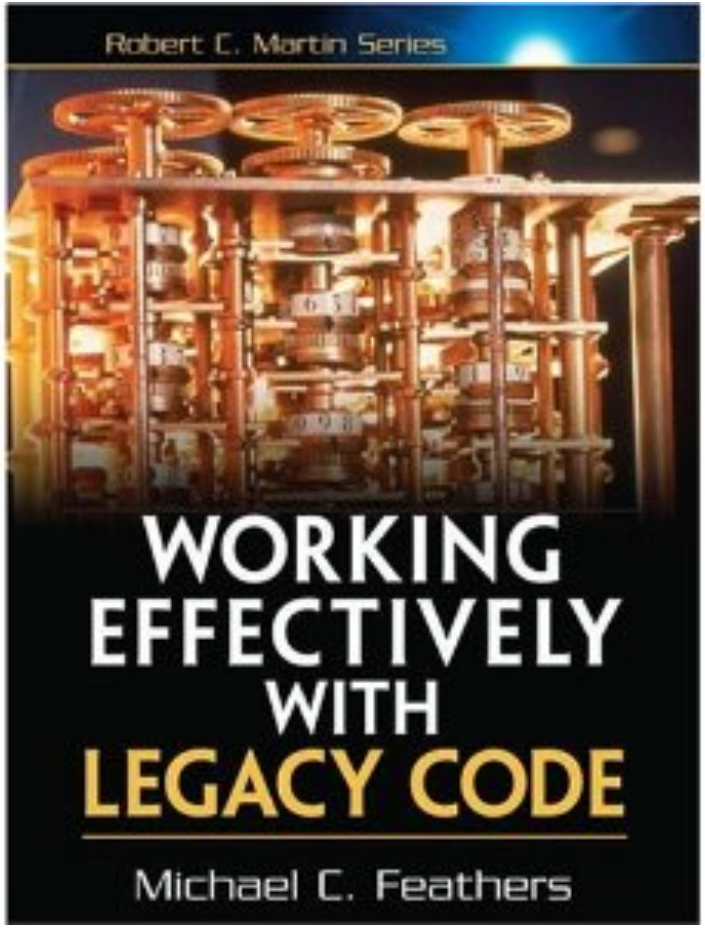
are discarded. Because the position of the reader in the stream cannot be changed, the characters that were already read are

Non-idempotent behavior

```
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

Reference: WELC



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

ThoughtWorks®

Exercise 3: Ticket Dispenser

TicketDispenser class:

gestisce il sistema delle code agli sportelli di un negozio.

Ci possono essere più distributori dei biglietti col numero del turno senza che questi distribuiscano lo stesso numero a due clienti diversi.

Exercise 3: Ticket Dispenser

TurnTicket class:

rappresenta il biglietto col numero del turno.

TurnNumberSequence class:

genera la sequenza dei numeri del turno.

Exercise 3: Ticket Dispenser

Scrivi gli unit test per la **TicketDispenser class**.

Fai il Refactoring del codice sino a rendere **la classe TicketDispenser testabile**.

Exercise 3: Ticket Dispenser

Scrivi gli unit test per la **TicketDispenser class**.

Fai il Refactoring del codice sino a rendere **la classe TicketDispenser testabile**.

Minimizza i cambiamenti alla API pubblica più che puoi.

Exercise 3: Ticket Dispenser

Scrivi gli unit test per la **TicketDispenser class**.

Fai il Refactoring del codice sino a rendere **la classe TicketDispenser testabile**.

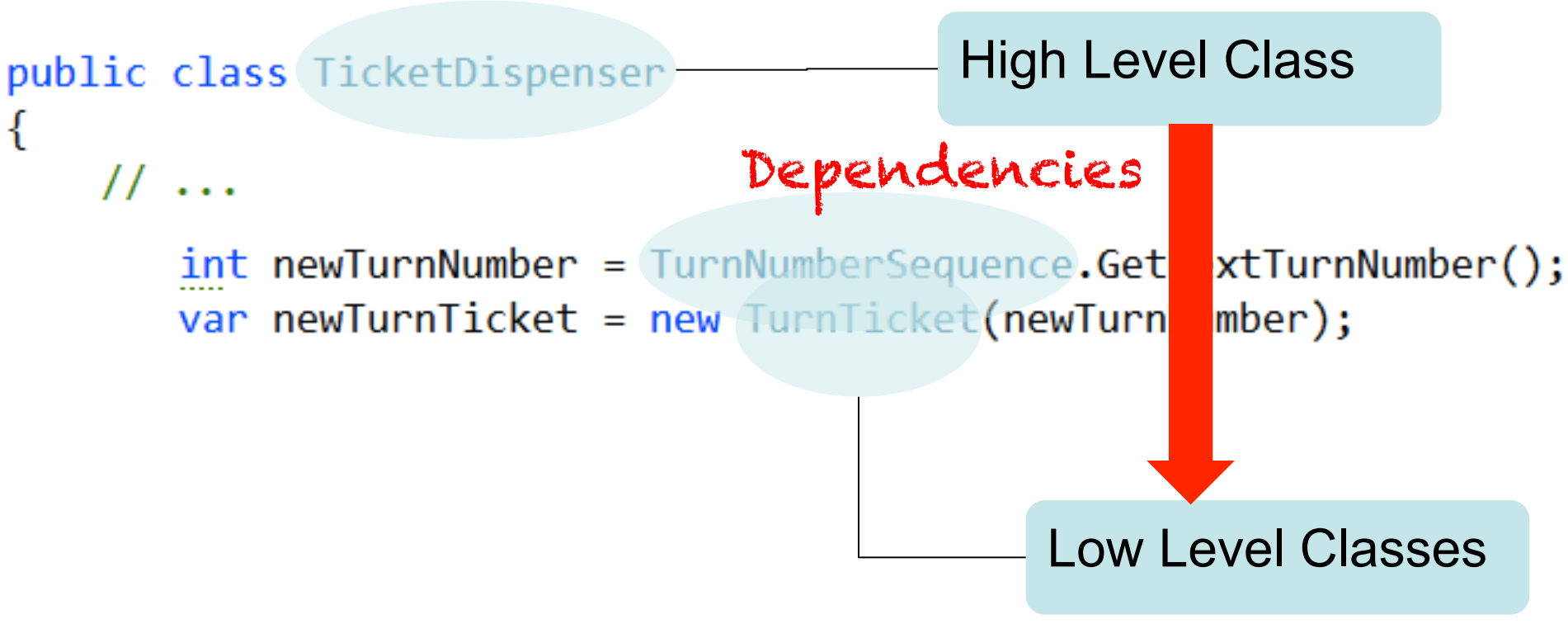
Minimizza i cambiamenti alla API pubblica più che puoi.

Extra credits:
TicketDispenser class viola alcuni dei principi OO e SOLID.
Prendi nota della linea di codice e il principio violato.

Encapsulation Violation In Example Code

```
public class TicketDispenser
{
    // ...

        int newTurnNumber = TurnNumberSequence.GetNextTurnNumber();
        var newTurnTicket = new TurnTicket(newTurnNumber);
```

DIP Violation In Example Code



OCP Violation In Example Code

```
public class TicketDispenser
{
    // ...

        int newTurnNumber = TurnNumberSequence.GetNextTurnNumber();
        var newTurnTicket = new TurnTicket(newTurnNumber);
```

Want to use a new type of sequence or ticket?
Must modify code; cannot extend it

```
public TicketDispenser() : this(TurnNumberSequence.GlobalTurnNumberSequence
{
}

public TicketDispenser(ITurnNumberSequence turnNumberSequence)
{
    this._turnNumberSequence = turnNumberSequence;
}

public TurnTicket GetTurnTicket()
{
    int newTurnNumber = _turnNumberSequence.GetNextTurnNumber();
```

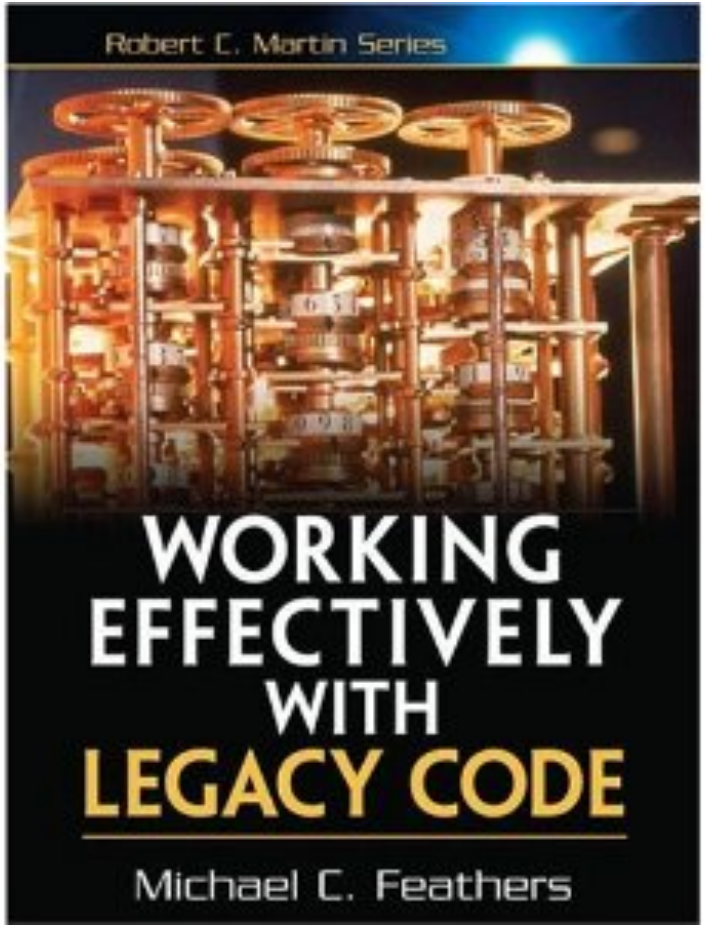
Dependency injection

```
public class TurnNumberSequence : ITurnNumberSequence
{
    public static readonly TurnNumberSequence GlobalTurnNumberSequence = new TurnNumberSequence();

    private int _turnNumber = 0;

    public int GetNextTurnNumber()
    {
        return _turnNumber++;
    }
}
```

Introduce Instance Delegator

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**.

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.

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.

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.

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.

Reference: SRP

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

Pag. 151/152

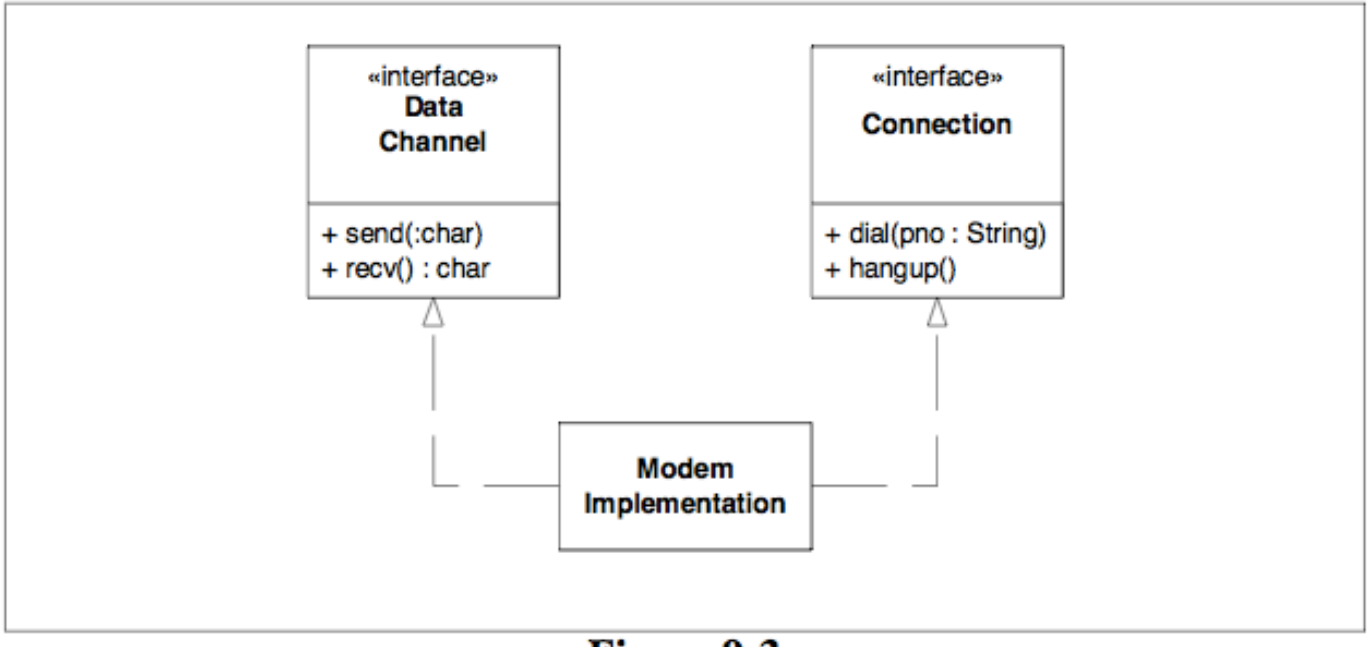
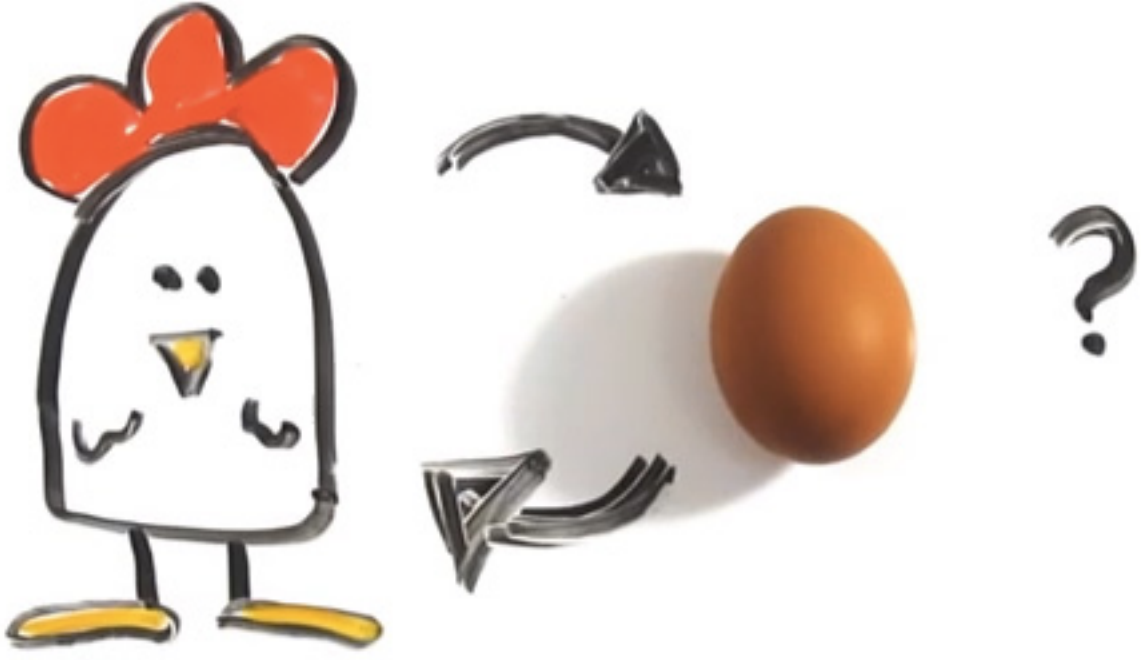


Figure 9-3
Separated Modem Interface

Refactoring and TDD



Synergy between testing and design

Michael Feathers:

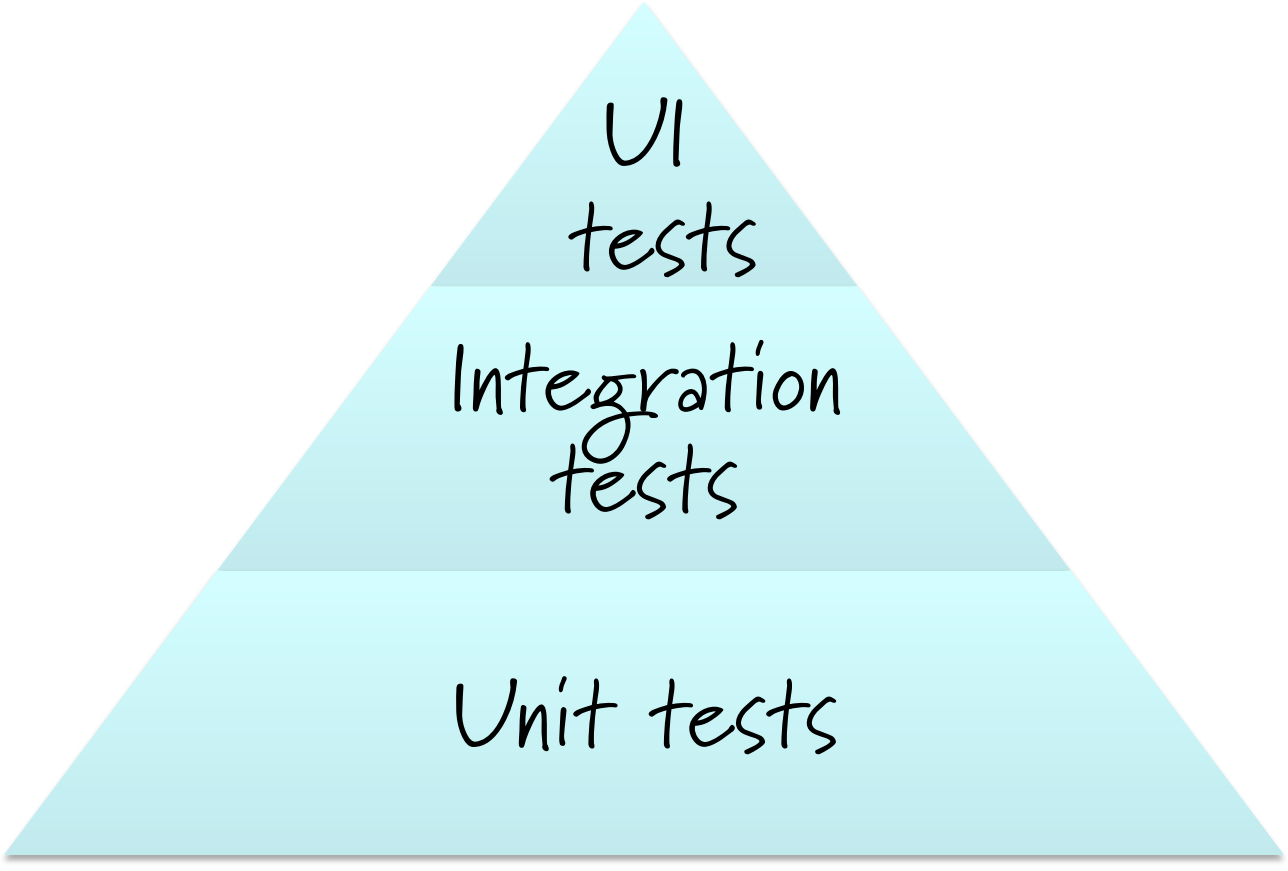
writing tests is another way to look the code and
locally understand it and reuse it,
and that is the same goal of good OO design.

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

Other testing strategies for legacy code

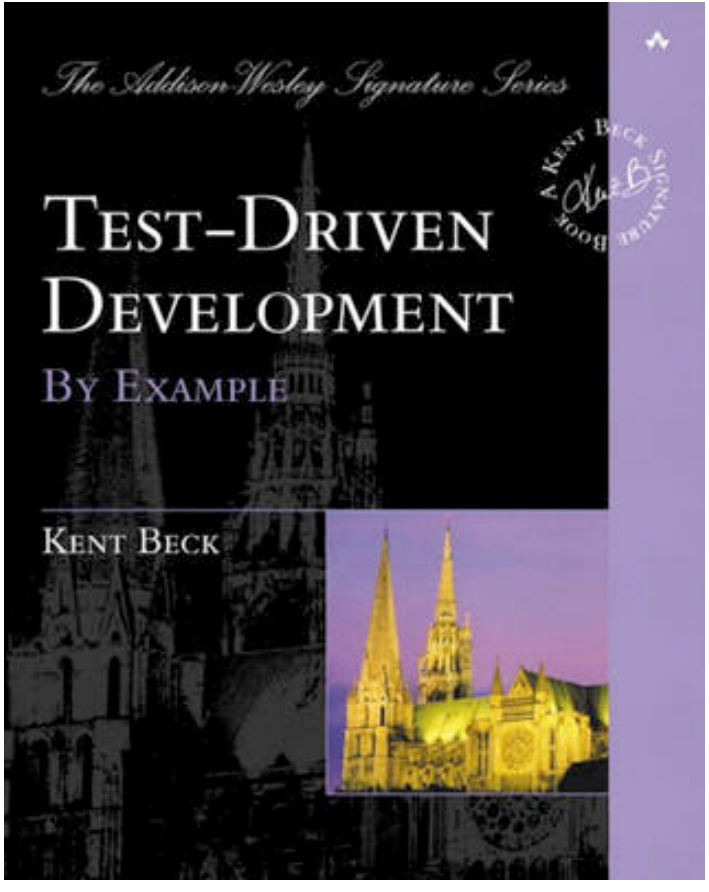
- Start with other tests
- Use an automatic refactoring tool
- Strangler pattern
- DDD anti-corruption layer

Mike Cohn's Test Pyramid. Explained !



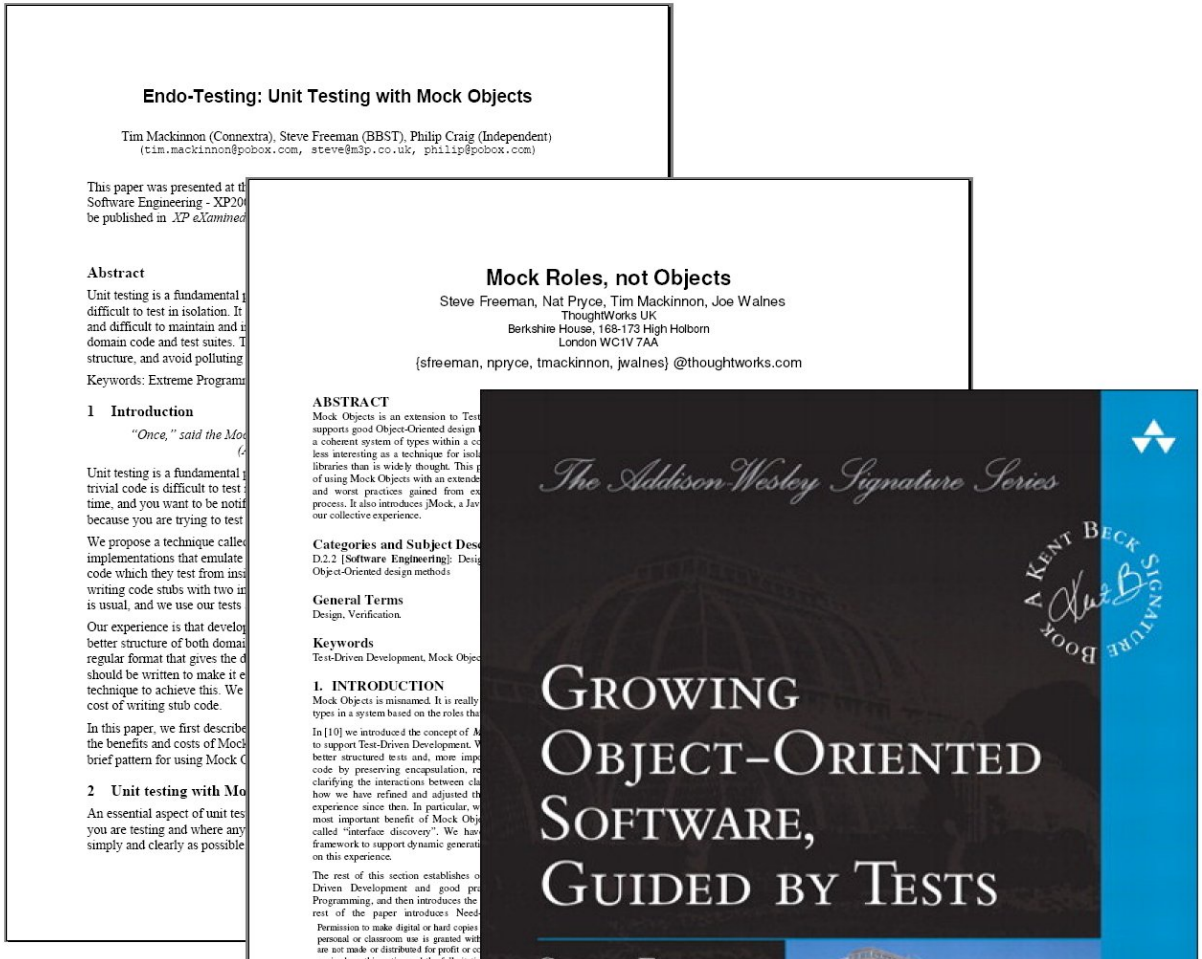
ThoughtWorks®

More references



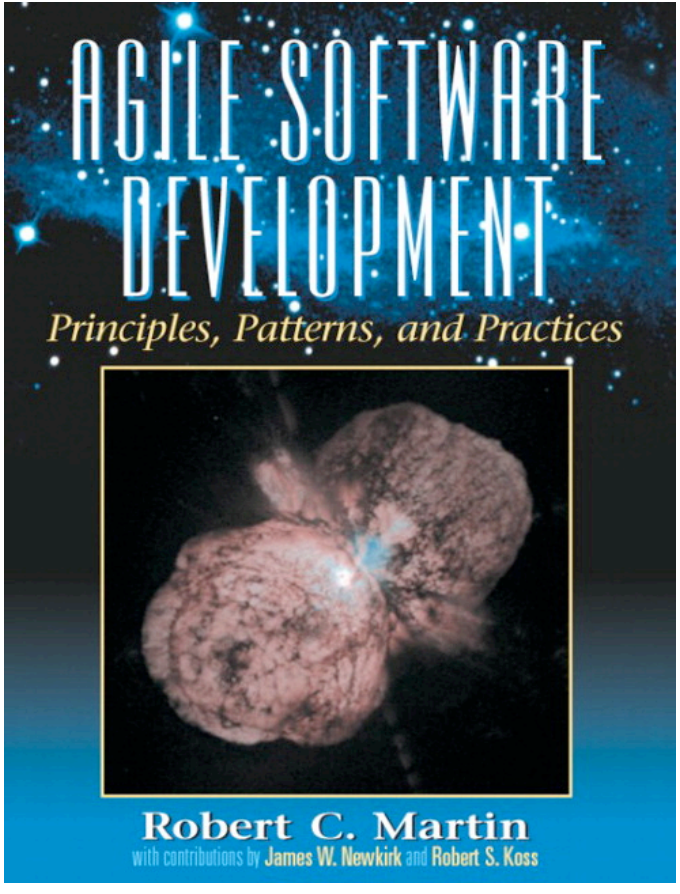
ThoughtWorks®

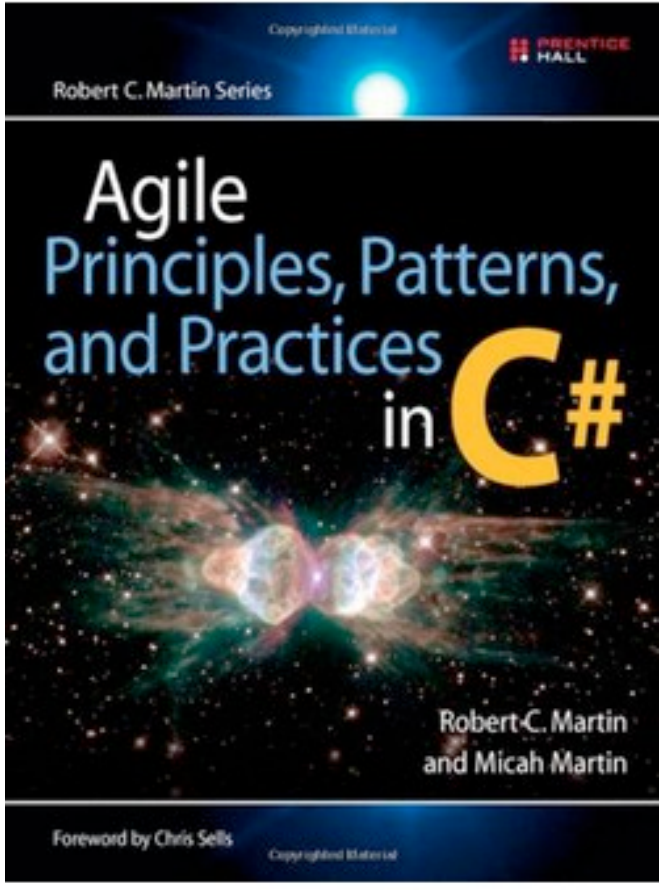
More references



ThoughtWorks®

More references





ThoughtWorks®

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 :

@GGBOZZO

@ILIASBARTOLINI

@LUKADOTNET

ThoughtWorks®