Speaker style bingo: **10 presentation anti-patterns** (by Troy Hunt)

… These ten behaviours undoubtedly limit the speaker's ability to perform meaningful, engaging and memorable talks. But they're still talking – they're up there putting their views forth and contributing in a way that most people don't…

From there on **it's just continual refinement** regardless of how long you've been doing it or how slick your style is.

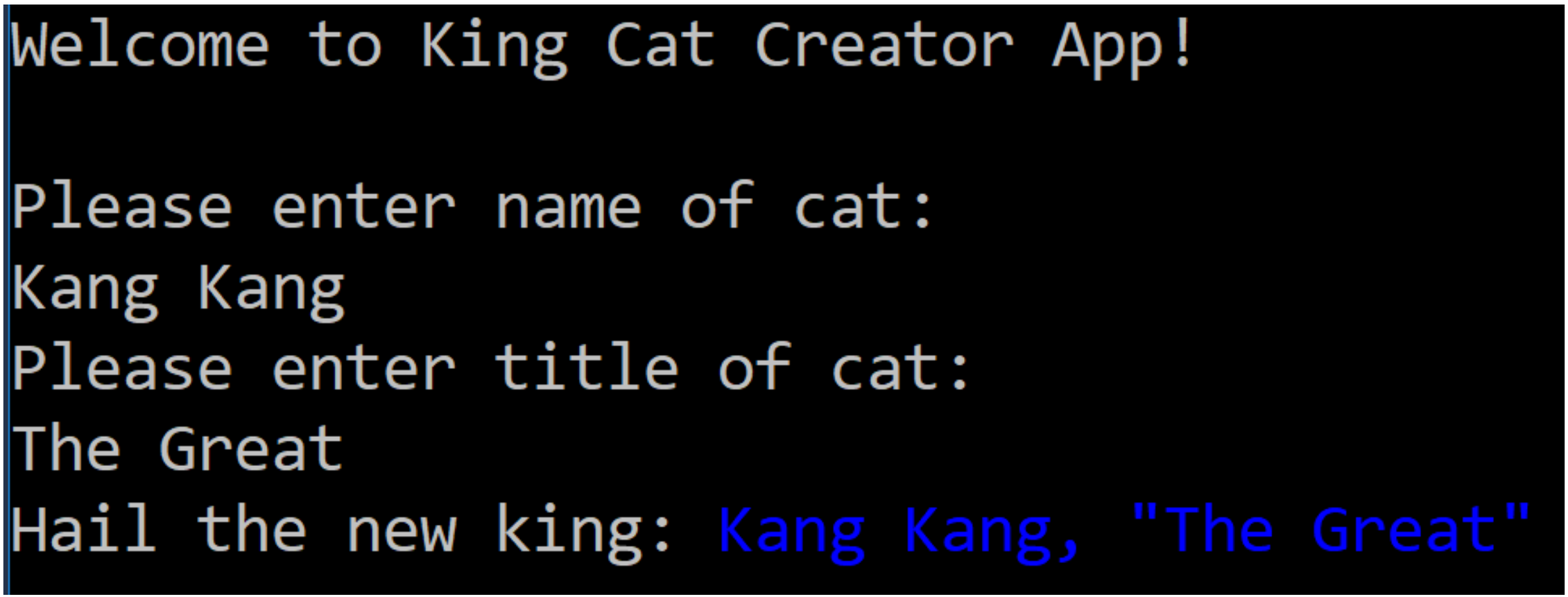
1 kilo of cotton

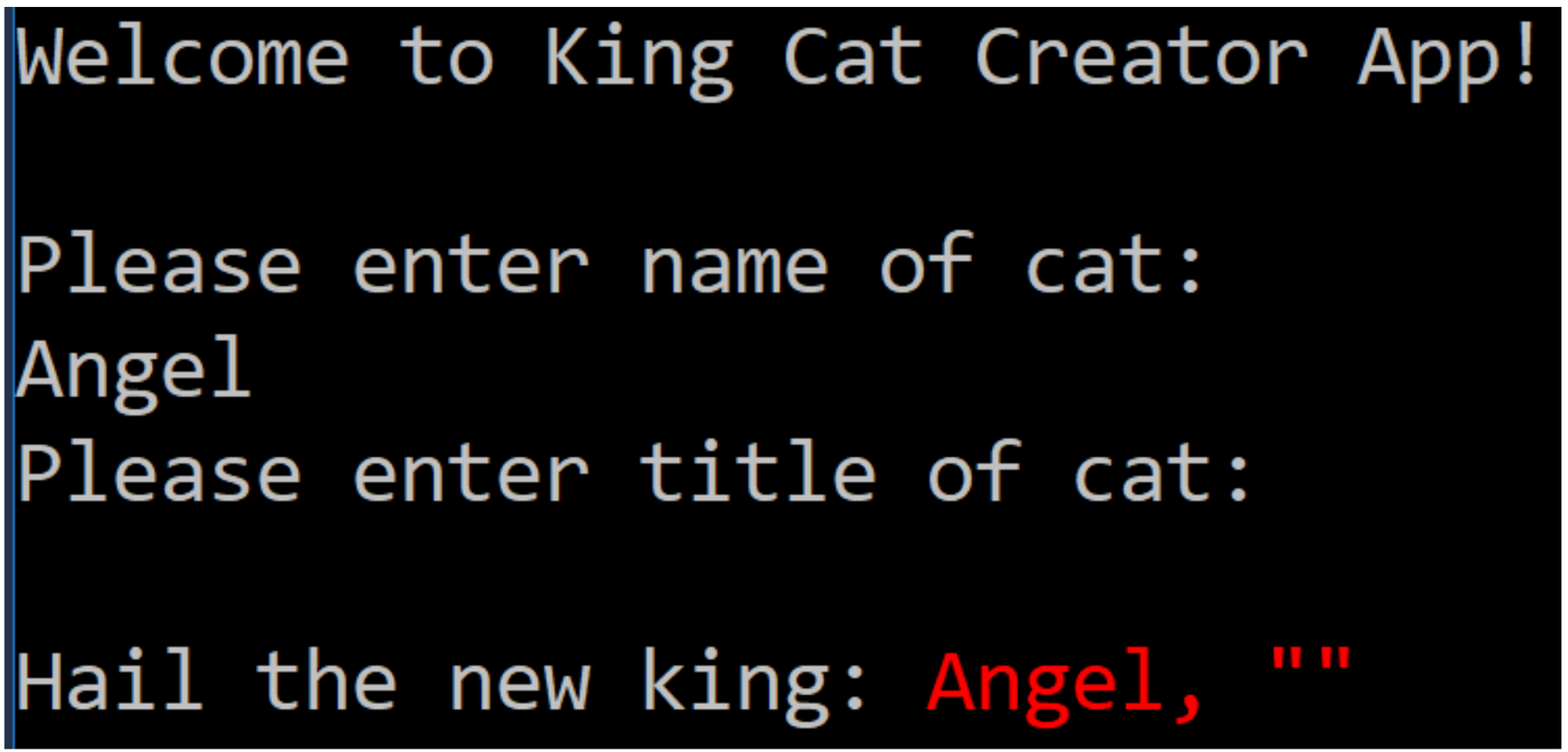
1 kilo of nails

SD card
w/o data

SD card
with data

“In ancient times, cats were worshiped as gods. They have never forgotten this.”


Welcome to King Cat Creator App!
Please enter name of cat:
Kang Kang
Please enter title of cat:
The Great
Hail the new king: Kang Kang, "The Great"


Welcome to King Cat Creator App!
Please enter name of cat:
Angel
Please enter title of cat:
Hail the new king: Angel, ""

Create Cat Use Case

Data:

<Name> (“Kang Kang”)

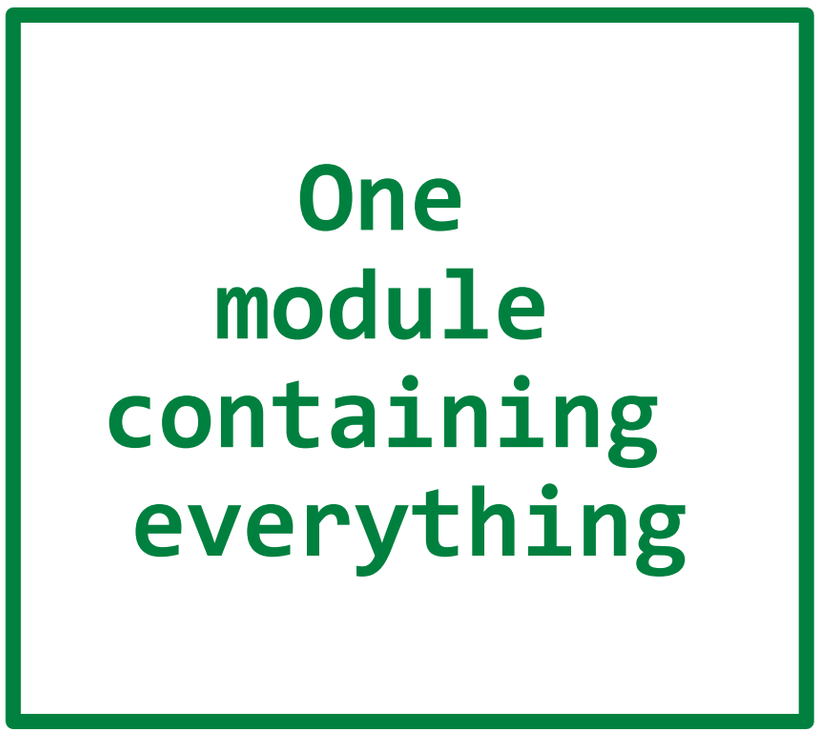
<Title> (“The Great”)

Primary Course:

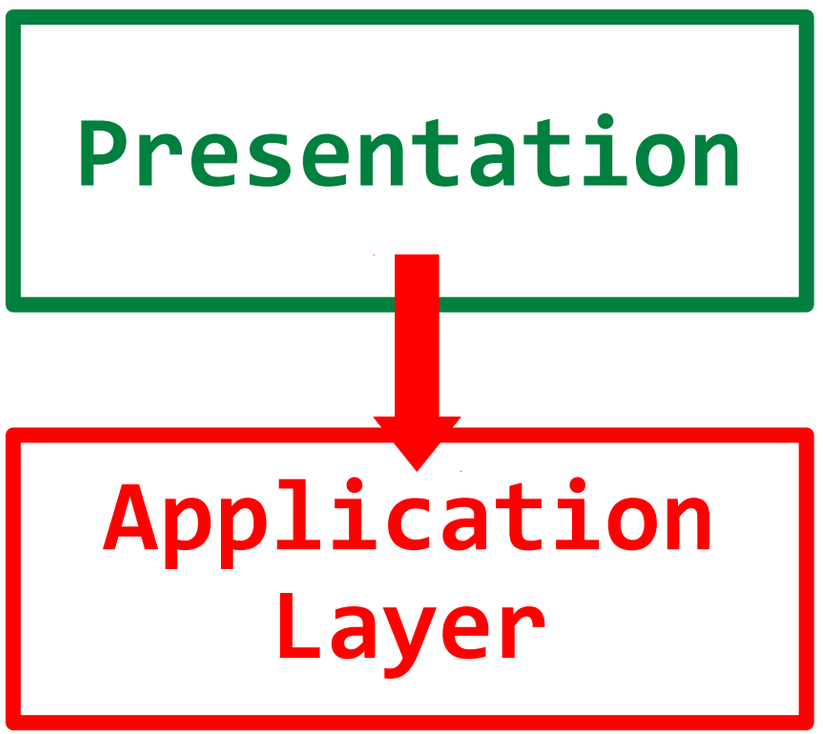
1. User issues “Create Cat” command with above data.
2. System creates new Cat
3. System displays Grandiose-Name of new Cat

Exception Course: Validation Error

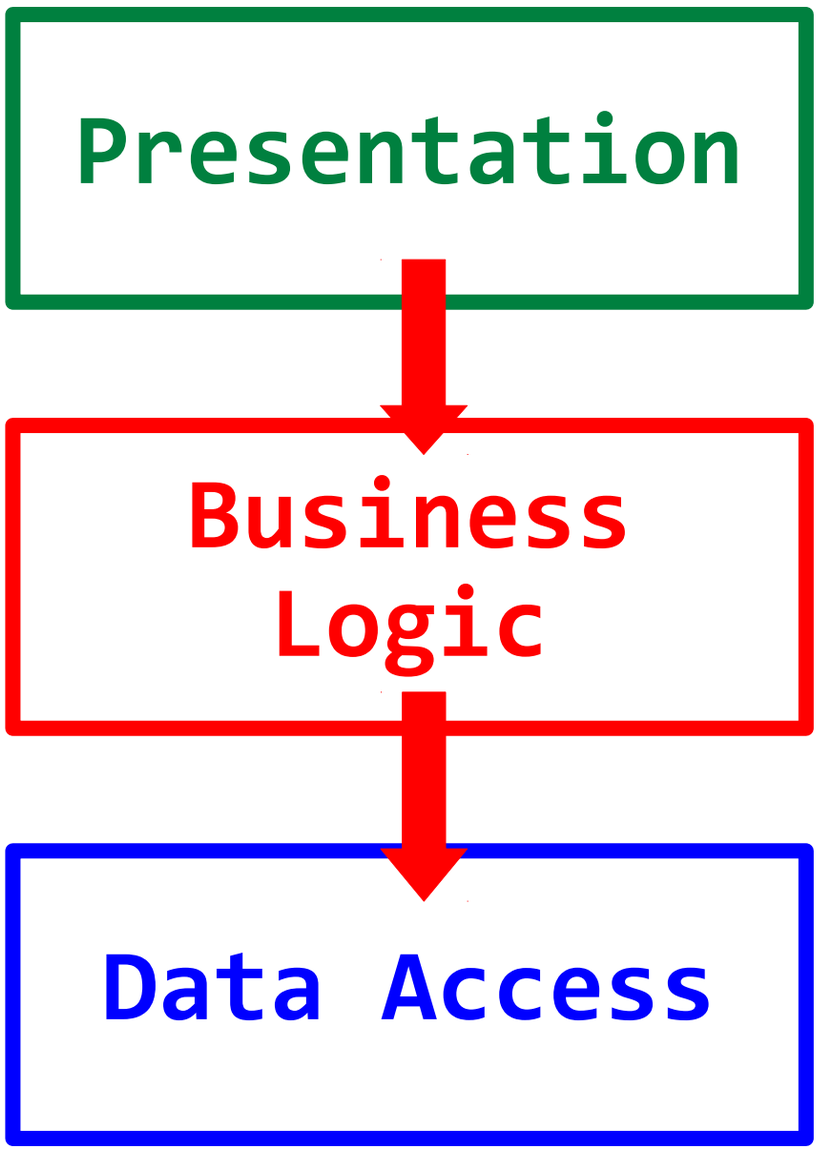
1. System delivers error message



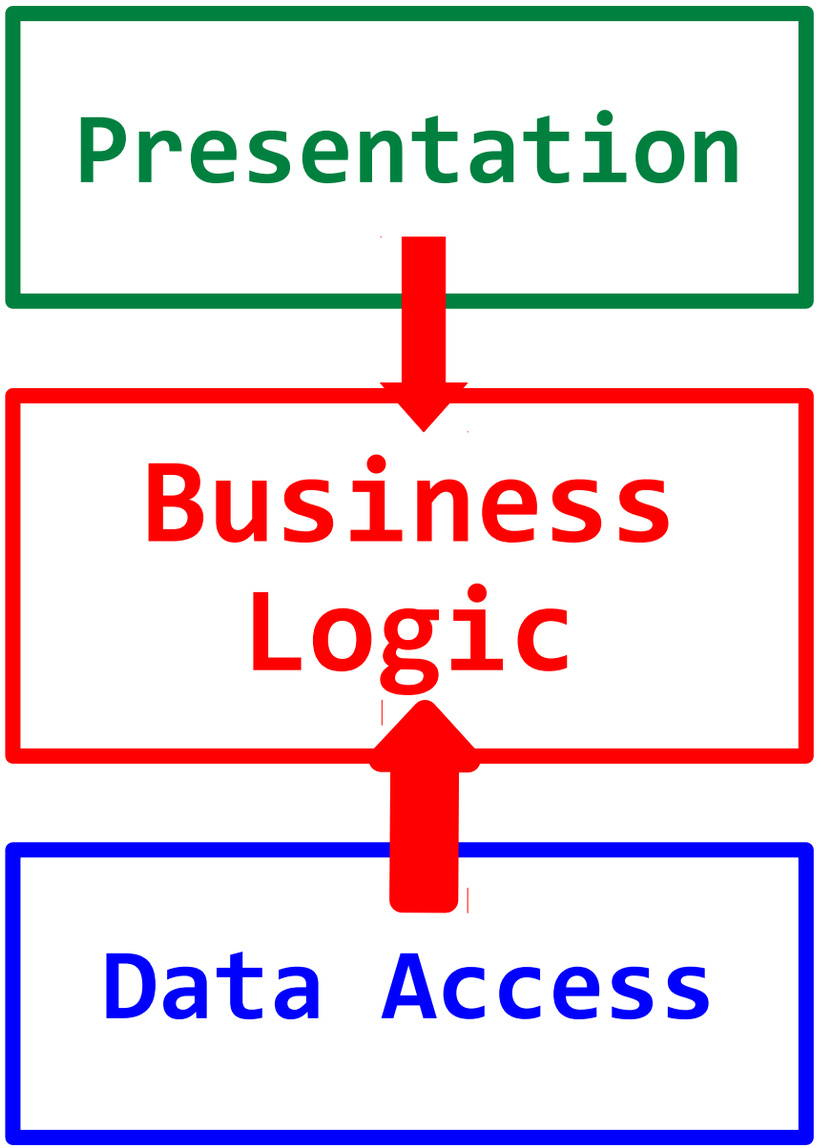
Monolith



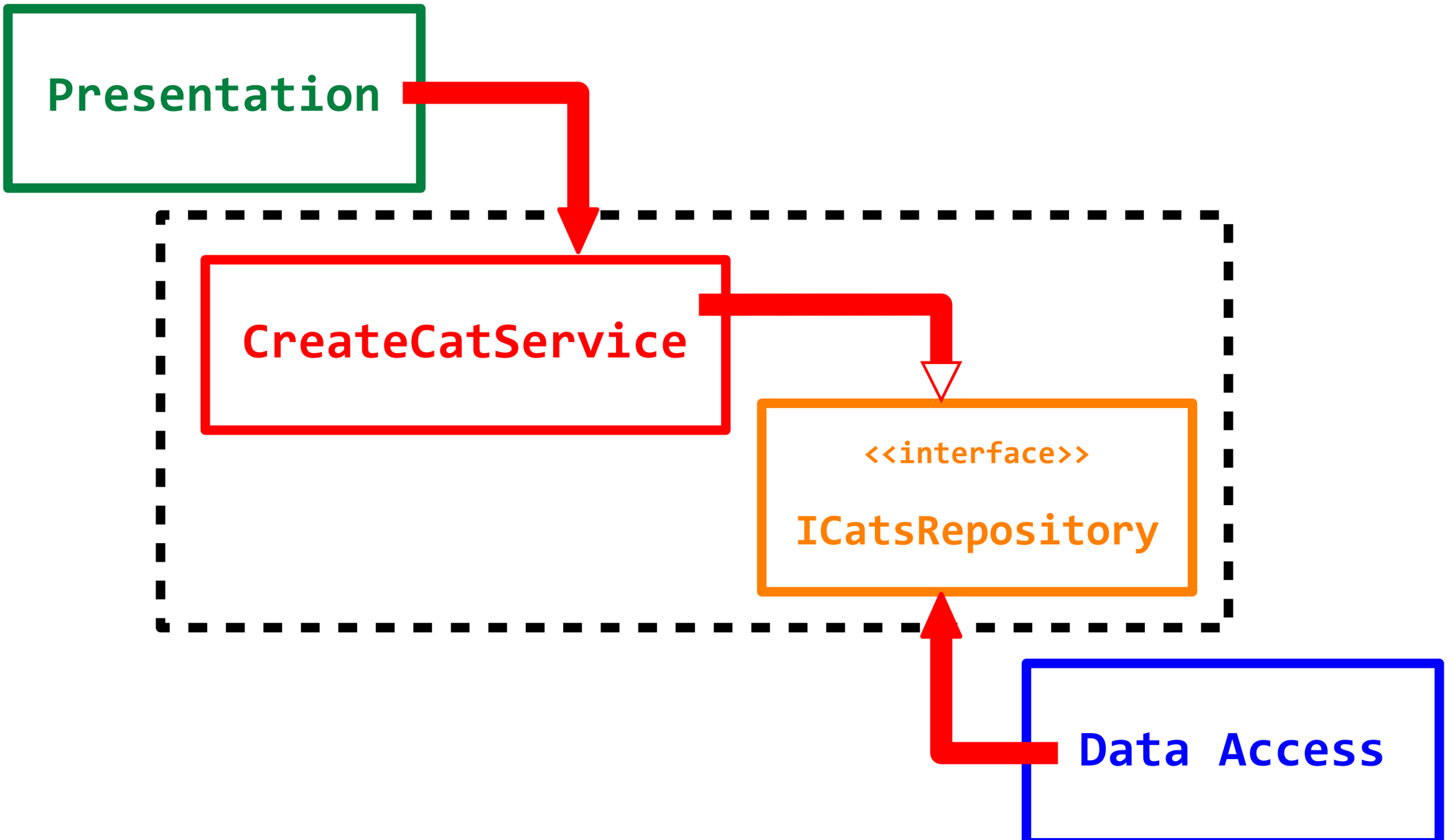
Traditional Layered Architecture

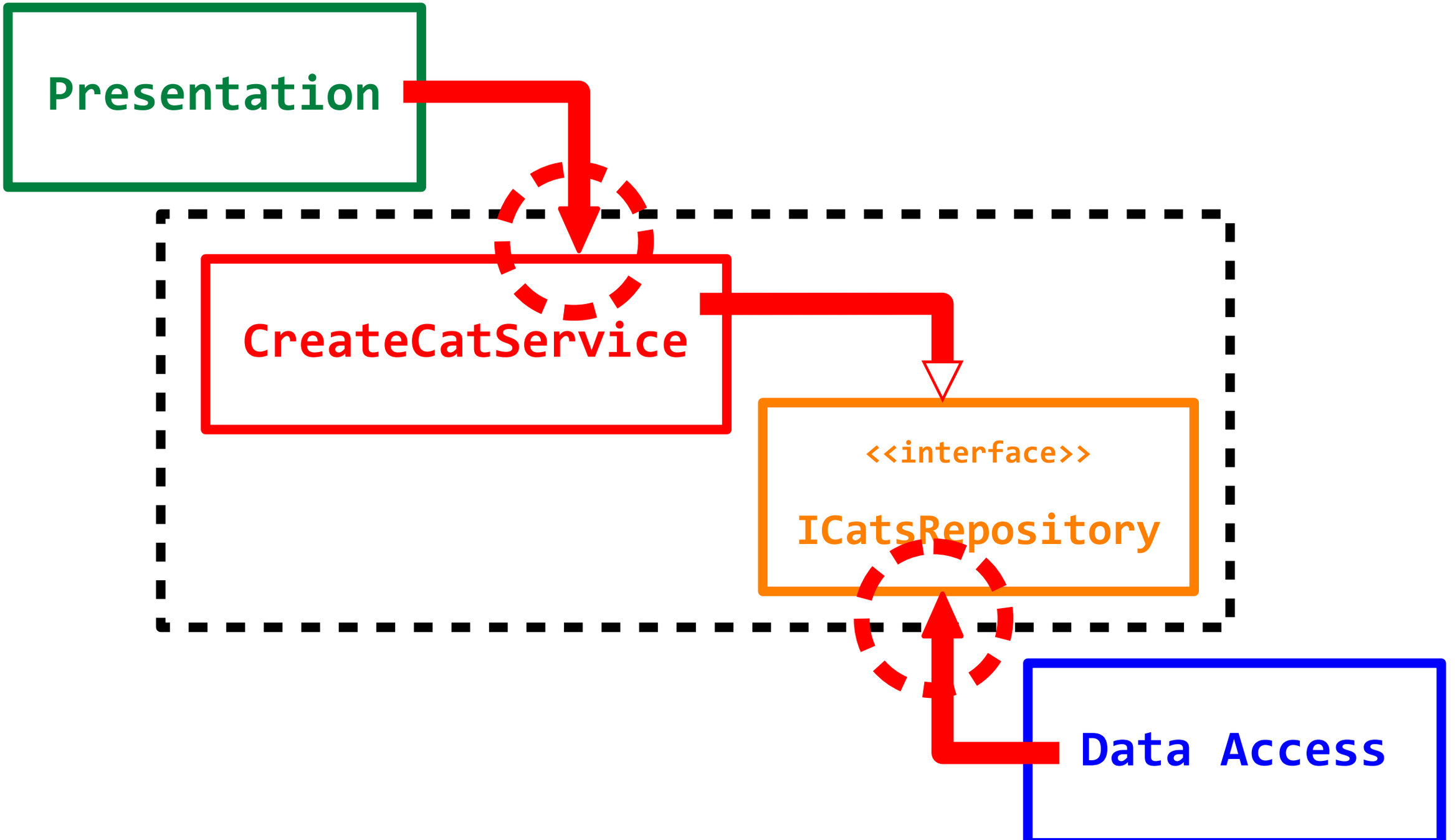


Traditional Layered Architecture

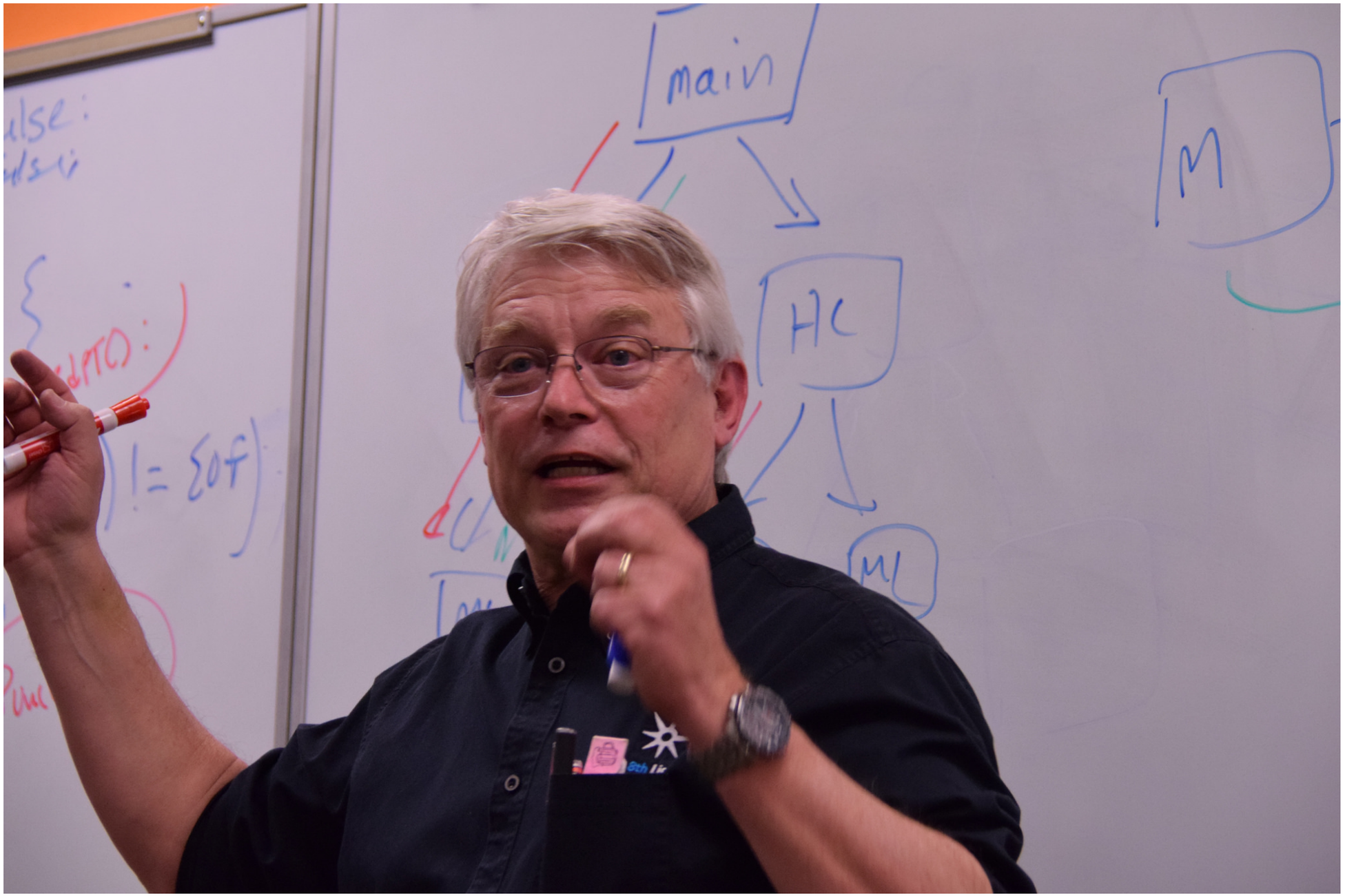


Clean/Decoupled/Plugin Architecture

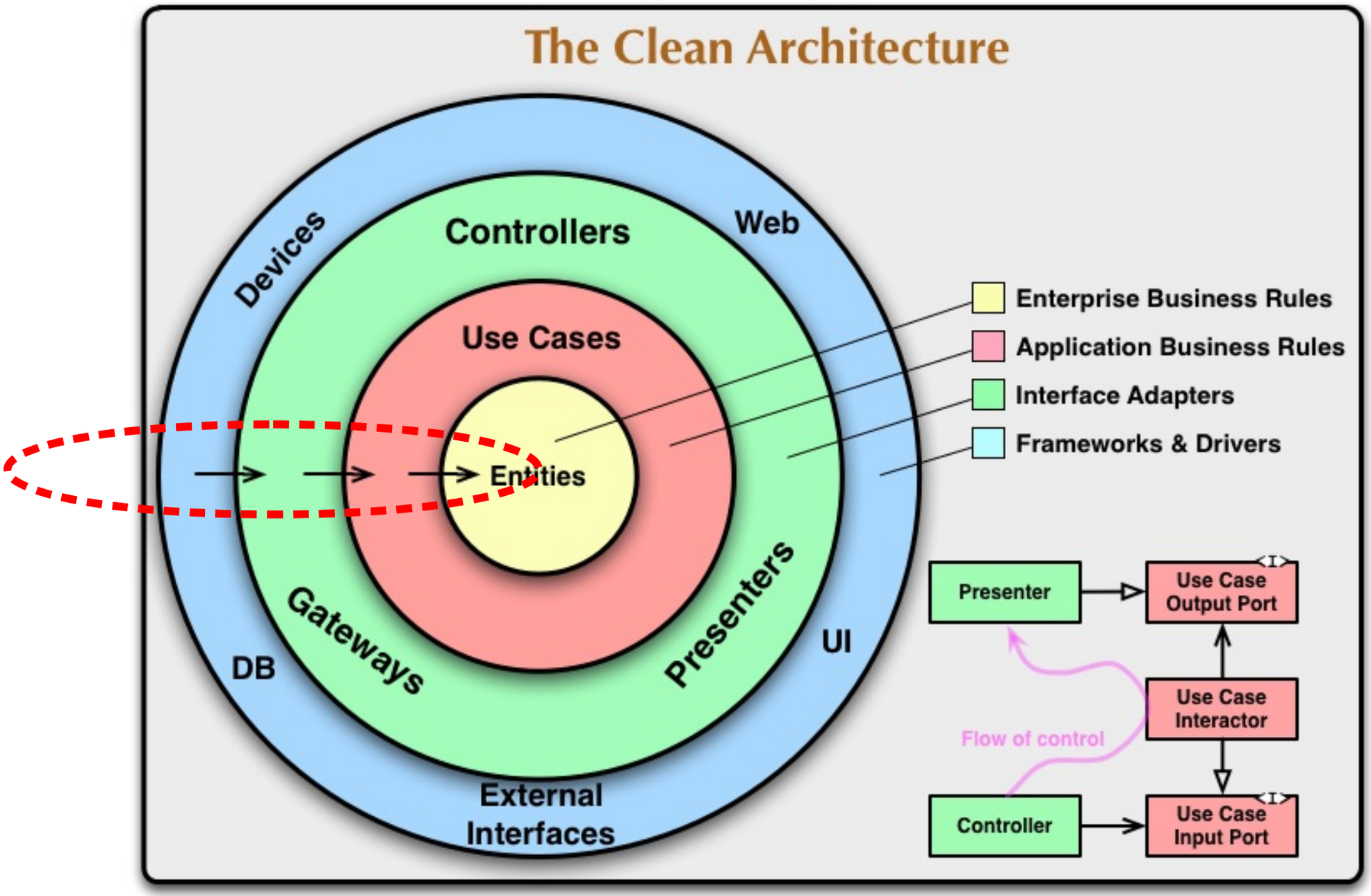

Presentation
CreateCatService
<<interface>>
ICatsRepository
Data Access


Presentation
CreateCatService
<<interface>>
ICatsRepository
Data Access

Clean Architecture



Source: https://tinyurl.com/y9649e6v



http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Software was invented to be “***soft***”.

- Uncle Bob Martin
(from Clean Architecture: A Craftsman's Guide to Software Structure and Design)

Customers value two things about software.

1. The way it makes a machine behave;

2. The ease with which it can be changed. - ***"soft"***

- Uncle Bob Martin
(from "The Cycles of TDD" article)

I'm keenly aware of **the Principle of Priority**, which states

(a) you must know the difference between what is **urgent** and what is **important**, and

(b) you must do what's important first.

- Steven Pressfield (from The War of Art)

Customers value two things about software.

1. The way it makes a machine behave;

2. The ease with which it can be changed. - ***"soft"***

- Uncle Bob Martin
(from "The Cycles of TDD" article)

Customers value two things about software.

URGENT

1. The way it makes a machine behave;

2. The ease with which it can be changed. - ***"soft"***

IMPORTANT

- Uncle Bob Martin

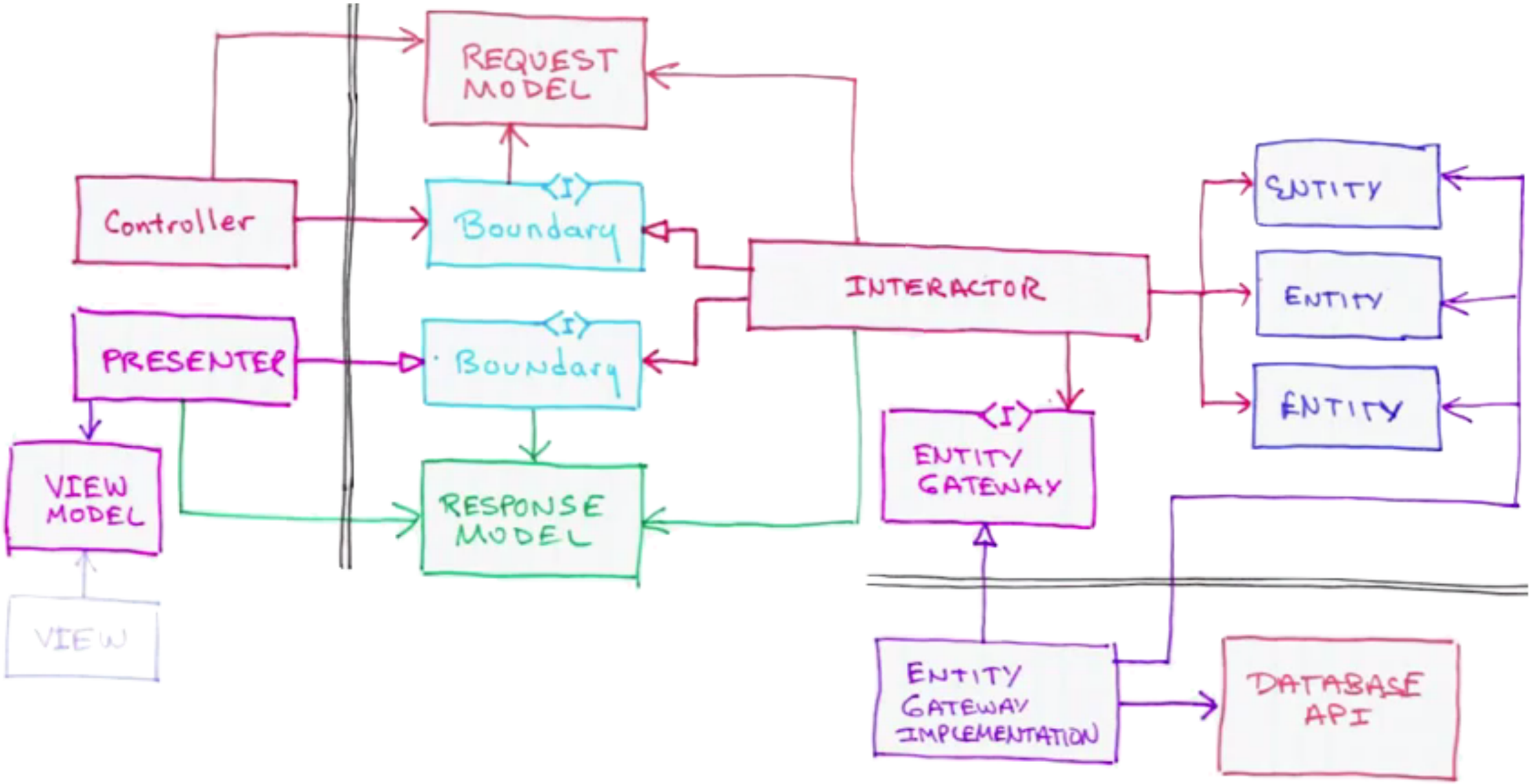
(from "The Cycles of TDD" article)

Every software system provides two different values to the stakeholders:

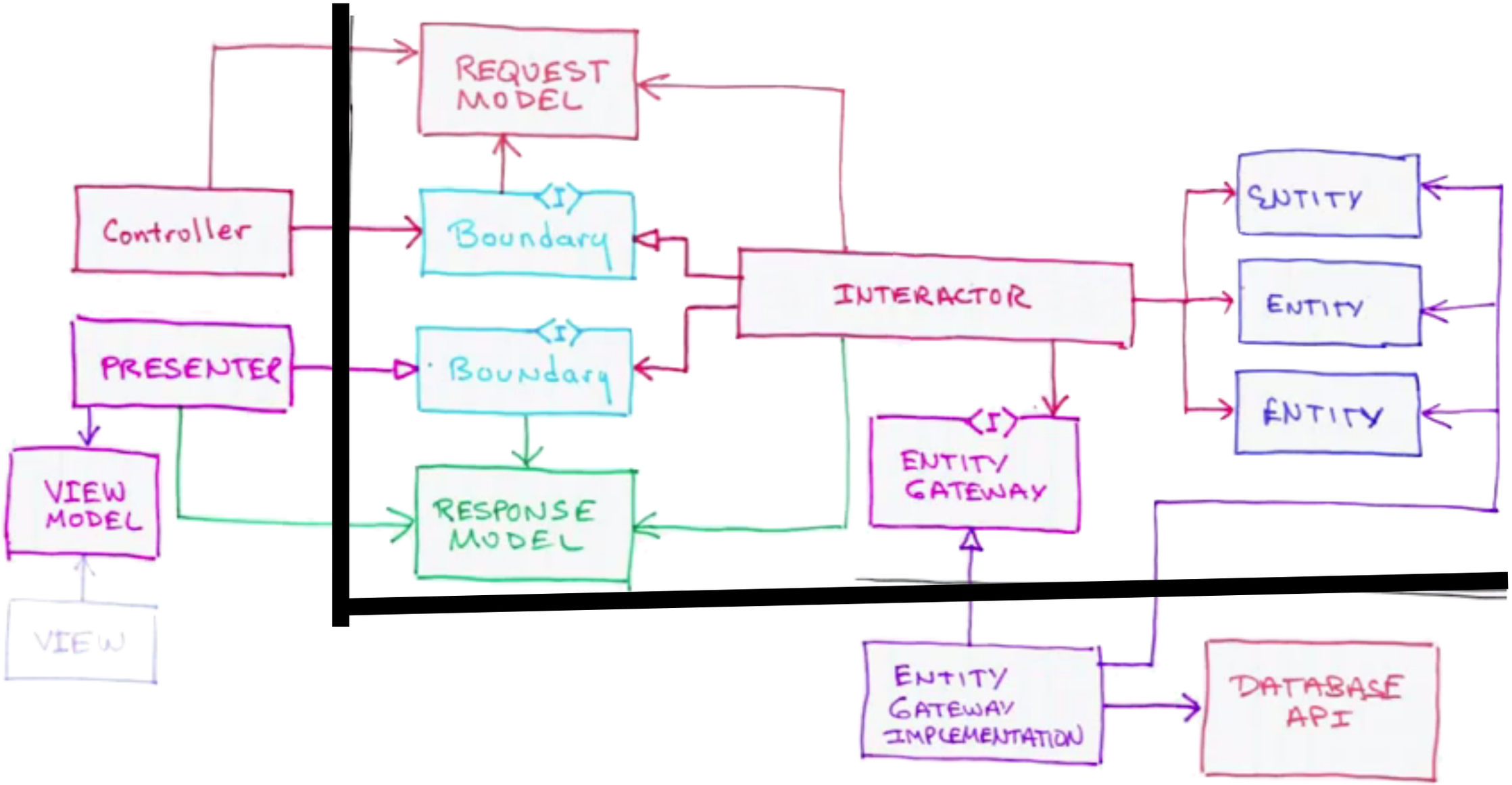
The first value of software — **behavior — is urgent** but not always particularly important.

The second value of software — **architecture — is important** but not particularly urgent.

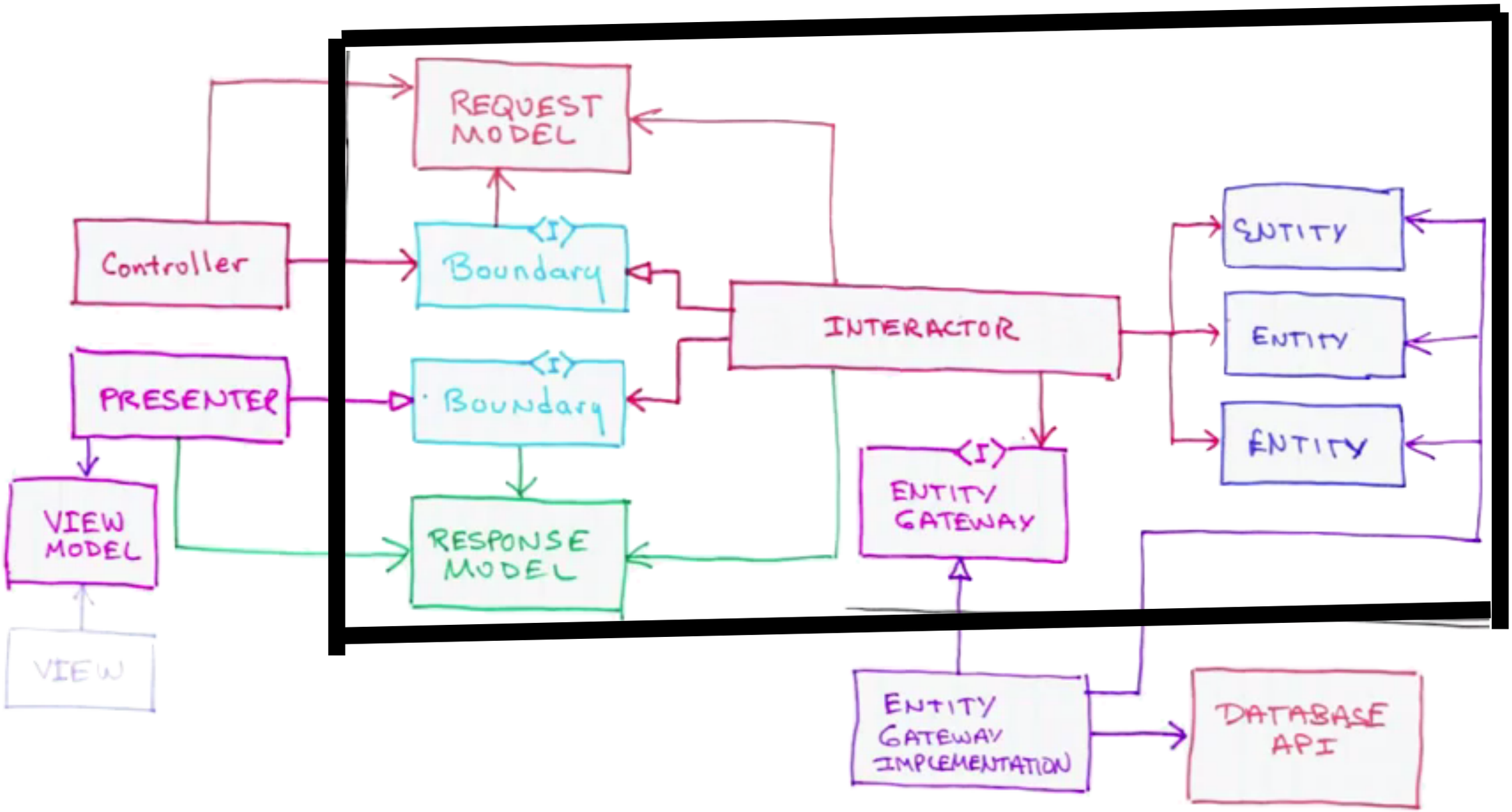
- Uncle Bob Martin
(from Clean Architecture: A Craftsman's Guide to Software Structure and Design)



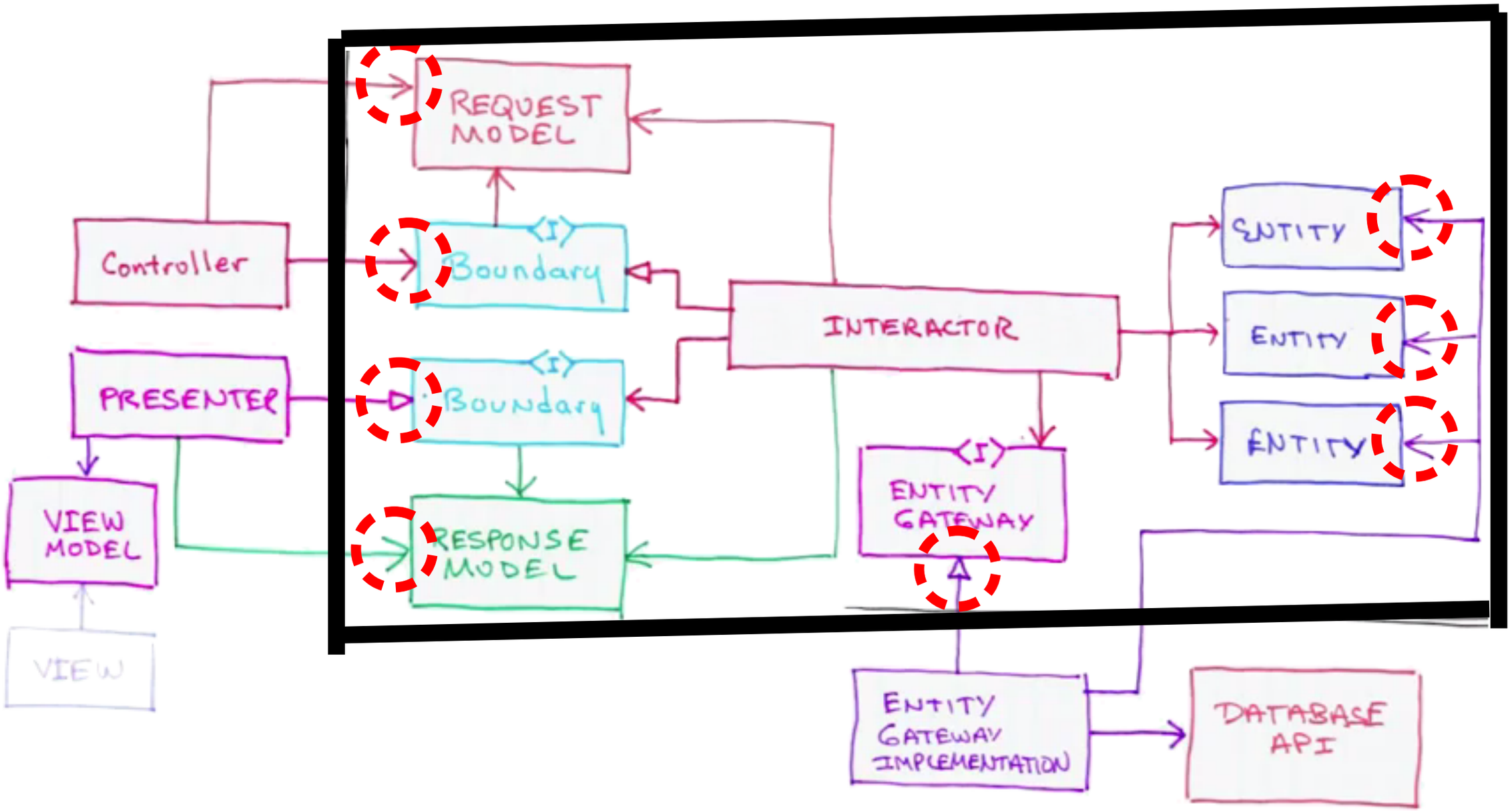
From Uncle Bob's slides of his talks on Clean Architecture



From Uncle Bob's slides of his talks on Clean Architecture



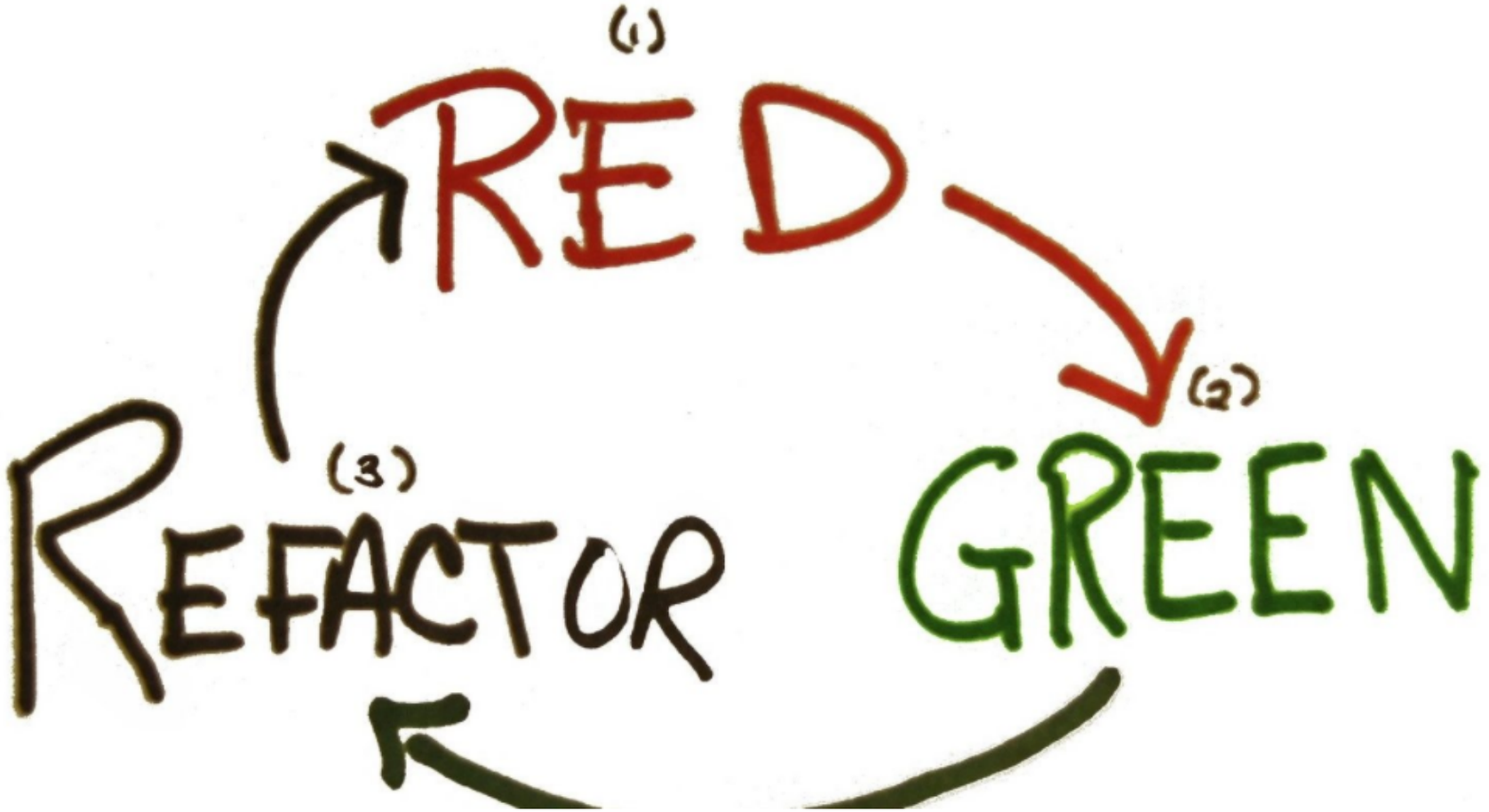
From Uncle Bob's slides of his talks on Clean Architecture



From Uncle Bob's slides of his talks on Clean Architecture

TDD

Test-Driven Development



From http://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html

```
public class GreetingCreatorTests
{

    [Fact]
    public void should_create_correct_greeting()
    {
        // arrange
        var greetingCreator = new GreetingCreator();

        // act
        string greeting =  greetingCreator.create("World");

        // assert
        Assert.Equal("Hello, World!", greeting);
    }
}
```

Example

```
public class GreetingCreatorTests
{

    [Fact]
    public void test_that_the_tests_are_working()
    {
        Assert.Equal(4, 2 + 2);

    }
}
```

```
public class GreetingCreatorTests
{

    [Fact]
    public void should_create_correct_greeting()
    {
        // arrange
        var greetingCreator = new GreetingCreator();


    }
}
```

```
public class GreetingCreator
{

}
```

```
public class GreetingCreatorTests
{

    [Fact]
    public void should_create_correct_greeting()
    {
        // arrange
        var greetingCreator = new GreetingCreator();


    }
}
```

```
public class GreetingCreatorTests
{

    [Fact]
    public void should_create_correct_greeting()
    {
        // arrange
        var greetingCreator = new GreetingCreator();

        // act
        string greeting =  greetingCreator.Greet("Jboy");


    }
}
```

```
public class GreetingCreator
{
    public string Greet(string name)
    {
        return null;
    }

}
```

```
public class GreetingCreatorTests
{

    [Fact]
    public void should_create_correct_greeting()
    {
        // arrange
        var greetingCreator = new GreetingCreator();

        // act
        string greeting =  greetingCreator.Greet("Jboy");


    }
}
```

```
public class GreetingCreatorTests
{

    [Fact]
    public void should_create_correct_greeting()
    {
        // arrange
        var greetingCreator = new GreetingCreator();

        // act
        string greeting =  greetingCreator.Greet("Jboy");

        // assert
        Assert.Equal("Hello, Jboy!", greeting);

    }
}
```

I wish the .Create() command returns the correct greeting

```
public class GreetingCreator
{
    public string Greet(string name)
    {
        return "Hello, Jboy!";
    }

}
```

```
public class GreetingCreatorTests
{

    [Fact]
    public void should_create_correct_greeting()
    {
        // arrange
        var greetingCreator = new GreetingCreator();

        // act
        string greeting =  greetingCreator.Greet("Jboy");

        // assert
        Assert.Equal("Hello, Jboy!", greeting);

    }
}
```

It’s working!

We’re done.

```
public class GreetingCreator
{
    public string Greet(string name)
    {
        return "Hello, Jboy!";
    }

}
```

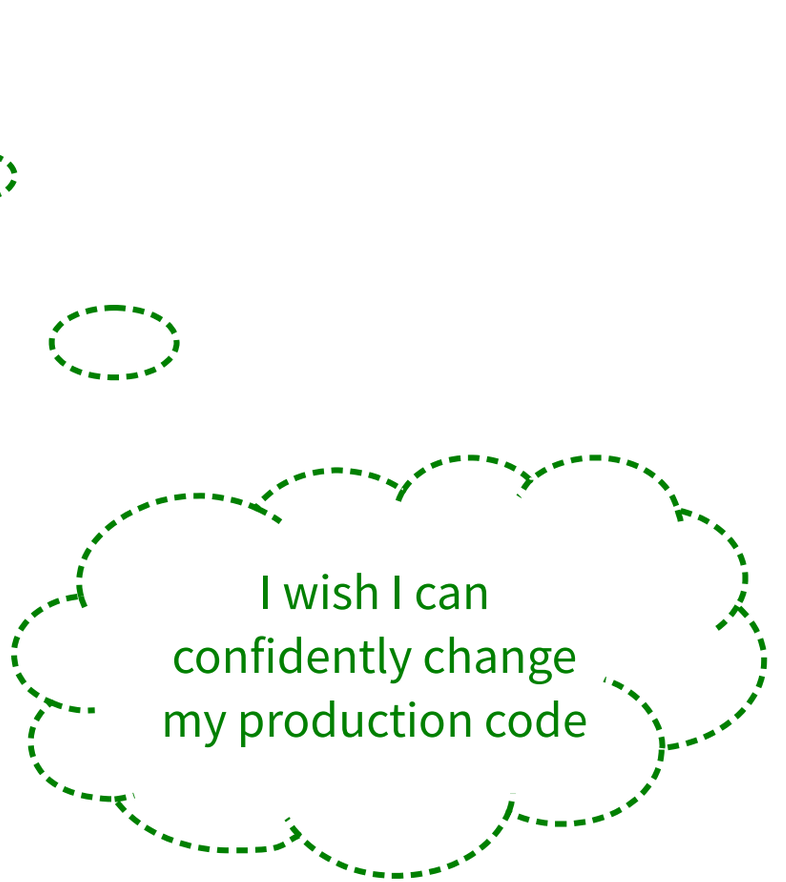
```
public class GreetingCreator
{
    public string Greet(string name)
    {
        return "Hello, " + name;
    }

}
```

```
public class GreetingCreatorTests
{

    [Fact]
    public void should_create_correct_greeting()
    {
        // arrange
        var greetingCreator = new GreetingCreator();

        // act
        string greeting =  greetingCreator.Greet("Jboy");

        // assert
        Assert.Equal("Hello, Jboy!", greeting);

    }
}
```

```
public class GreetingCreator
{
    public string Greet(string name)
    {
        return "Hello, " + name + "!";
    }

}
```

```
public class GreetingCreatorTests
{

    [Fact]
    public void should_create_correct_greeting()
    {
        // arrange
        var greetingCreator = new GreetingCreator();

        // act
        string greeting =  greetingCreator.Greet("Jboy");

        // assert
        Assert.Equal("Hello, Jboy!", greeting);

    }
}
```

```
public class GreetingCreator
{
    public string Greet(string name)
    {
        return string.Format("Hello, {0}!", name);
    }

}
```



```
public class GreetingCreatorTests
{

    [Fact]
    public void should_create_correct_greeting()
    {
        // arrange
        var greetingCreator = new GreetingCreator();

        // act
        string greeting =  greetingCreator.Greet("Jboy");

        // assert
        Assert.Equal("Hello, Jboy!", greeting);

    }
}
```

Why write tests first?

Source: https://tinyurl.com/y7rlue57

"... [TDD] practitioners discovered that **writing tests first made a significant improvement to the design process.**"

- Martin Fowler

https://martinfowler.com/articles/mocksArentStubs.html

Some TDD history

Kent Beck
Source: https://tinyurl.com/y7bb7knx

TDD was developed by Kent Beck in the late 1990's as part of Extreme Programming.

In essence you follow three simple steps repeatedly:

1. Write a test for the next bit of functionality you want to add.
2. Write the functional code until the test passes.
3. Refactor both new and old code to make it well structured.

- Martin Fowler

https://martinfowler.com/bliki/TestDrivenDevelopment.html

“The original XP folks were writing tests for everything that could break,

then they started writing the tests first.

Eventually we ended up at TDD.”

- David Astels

https://daveastels.com/2014/09/29/a-new-look-at-test-driven-development

Why Clean Architecture and TDD?

They can save us from becoming slaves of our software creations.

They help make keep our **software** from becoming **hardware**

They make our software
"soft" and **keep them**
"soft"

Where to go from here...

Blog posts of Uncle Bob on Clean Architecture and TDD (http://blog.cleancoder.com/)

- Screaming Architecture
- Clean Architecture
- The Clean Architecture
- A Little Architecture
- The Cycles of TDD
- TDD Harms Architecture
- Test First
- etc.

Talks of Uncle Bob on Clean Architecture

- Clean Architecture and Design
- Architecture: The Lost Years
- etc.

Book

- Clean Architecture: A Craftsman's Guide to Software Structure and Design

Examples

- [the] google [search engine] is your friend

“The only way to go **fast** is to go **well**”

- Uncle Bob Martin

Thank you!

Jeremiah (Jboy) Flaga

jboyflaga.github.io

Mynd Consulting