https://github.com/shawnewallace/tdd-workshop

((CENTRIC))

www.centricconsulting.com

Test Driven Design

(60 min) First session presentation

Introduction

The Case for TDD

Types of Testing

Example

(120 min) Pairing session. 30-minute sessions executing any of several code katas.

LUNCH BREAK

(60 min) Afternoon session presentation

Design for testability

Mocking

How to get started on my project

(120 min) Pairing session, Legacy refactor.

Introduction

TESTING

I FIND YOUR LACK OF TESTS DISTURBING.

I DON'T ALWAYS TEST MY CODE
BUT WHEN I DO I DO IT IN PRODUCTION
memegenerator.net

What it is

A Software Development Practice

What it is

Benefits

- Productivity
- Emergent Design
- Better Code
- Reduced Gold-Plating
- Regression Test Suite

What it’s not

A panacea

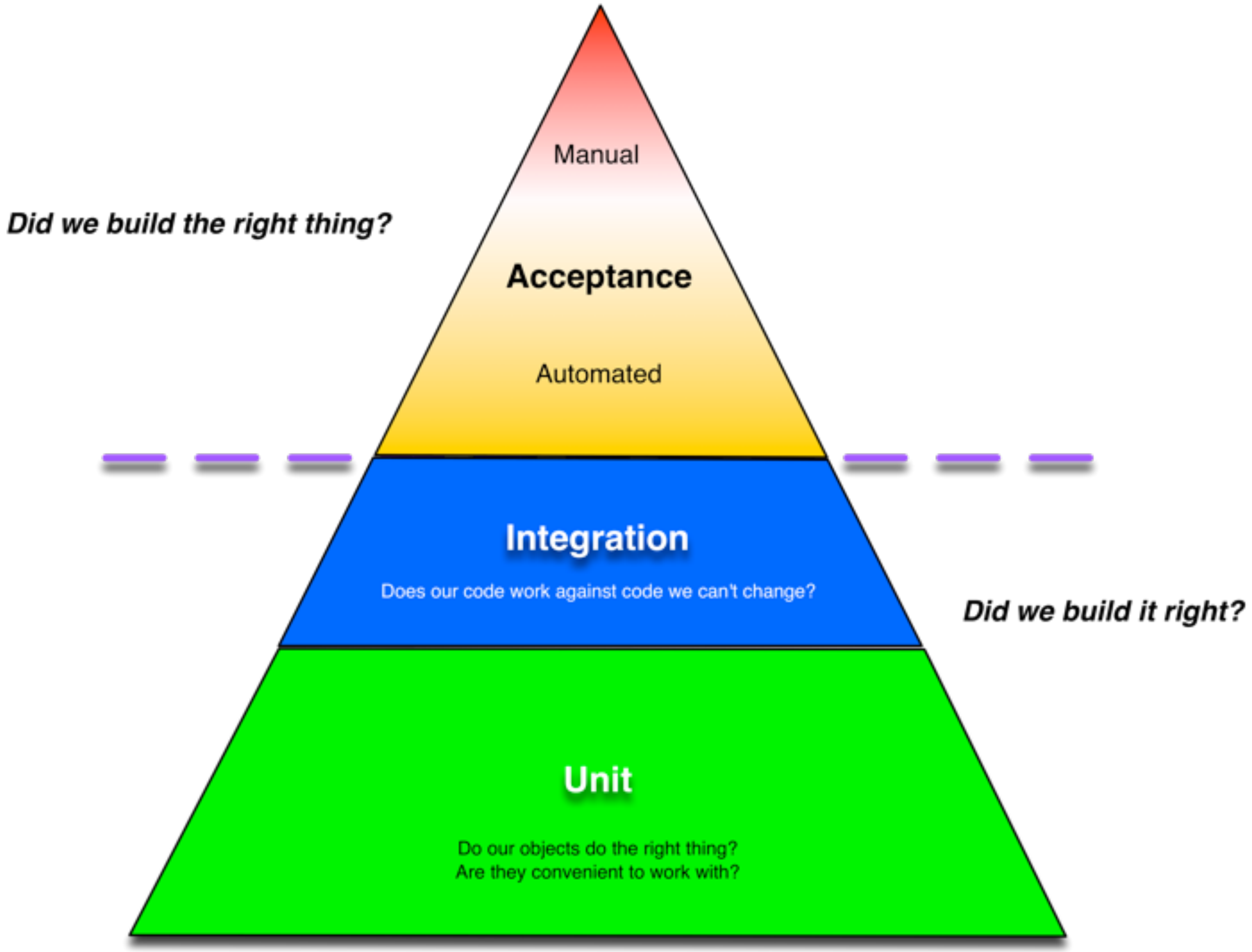
Done wrong, it’s still wrong

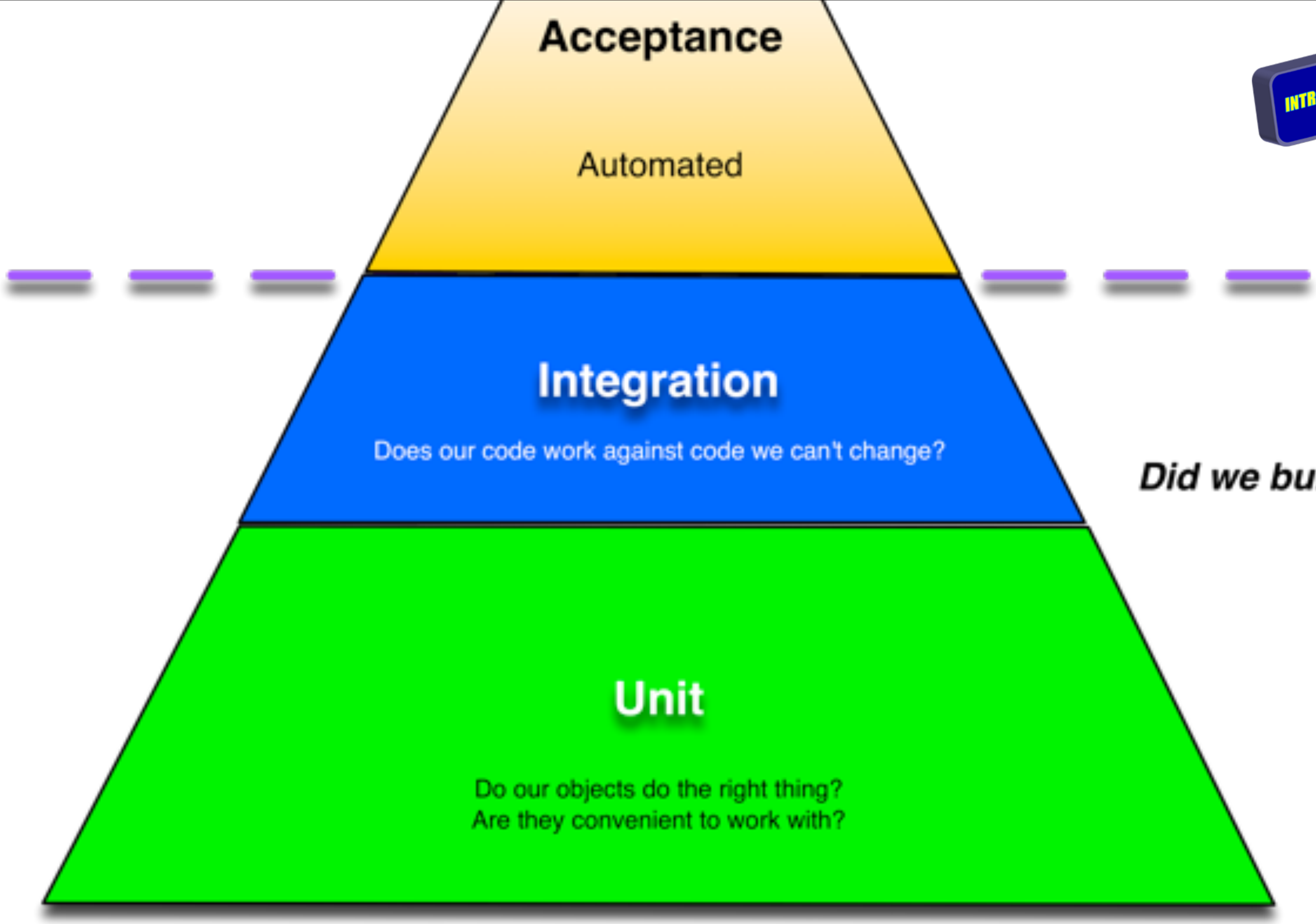
What it’s not

Shortcomings

- Can be difficult
- Management support is crucial
- Self-test paradigm
- Overhead
- Hard to get meaningful coverage in legacy systems

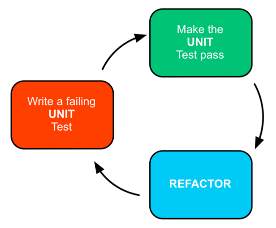
Types of Testing


Manual
Did we build the right thing?
Acceptance
Automated
Integration
Does our code work against code we can't change?
Did we build it right?
Unit
Do our objects do the right thing?
Are they convenient to work with?

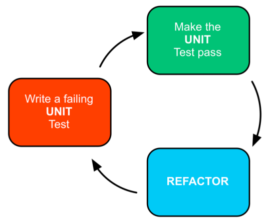

Acceptance
Automated
Integration
Does our code work against code we can't change?
Unit
Do our objects do the right thing?
Are they convenient to work with?

Did we build it ri

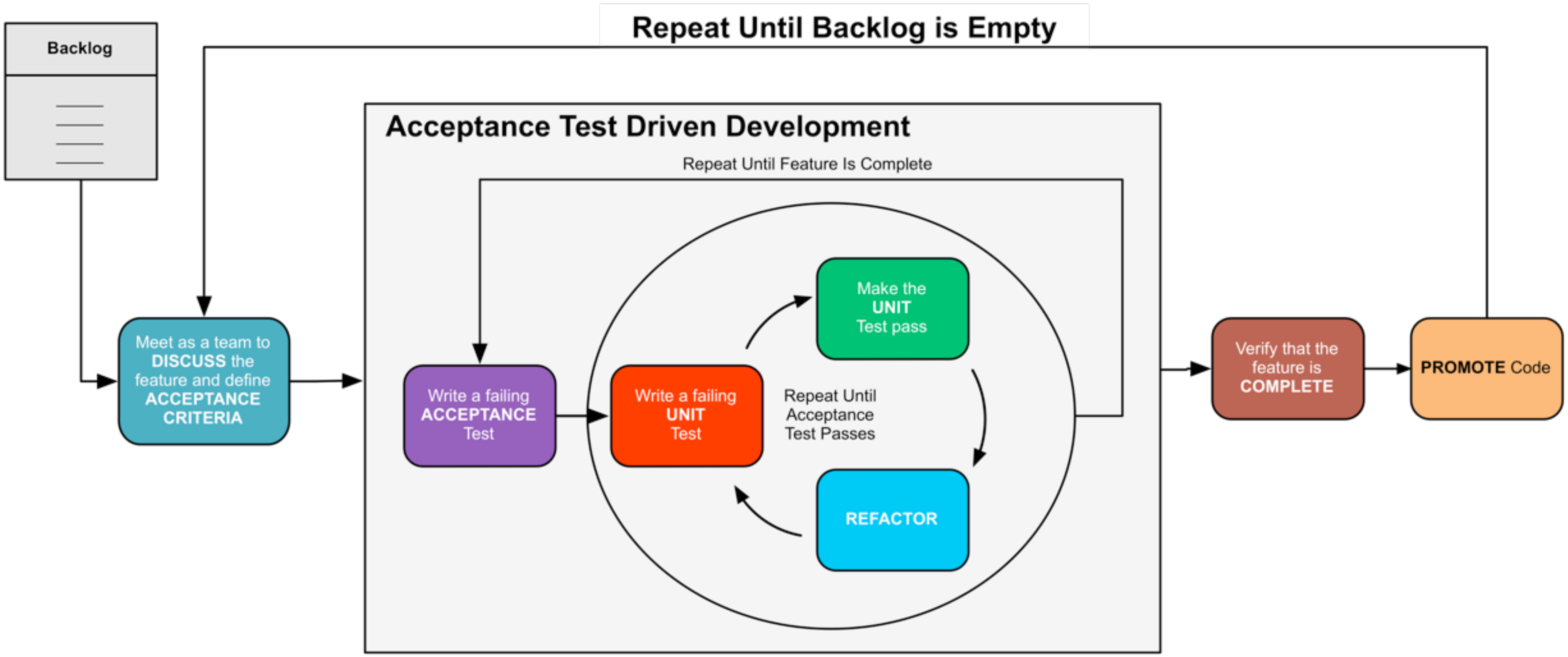
Studies

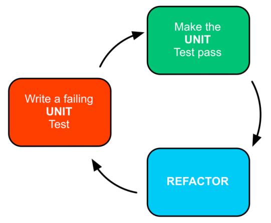


Test Driven Development

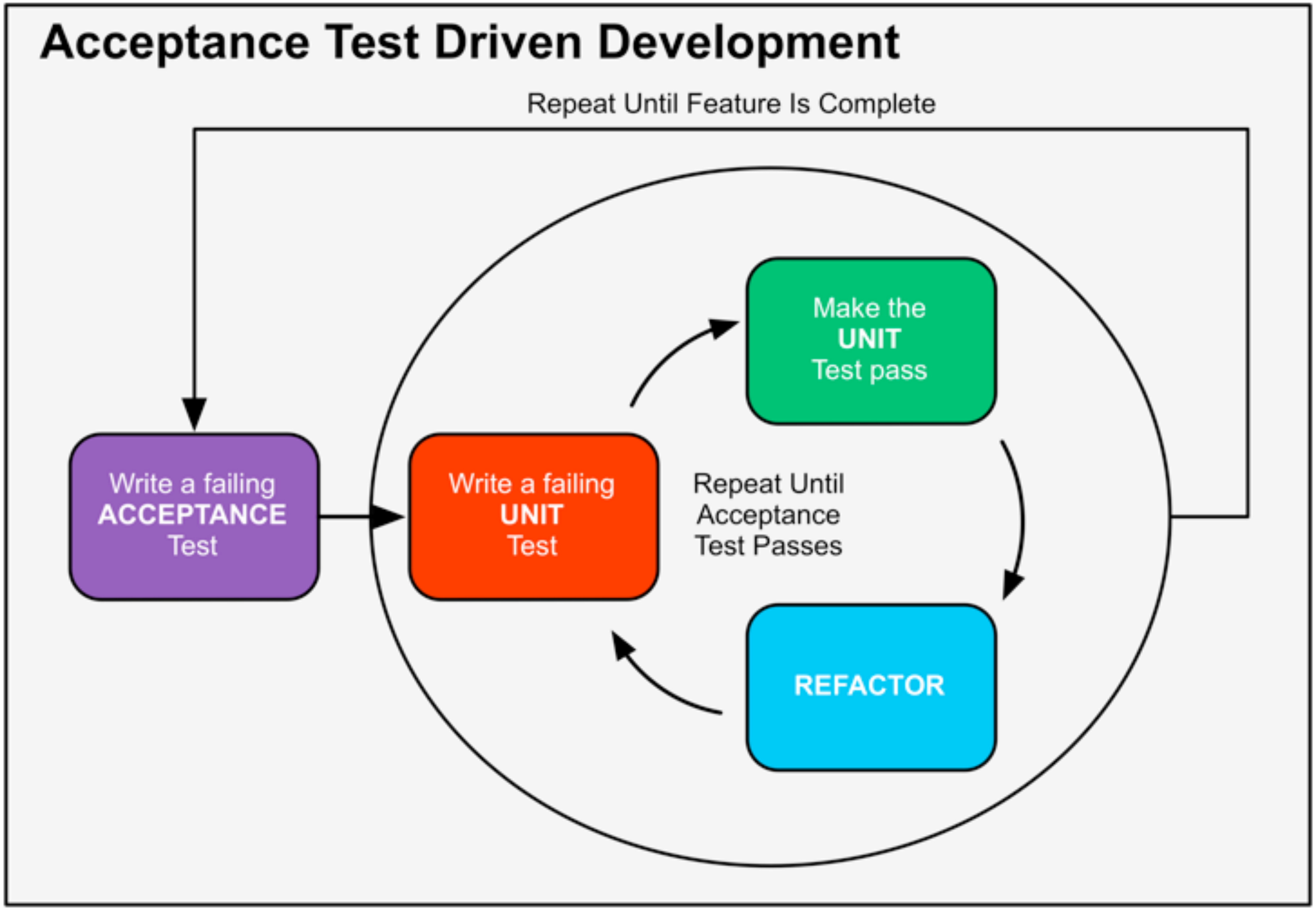

Make the UNIT Test pass
Write a failing UNIT Test
REFACTOR

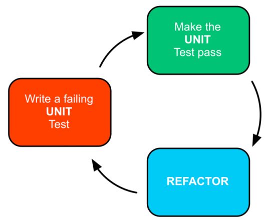
Workflow


Repeat Until Backlog is Empty
Backlog
Meet as a team to DISCUSS the feature and define ACCEPTANCE CRITERIA
Acceptance Test Driven Development
Repeat Until Feature Is Complete
Write a failing ACCEPTANCE Test
Write a failing UNIT Test
Make the UNIT Test pass
Repeat Until Acceptance Test Passes
REFACTOR
Verify that the feature is COMPLETE
PROMOTE Code

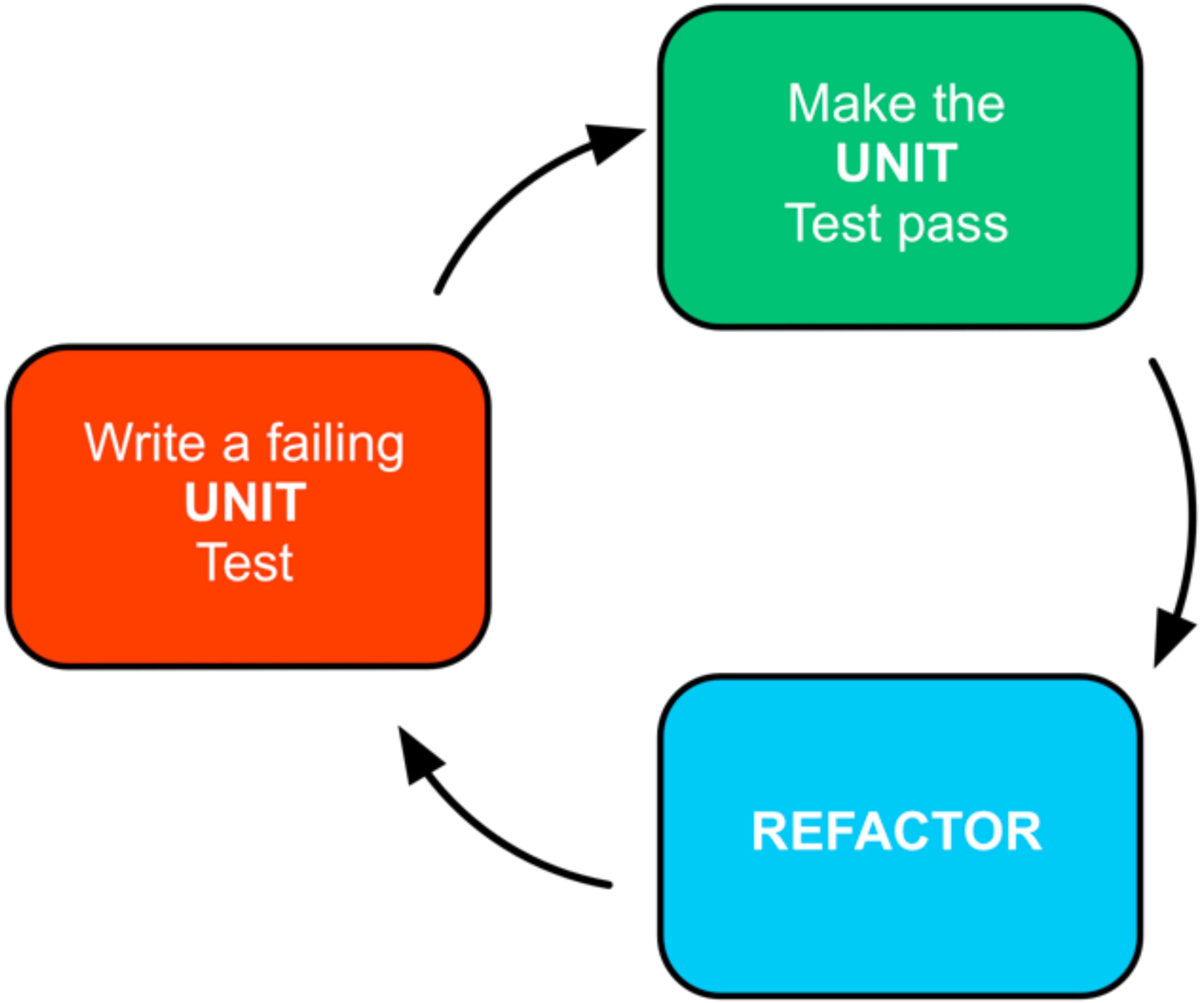

Make the UNIT Test pass
Write a failing UNIT Test
REFACTOR

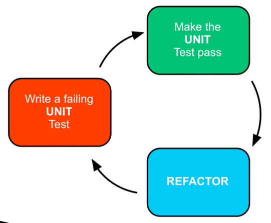
ATDD


Acceptance Test Driven Development
Repeat Until Feature Is Complete
Write a failing ACCEPTANCE Test
Write a failing UNIT Test
Make the UNIT Test pass
REFACTOR
Repeat Until Acceptance Test Passes

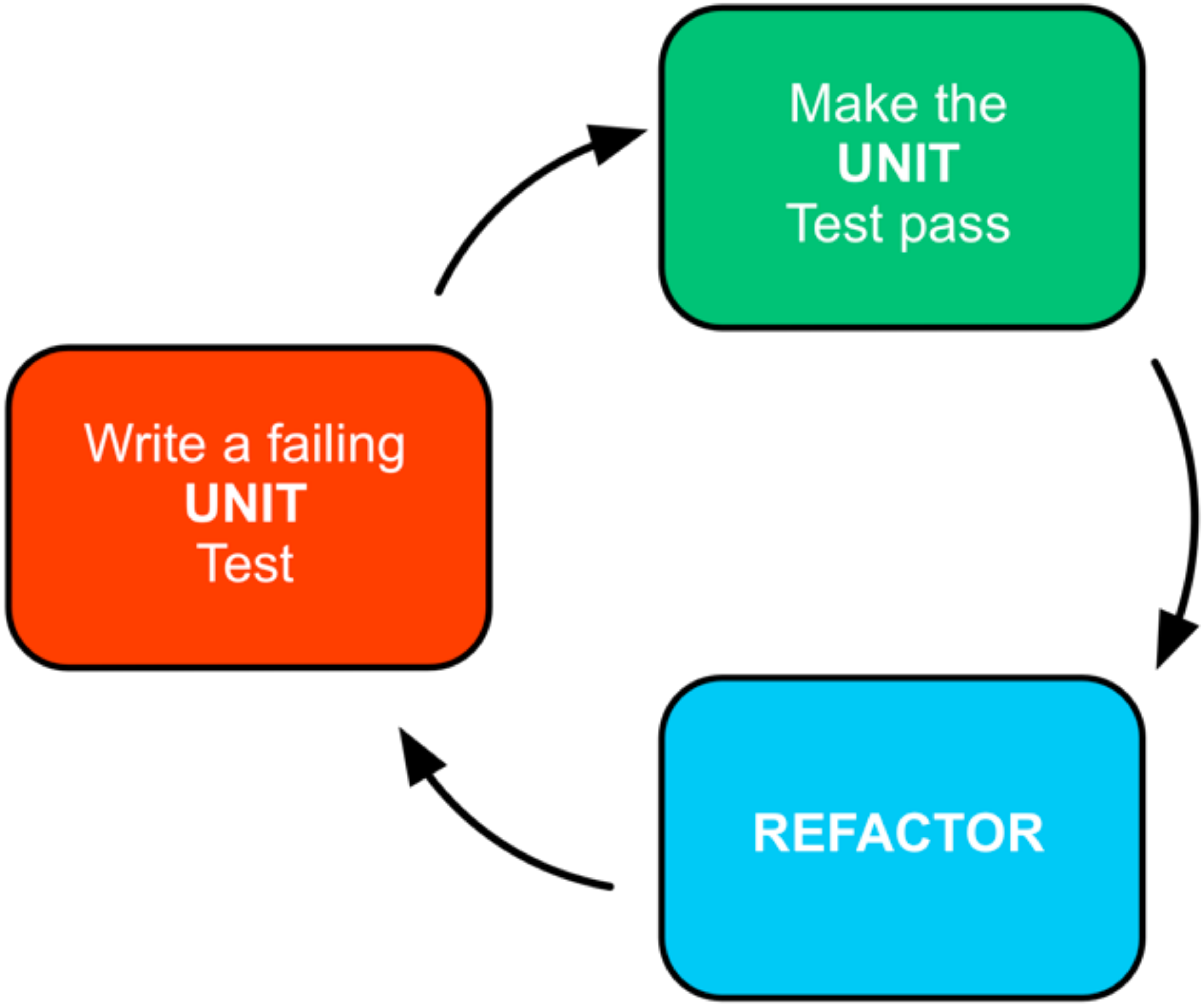

Make the
UNIT
Test pass
Write a failing
UNIT
Test
REFACTOR

TDD Flow

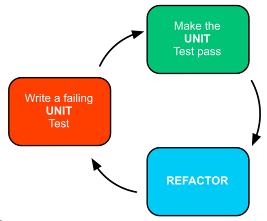

Make the
UNIT
Test pass
Write a failing
UNIT
Test
REFACTOR



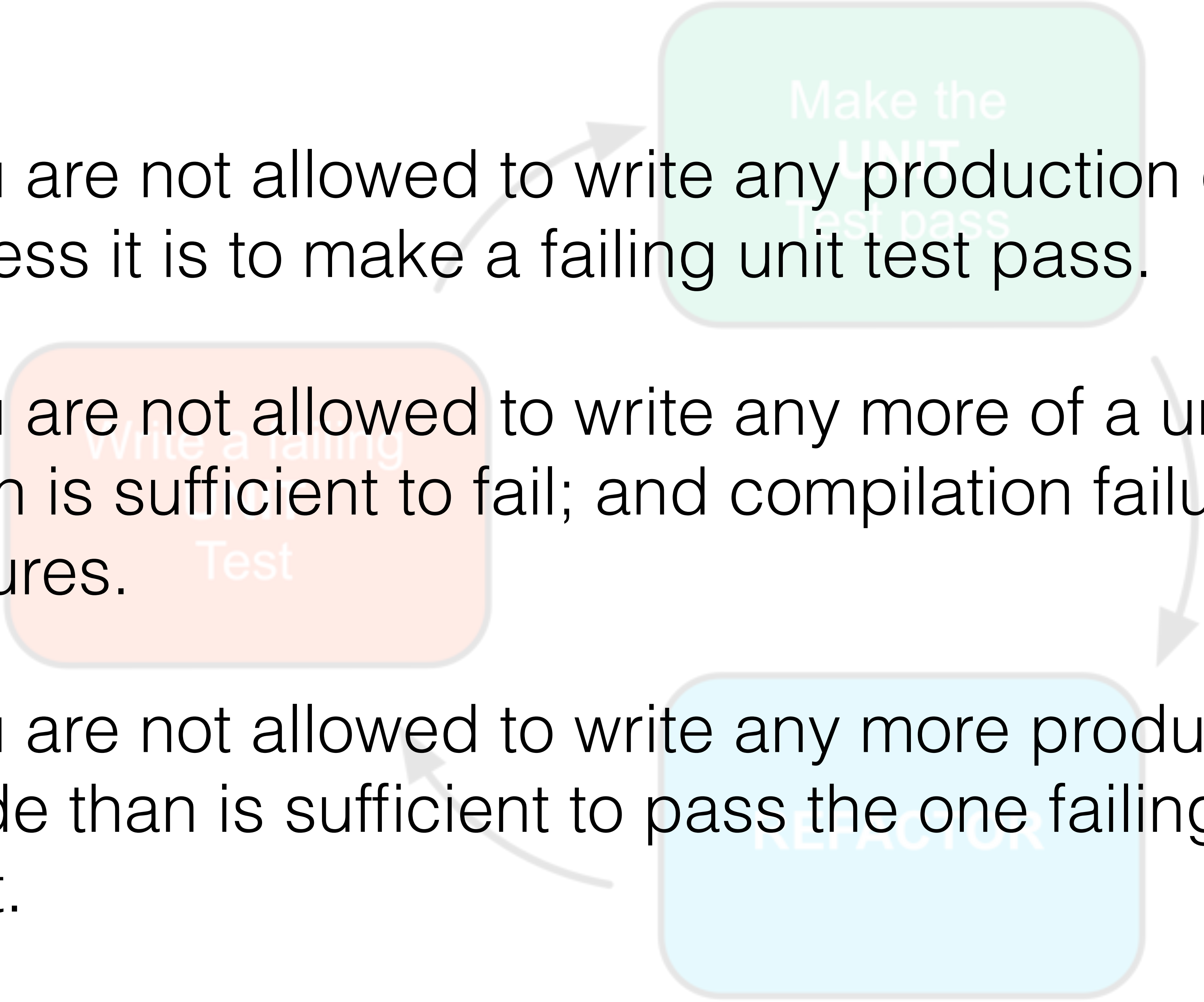
The Three Rules of TDD



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



The Three Rules of TDD

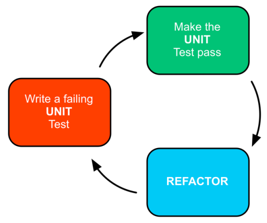


1. You are not allowed to write any production code unless it is to make a failing unit test pass.

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.

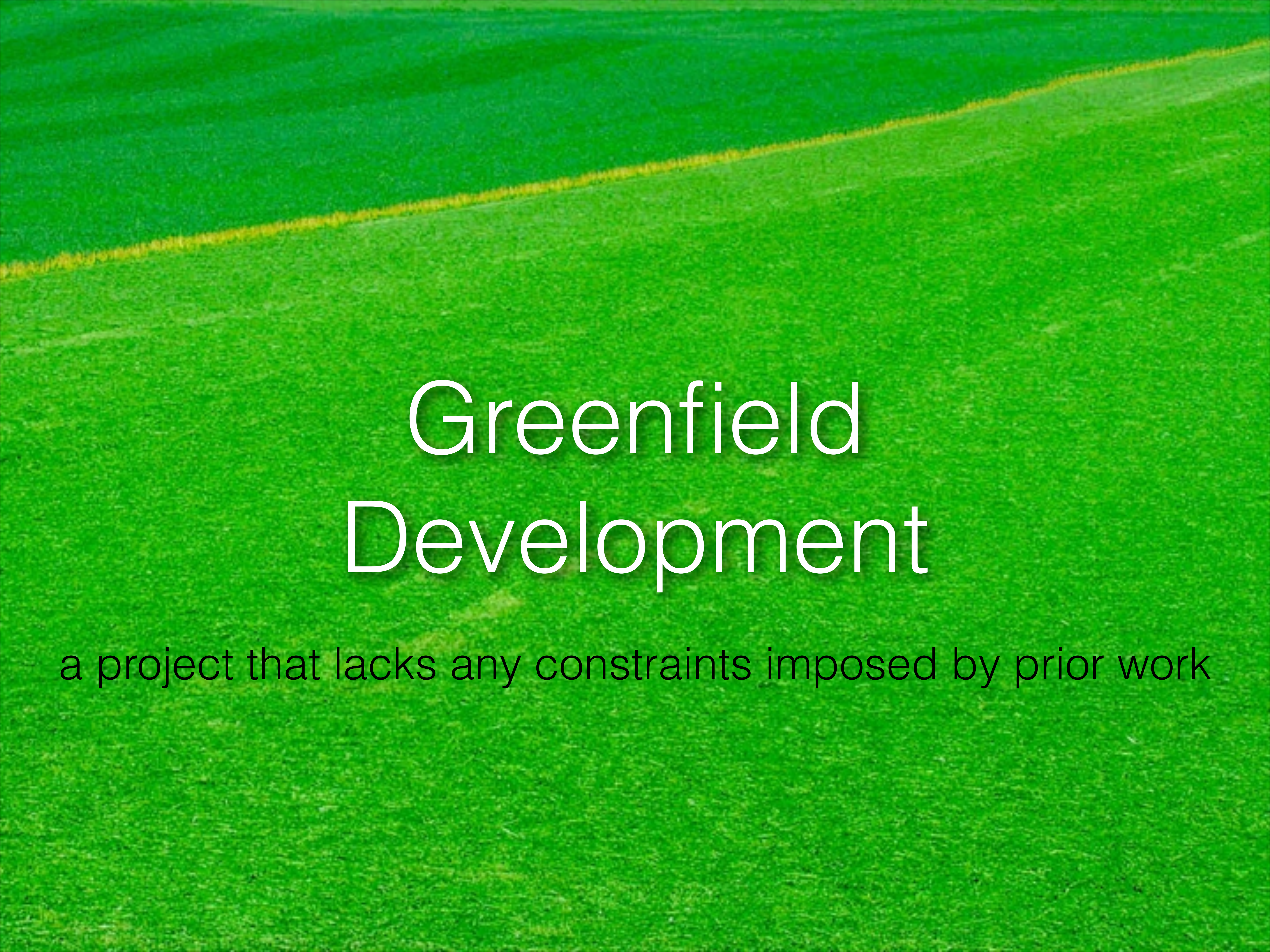
http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd



How to Decide

Not really qualified until you do it and are good at it.

It has to work for your project and your team, it might not.



Greenfield Development

a project that lacks any constraints imposed by prior work

How do I start

Workflow

Fibonacci Numbers

$$F_n = F_{n-1} + F_{n-2}$$

$$where$$

$$F_0 = 0, F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$
$$where$$
$$F_0 = 0, F_1 = 1$$

Red

$$F_n = F_{n-1} + F_{n-2}$$
$$where$$
$$F_0=0, F_1=1$$

Green

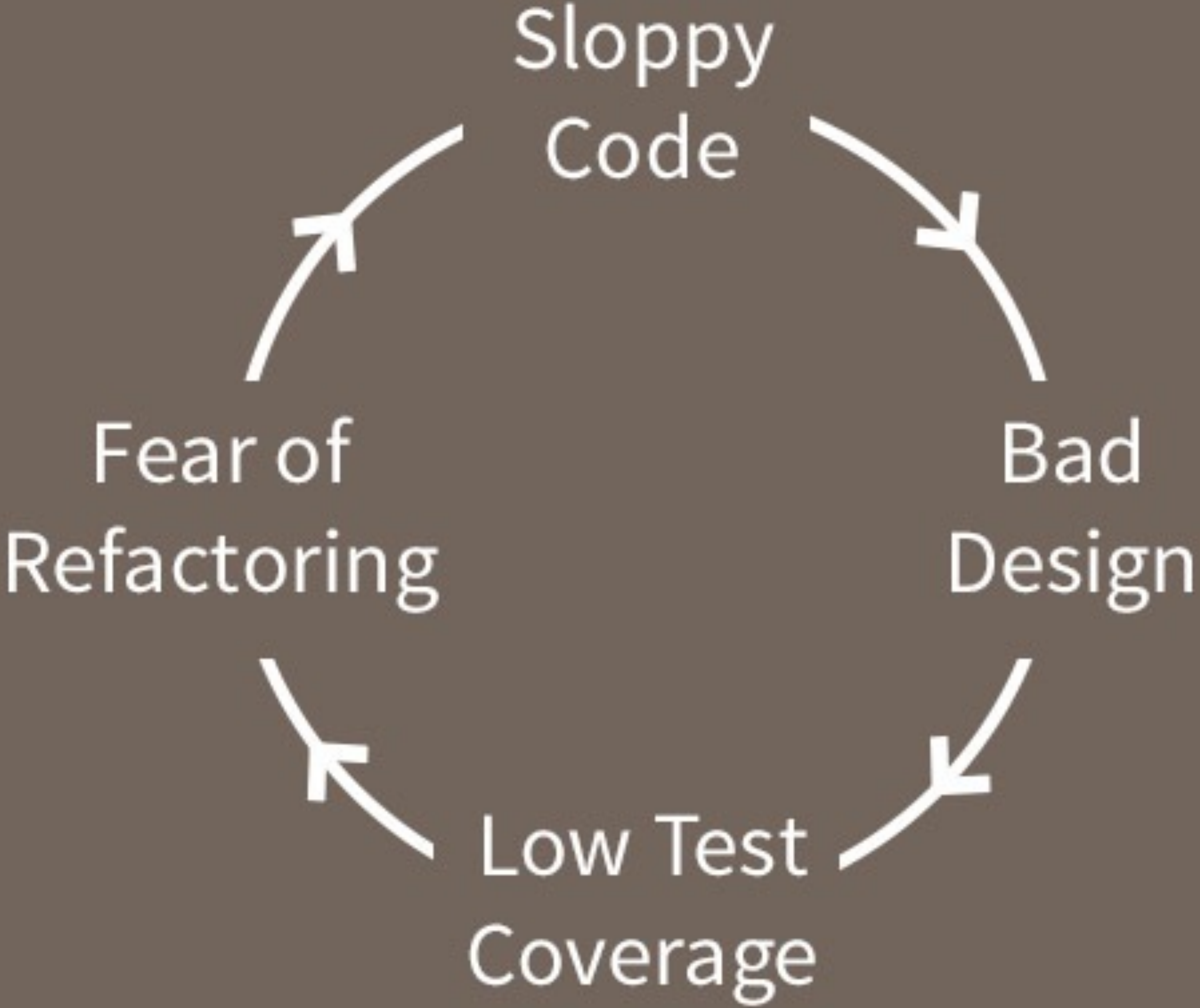
$$F_n = F_{n-1} + F_{n-2}$$
$$where$$
$$F_0 = 0, F_1 = 1$$

Refactor

JET LI IN HIS FINAL MARTIAL ARTS EPIC
JET LI'S
FEARLESS
A RONNY YU FILM
"Mastering others is strength. Mastering yourself makes you fearless." – LAO TZU
PG-13 PARENTS STRONGLY CAUTIONED
Violence and Martial Arts Action Throughout
www.jetlisfearless.com
ROGUE PICTURES
From the producers of "HERO" and "CROUCHING TIGER, HIDDEN DRAGON"

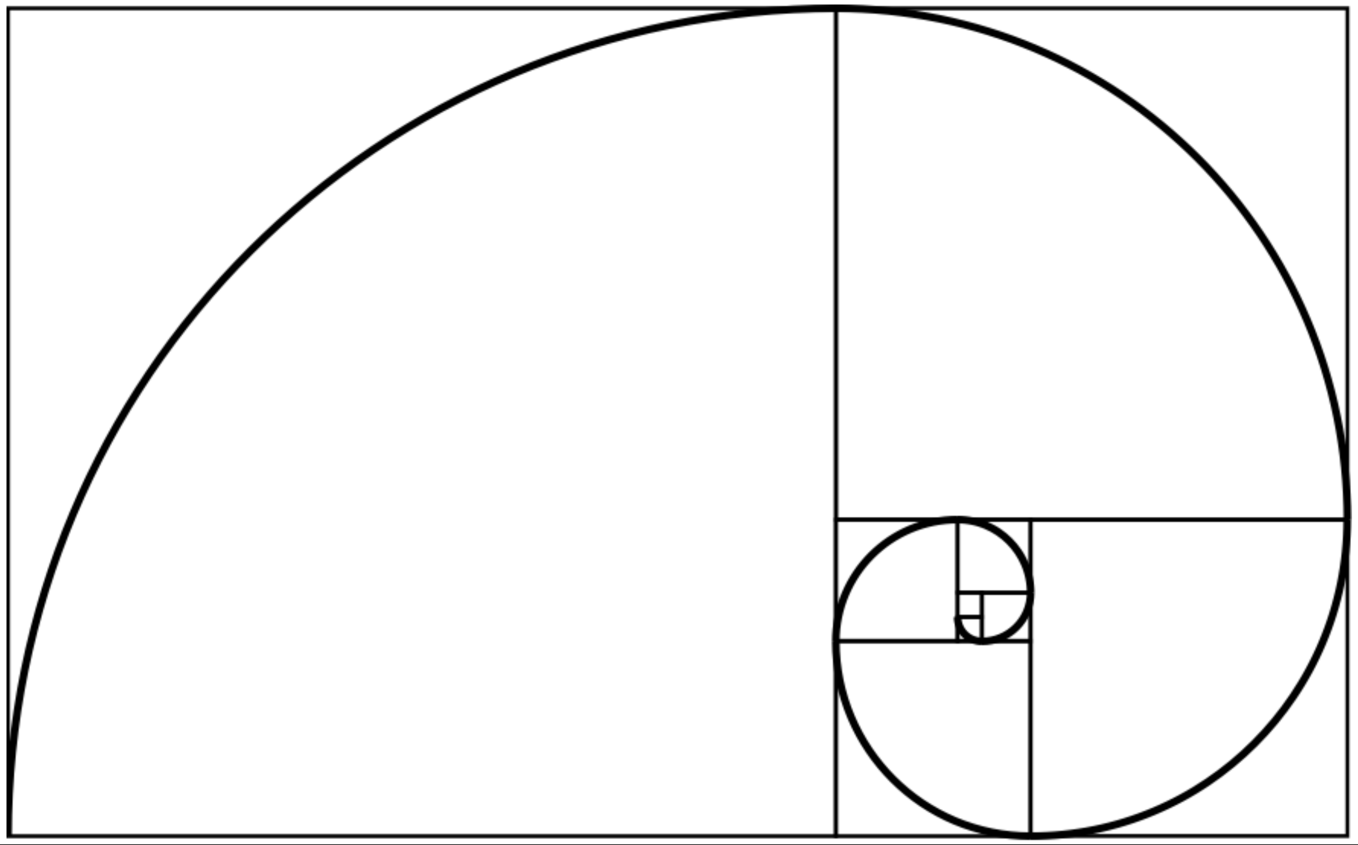
$$F_n = F_{n-1} + F_{n-2}$$
$$where$$
$$F_0 = 0, F_1 = 1$$

Refactor without Fear


Sloppy Code
Bad Design
Low Test Coverage
Fear of Refactoring

$$Fibonacci(n) = \frac{\emptyset^n - (1 - \emptyset)^n}{\sqrt{5}}$$

$$\emptyset = \frac{1 + \sqrt{5}}{2}$$

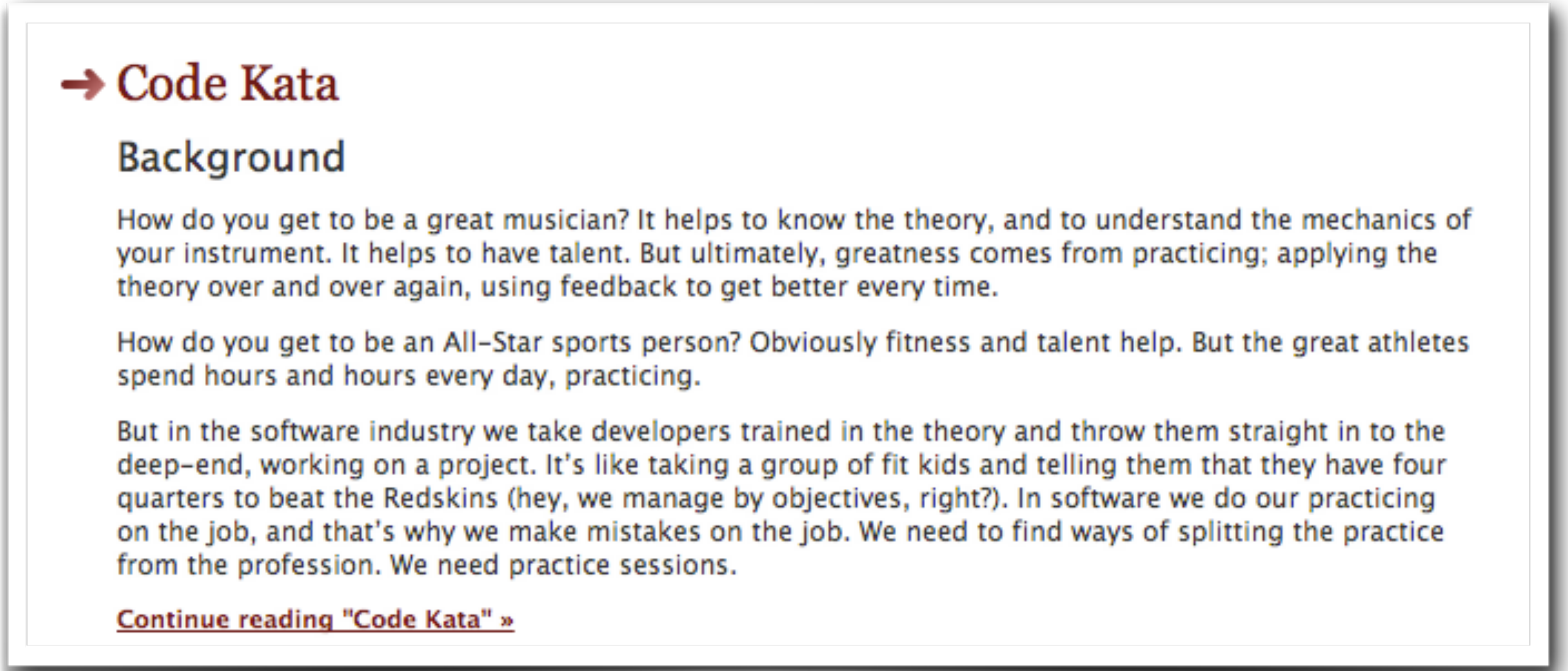


<lab 1>

<lab 1>

Work in Pairs

The Code Kata



→ Code Kata

Background

How do you get to be a great musician? It helps to know the theory, and to understand the mechanics of your instrument. It helps to have talent. But ultimately, greatness comes from practicing; applying the theory over and over again, using feedback to get better every time.

How do you get to be an All-Star sports person? Obviously fitness and talent help. But the great athletes spend hours and hours every day, practicing.

But in the software industry we take developers trained in the theory and throw them straight in to the deep-end, working on a project. It's like taking a group of fit kids and telling them that they have four quarters to beat the Redskins (hey, we manage by objectives, right?). In software we do our practicing on the job, and that's why we make mistakes on the job. We need to find ways of splitting the practice from the profession. We need practice sessions.

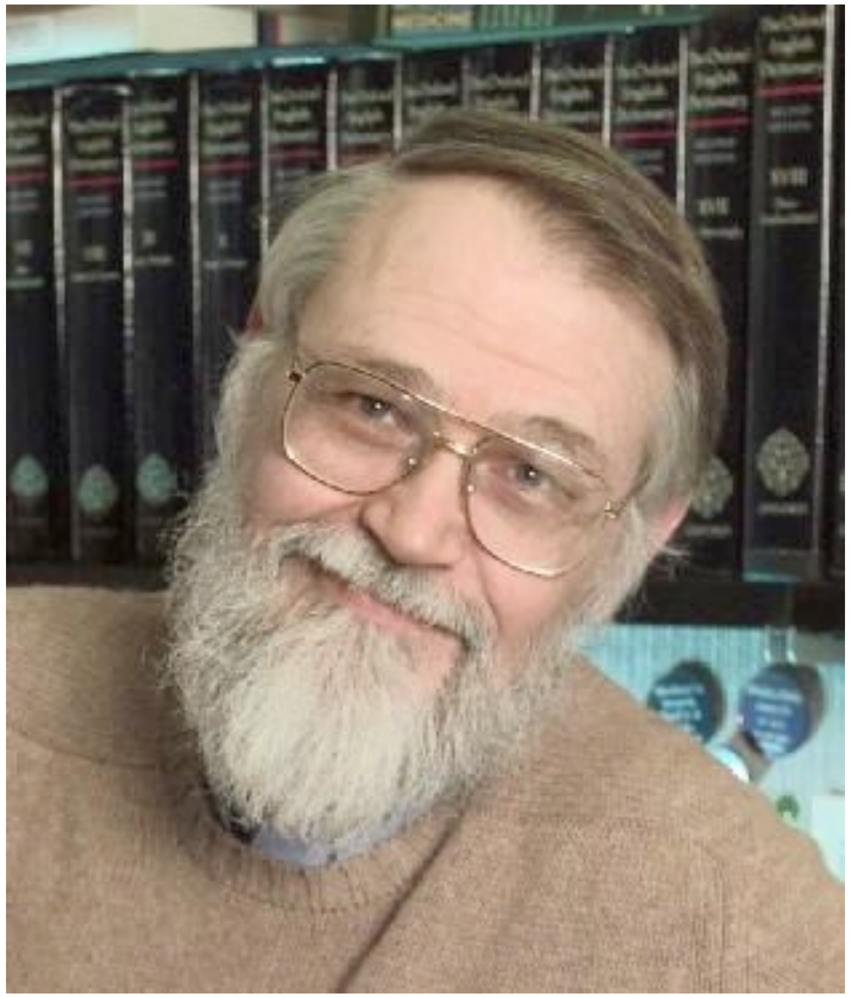
Continue reading "Code Kata" »

http://codekata.pragprog.com/

The Code Kata

- FizzBuzz
- Bowling Game
- Leap Year Calculator
- Tennis Match
- Roman Numeral Converter
- Urinal Kata

Design for Testability



"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

--Brian Kernighan

Object Oriented Principles

Coupling and Cohesion

Tight vs. Loose Coupling
Interdependency
Coordination
Information Flow

High vs. Low Cohesion
Robustness
Reliability
Reusability

We want **LOOSE COUPLING**

and

HIGH COHESION

Object Oriented Principles

Object Oriented Principles

Single Responsibility Principle

Open/closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Object Oriented Principles

- ★Single Responsibility Principle
- Open/closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Object Oriented Principles

Single Responsibility Principle

★Open/closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Object Oriented Principles

Single Responsibility Principle

Open/closed Principle

★Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Object Oriented Principles

Single Responsibility Principle

Open/closed Principle

Liskov Substitution Principle

★Interface Segregation Principle

Dependency Inversion Principle

Interface Segregation Principle

Code to interfaces, depending on your languages

Object Oriented Principles

Single Responsibility Principle

Open/closed Principle

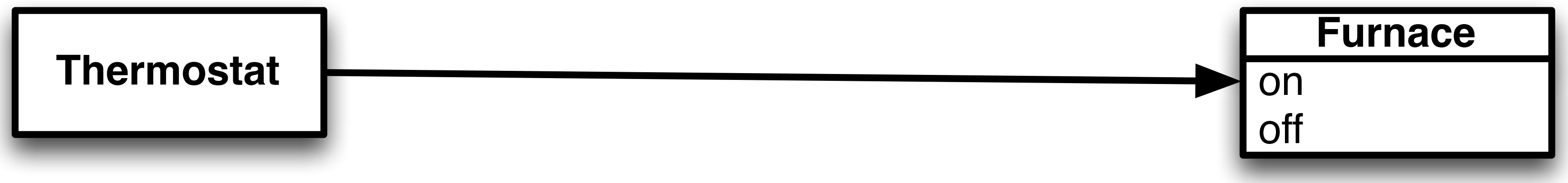
Liskov Substitution Principle

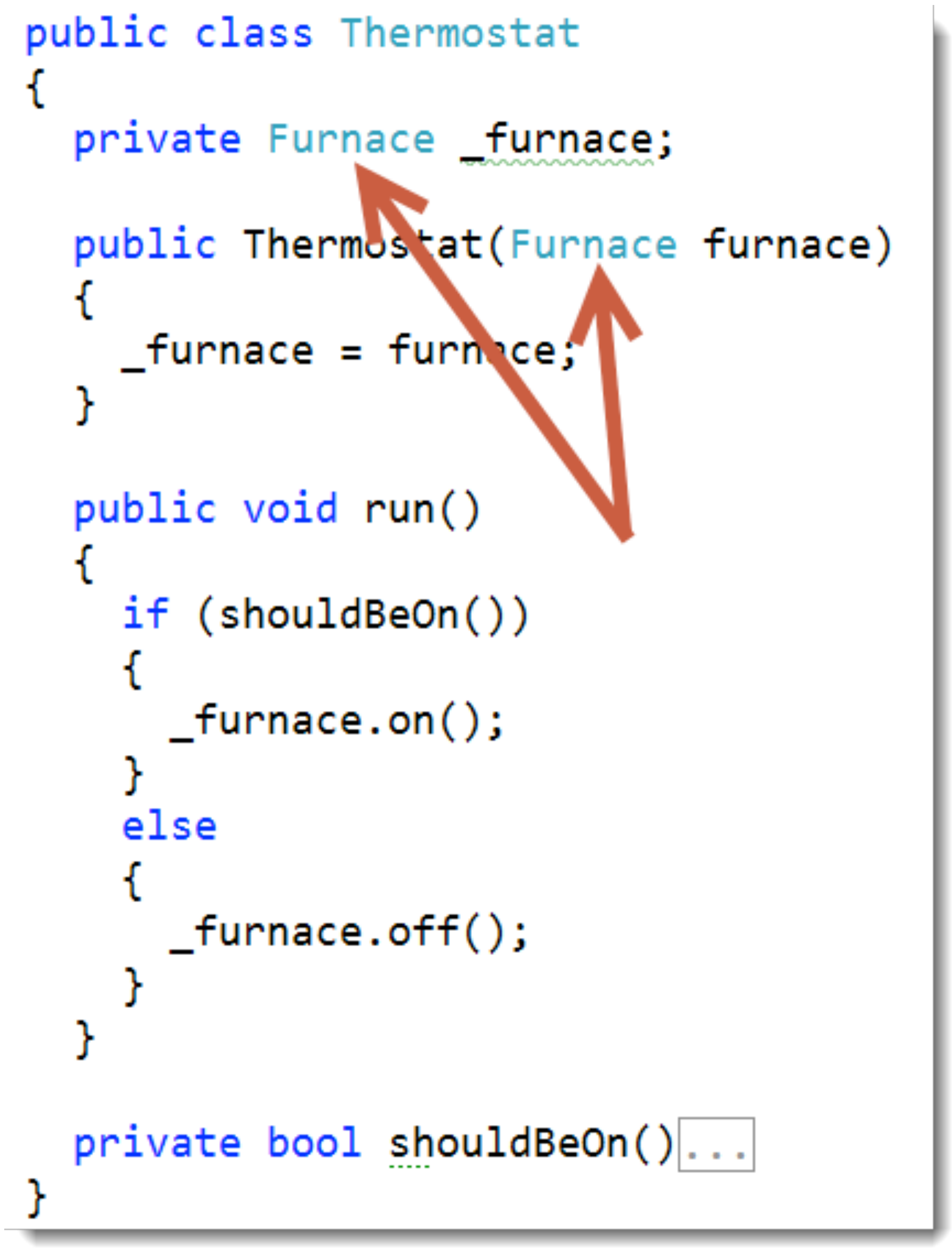
Interface Segregation Principle

★Dependency Inversion Principle

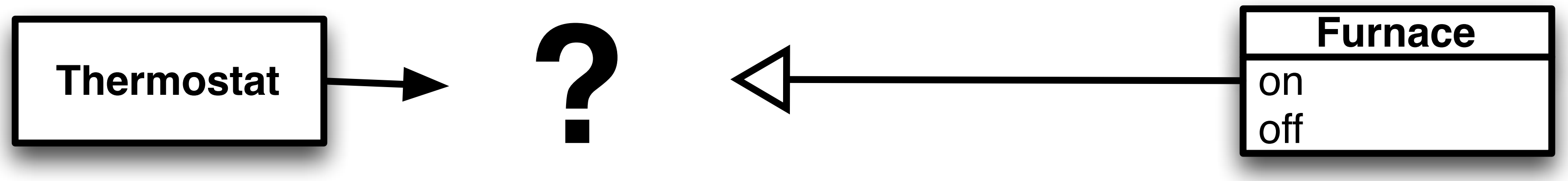
Dependency Inversion Principle

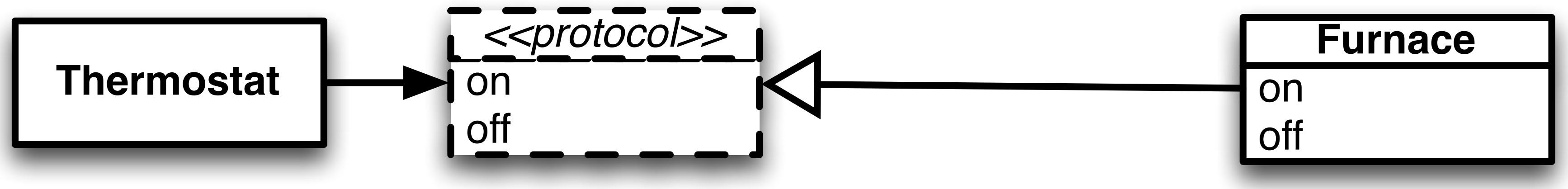
“Depend on abstractions, not concretions”

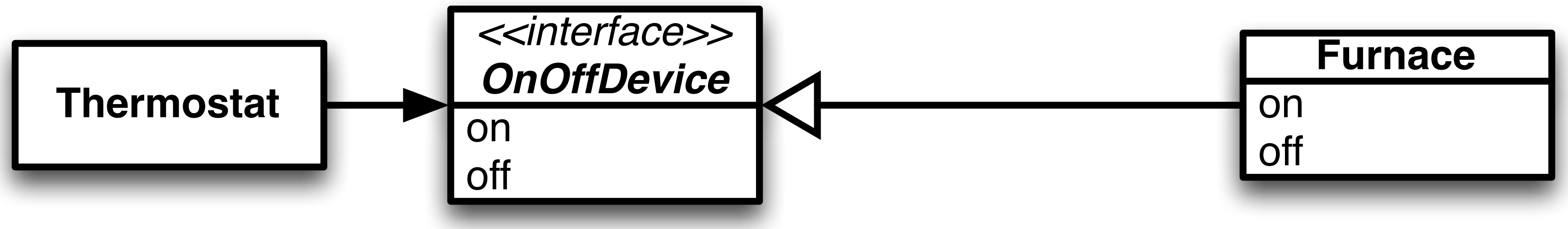

Thermostat
Furnace
on
off

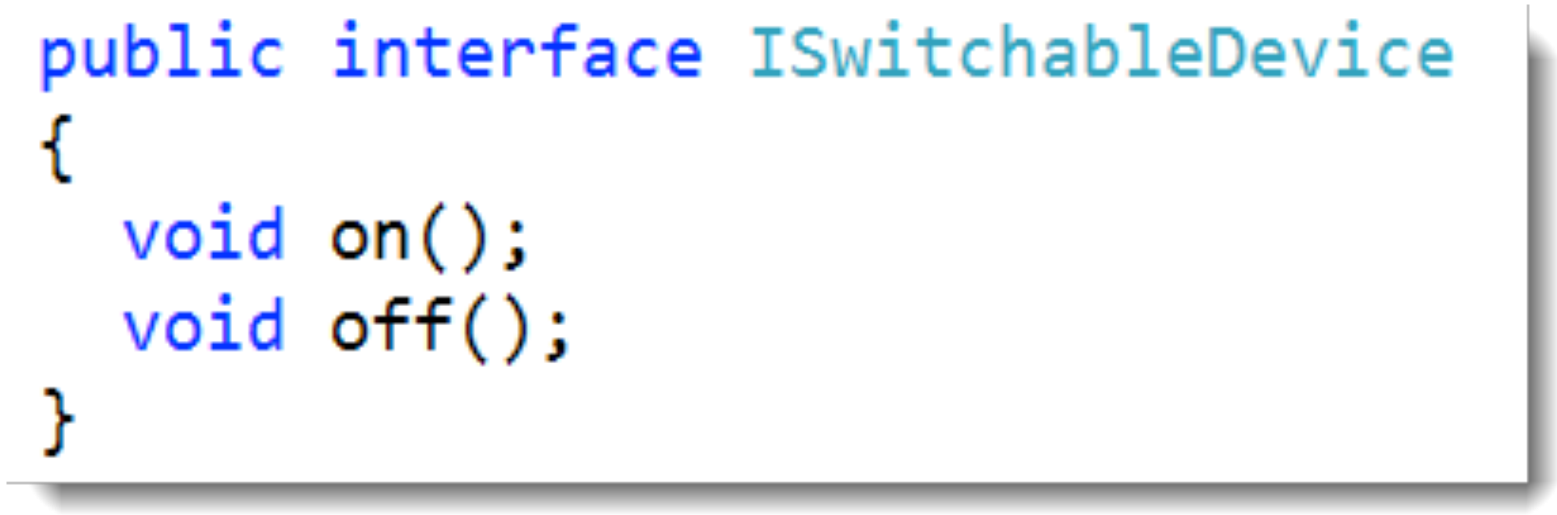


```
public class Thermostat
{
  private Furnace _furnace;

  public Thermostat(Furnace furnace)
  {
    _furnace = furnace;
  }

  public void run()
  {
    if (shouldBeOn())
    {
      _furnace.on();
    }
    else
    {
      _furnace.off();
    }
  }

  private bool shouldBeOn()...
}
```

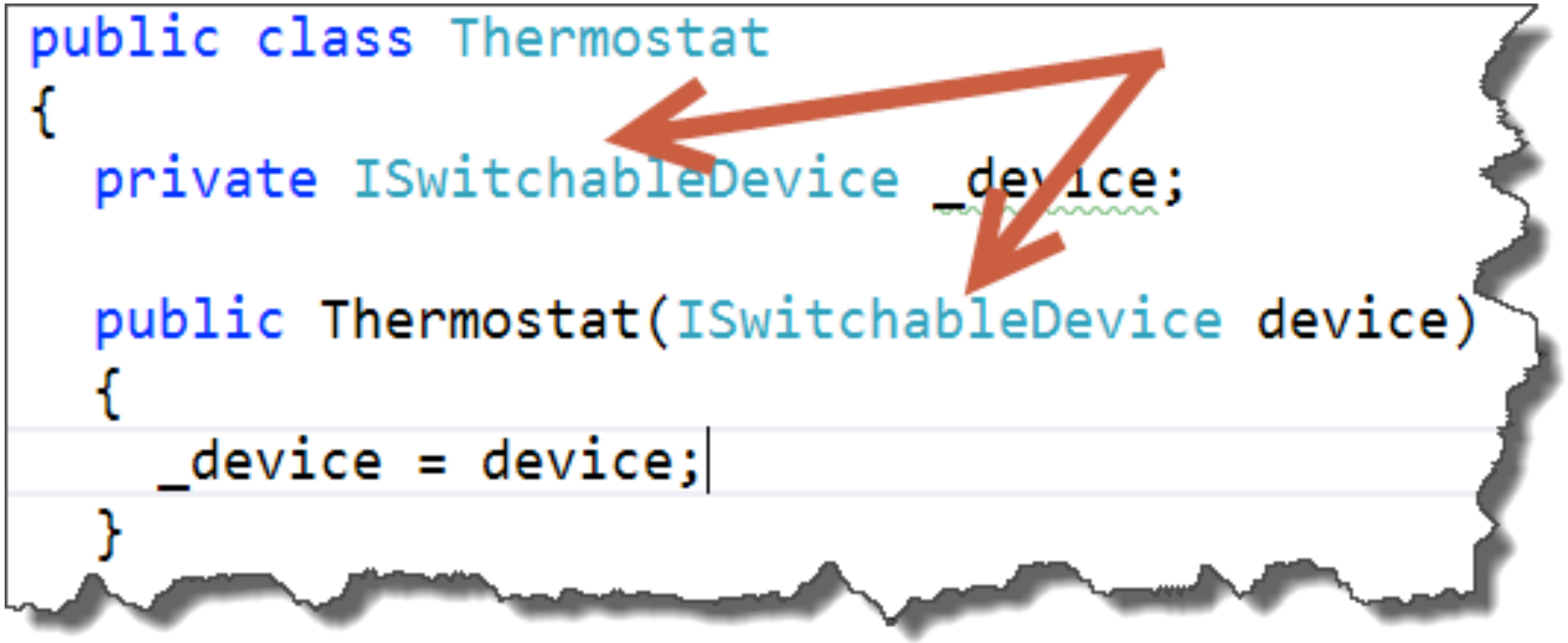

Thermostat
?
Furnace
on
off


Thermostat
<<protocol>>
on
off
Furnace
on
off

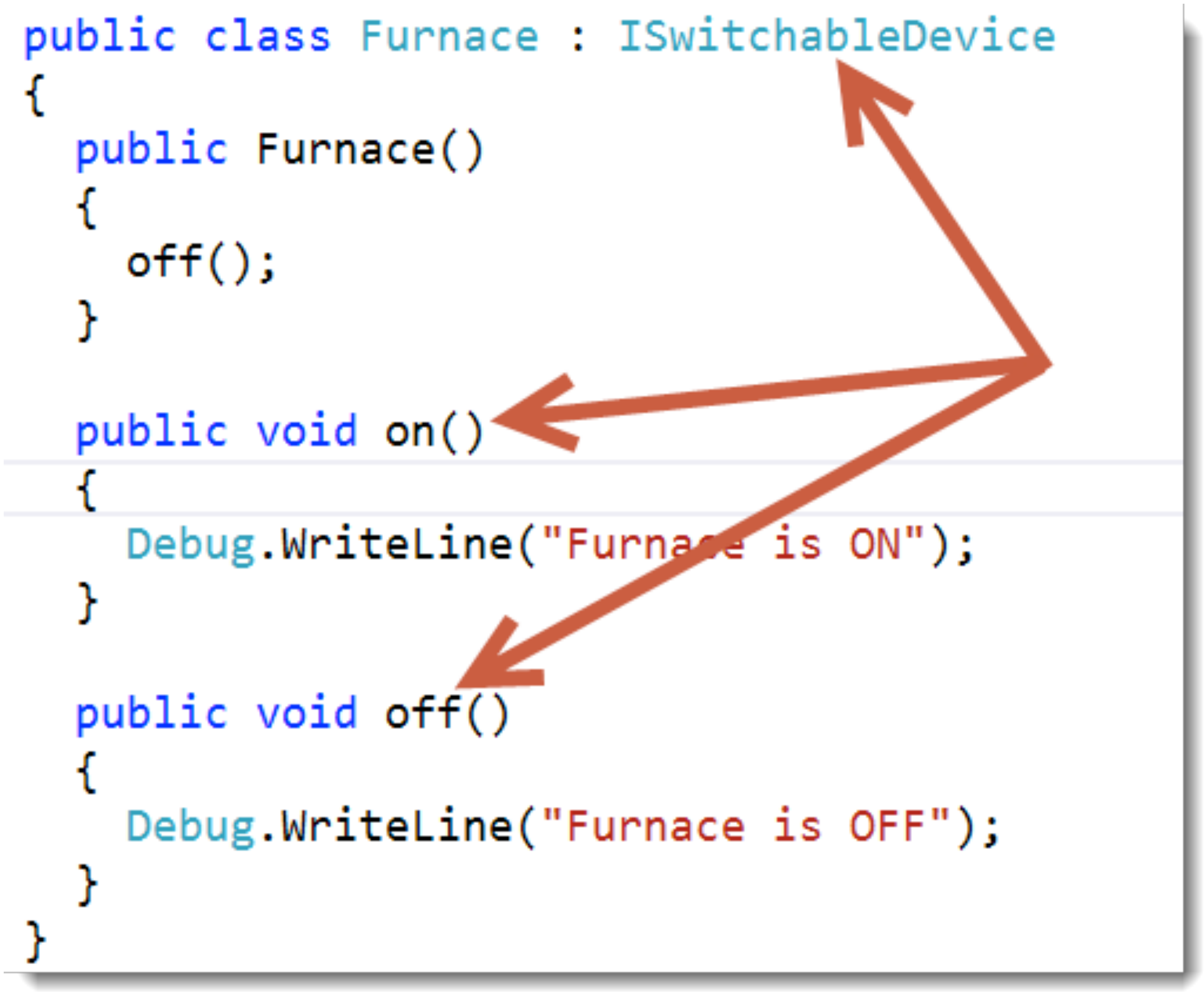

Thermostat
<<interface>>
OnOffDevice
on
off
Furnace
on
off



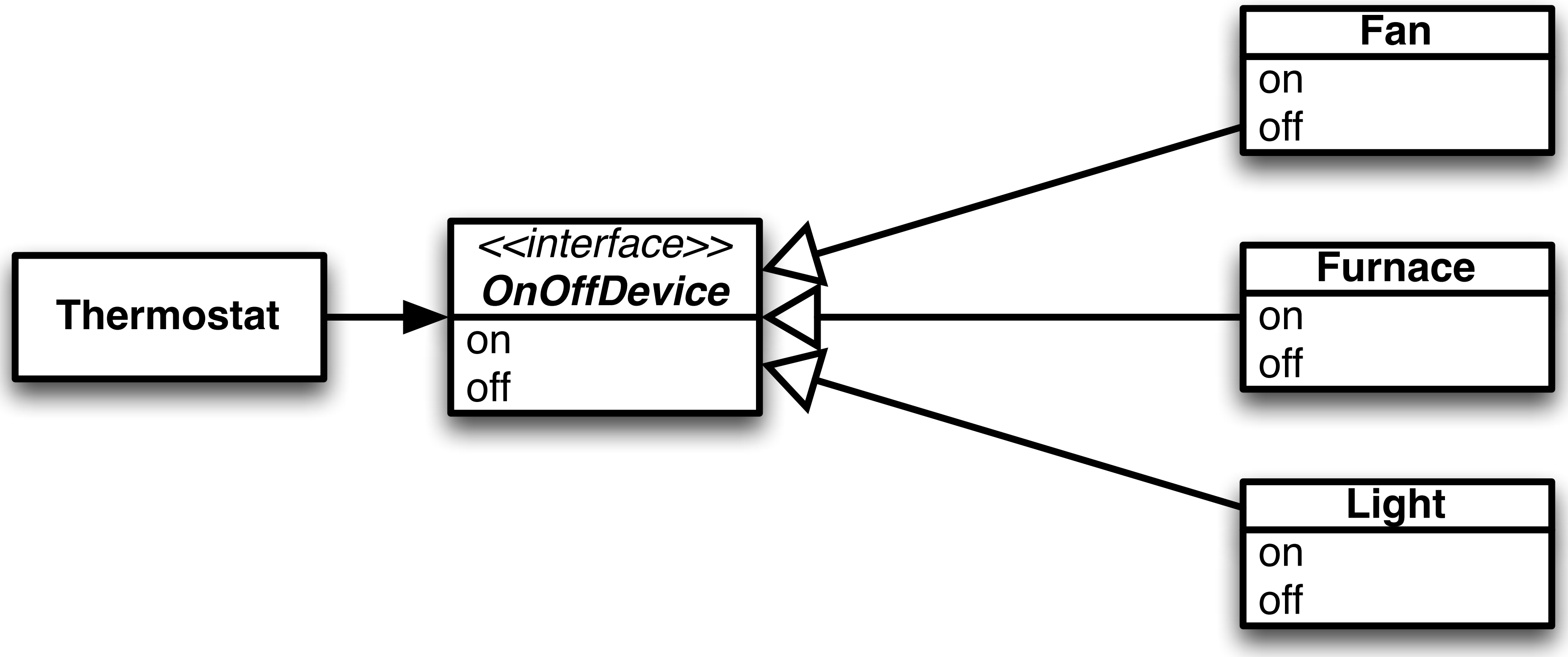
```
public interface ISwitchableDevice
{
  void on();
  void off();
}
```



```
public class Thermostat
{
  private ISwitchableDevice _device;

  public Thermostat(ISwitchableDevice device)
  {
    _device = device;
  }
```



```
public class Furnace : ISwitchableDevice
{
  public Furnace()
  {
    off();
  }

  public void on()
  {
    Debug.WriteLine("Furnace is ON");
  }

  public void off()
  {
    Debug.WriteLine("Furnace is OFF");
  }
}
```


Thermostat
<<interface>>
OnOffDevice
on
off
Fan
on
off
Furnace
on
off
Light
on
off

<lab>

Brownfield Development

Can we benefit?

Can we benefit?

We **can** improve design going forward

The goal is writing **working code**/providing **value**

How to start

- Test KEY use cases
- Test defects
- Test new features

Refactor

- Discover the intent
- Isolate Dependencies (Inversion of Control)
- Re-design for testability
- Introduce helpful abstractions
- Address anti-patterns

Refactor

Key anti-patterns

- Magic Numbers
- Long Methods
- Long Class
- Poor Naming
- Empty Catches
- Similar Code
- Unclear Tests
- Large Tests

<lab>

Shawn Wallace

shawn.wallace@centricconsulting.com

http://about.me/shawnwallace

@ShawnWallace

614-270-1600