PRACTICAL TEST DRIVEN DEVELOPMENT

Refactoring your applications towards testing

WHO IS TIM RAYBURN?

- Principal Consultant with Improving Enterprises
- Organizer of Dallas TechFest
- Blogger
- Gamer
- Troublemaker

improving
It's what we do.

EXPECTATIONS OF THE AUDIENCE

- You are familiar with:
  - Test Driven Development
  - Dependency Injection
  - Inversion of Control
  - Mocking

QUICK REVIEW : TDD

- Test Driven Development
  - The practice of writing automated unit tests which exercise just the code you are writing without concern for its place in your system.
  - Focuses development first on how a component will be consumed through the practice of **Test First**.
  - Small iterative cycles : **Red**, **Green**, **Refactor**

QUICK REVIEW : DI

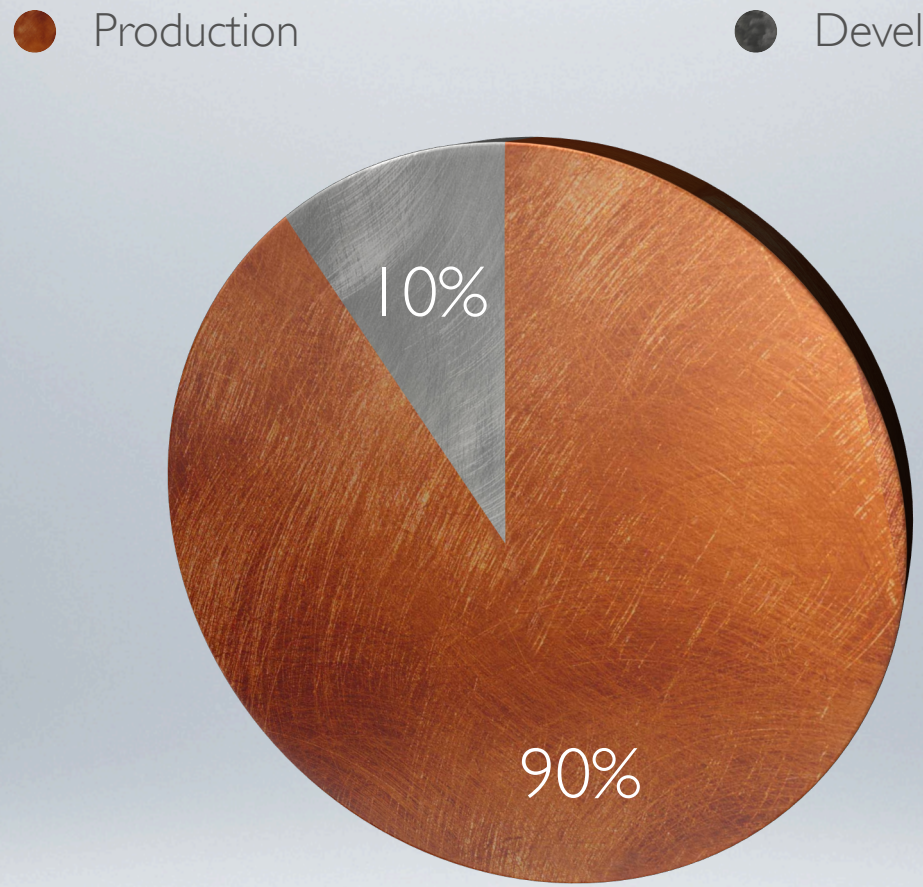
- Dependency Injection
  - The practice of allowing classes which you depend on for functionality to be "injected" into your class.
  - The most common forms of injection are constructor and property injection.
  - By injecting dependencies, when testing you can replace actual dependencies with **Mocks**.

QUICK REVIEW : IOC

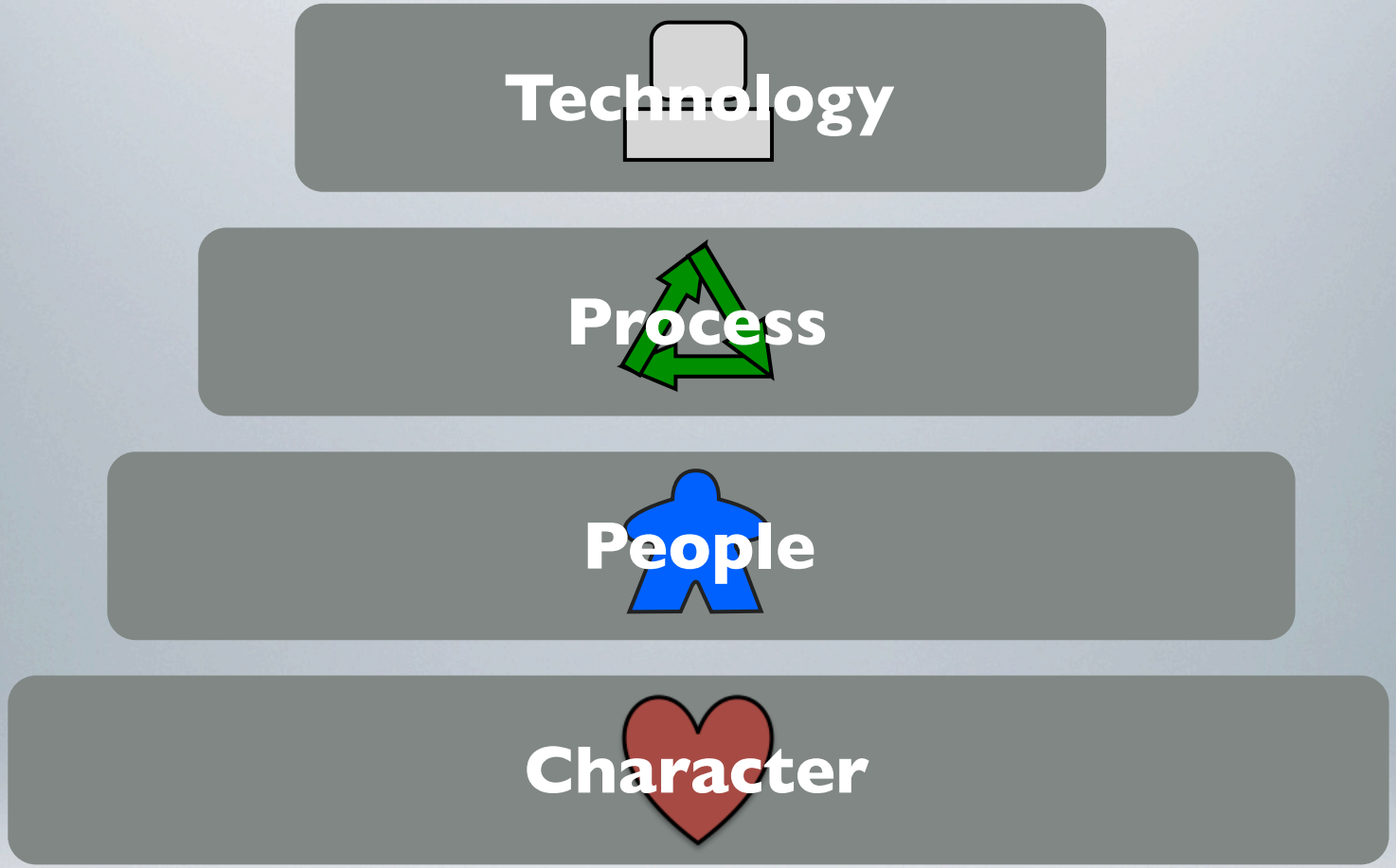
- Inversion of Control Containers
  - Component which handles construction of instances which have large numbers of dependent classes.
  - There are many containers available, most are open source. Popular frameworks include : Castle Windsor, Ninject, Structuremap, and others.
  - Usually IoC is mapping Interfaces to Concrete classes in the process of construction.

QUICK REVIEW : MOCKING

- Mocking
  - The practice of replacing dependencies with stub implementations which satisfy the needs of one or more tests.
  - There are many mocking frameworks, but one of these is far more popular than the rest, **Rhino.Mocks**.
  - Rhino.Mocks makes it very easy to mock an interface.



YOUR APP'S LIFETIME



MOST PROBLEMS RELATE TO...

TEST FIRST IS HARD

The Problem

- We want to follow best practices, which would mean **Red**, **Green**, **Refactor**.
- Unfortunately that is very different from the normal workflow.

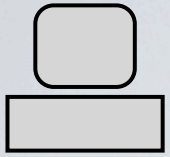
TEST FIRST IS HARD

A Solution

- Suck it up!
- Consider using the Given/When/Then structure for your tests.

TEST FIRST IS HARD

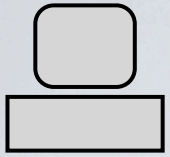
Demo



NO IOC CONTAINER

The Problem

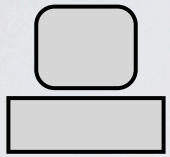
- Most TDD talks assume that you have a simple, container driven approach.
- Existing projects rarely already have IoC containers.
- We want to enable Dependency Injection without requiring a total refactoring of every class.
- We'd prefer if once we have incrementally reached a proper level of inject-ability that we are not stuck with remnants.



NO IOC CONTAINER

A Solution

- We can reach all of the desired by **wisely** using the Service Locator pattern.
- Service Locator can get a bad reputation from purists, because they feel so strongly about inject-ability.
- Our solution will use constructor injection, and constructor overloading.



NO IOC CONTAINER

Demo

HUGE BACKLOG

The Problem

- To create tests that cover the whole of our already developed application.
- Contrary to **Test First** but necessary because that ship has sailed.

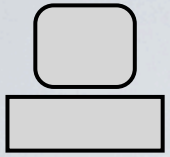
HUGE BACKLOG

A Solution

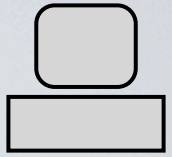
- You break it, you bought it.
  - A philosophy that is useful on teams with this problem which states that if your current feature requires you to modify a method, then you are responsible for creating at least one "happy path" test for that method.
  - If and only if there are no reasonable "happy path" tests still to be written, then you can write an exception case.
  - If none of those exist, congratulations it is a tested class.

HUGE BACKLOG

Are you nuts Tim?

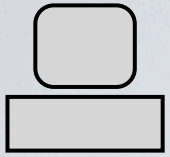


DEPENDENCIES

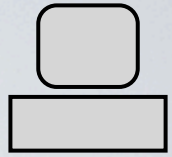


The Problem

- A class has a huge number of dependencies, some of which we have no control over.
- The class is amazingly poorly factored, and is likely violating multiple of the SOLID principles.
- You break it, you bought it is not going to work on this 5,000 line method.
- Yet we want to add functionality to it which is tested.

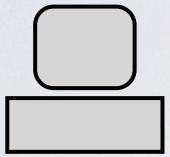


DEPENDENCIES

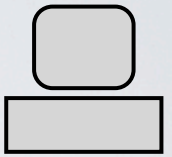


A Solution

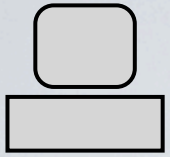
- First, as painful as it may be, **seriously** consider re-writing this class, breaking it down into reasonably factored classes each of which has a **Single Responsibility**.
- Failing that, create a new class which encapsulates only the desired change, and then inject that class into the monster class.
- Call out to this class at the appropriate places in the monster class.



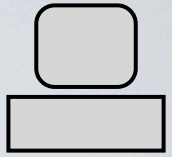
DEPENDENCIES



Demo

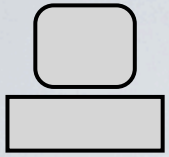


STATIC

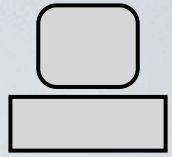


The Problem

- My product is littered with **static** methods, **static** types, and **static** everything.
- You can't apply interfaces to static, so all these polymorphic tricks used in TDD don't net me any testability.



STATIC



A Solution

- First, as painful as it may be, **seriously** consider re-writing this class, breaking it down into reasonably factored classes each of which has a **Single Responsibility**.
- Failing that, create a new class which encapsulates only the desired change, and then inject that class into the monster class.
- Call out to this class at the appropriate places in the monster class.

TypeMock Isolator

Questions?

improving

It's what we do.

tim@timrayburn.net

timrayburn.net