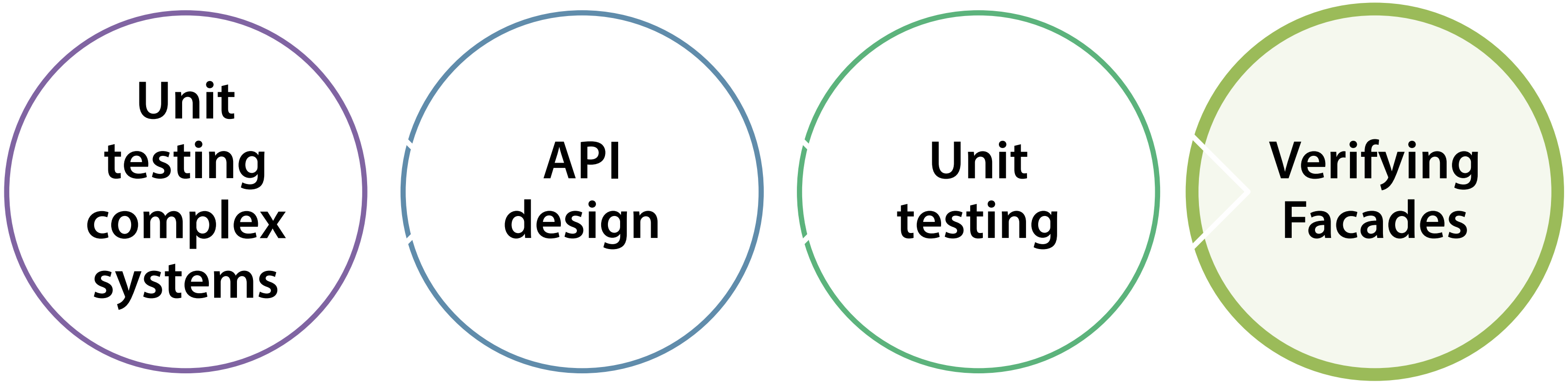
Advanced Unit Testing
Structural Inspection

Mark Seemann

http://blog.ploeh.dk

pluralsight
hardcore developer training

Outline



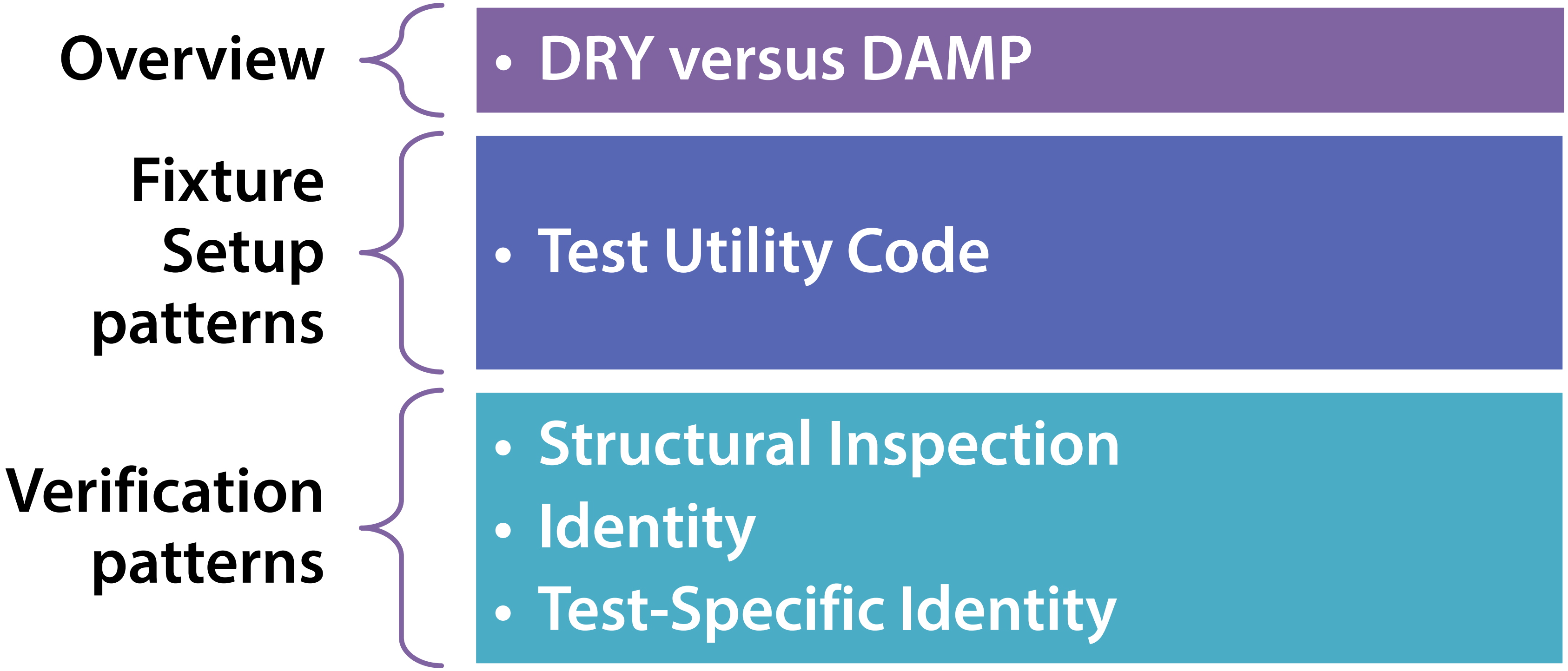
Unit testing complex systems

API design

Unit testing

Verifying Facades

Verification patterns

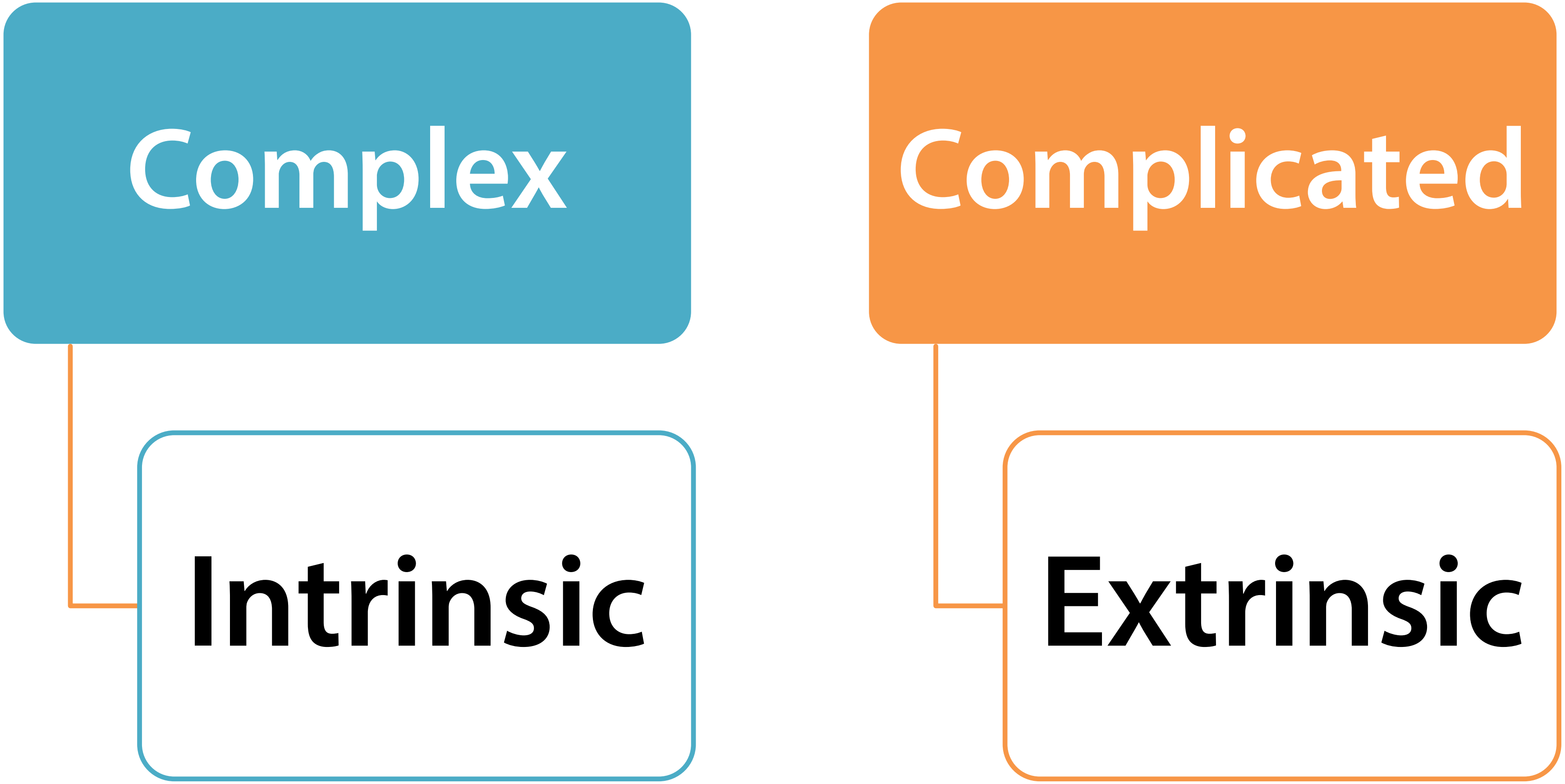


Overview	- DRY versus DAMP
Fixture Setup patterns	- Test Utility Code
Verification patterns	- Structural Inspection - Identity - Test-Specific Identity

How do you unit test a complex system?

How do you apply TDD against a complex system?

Digression



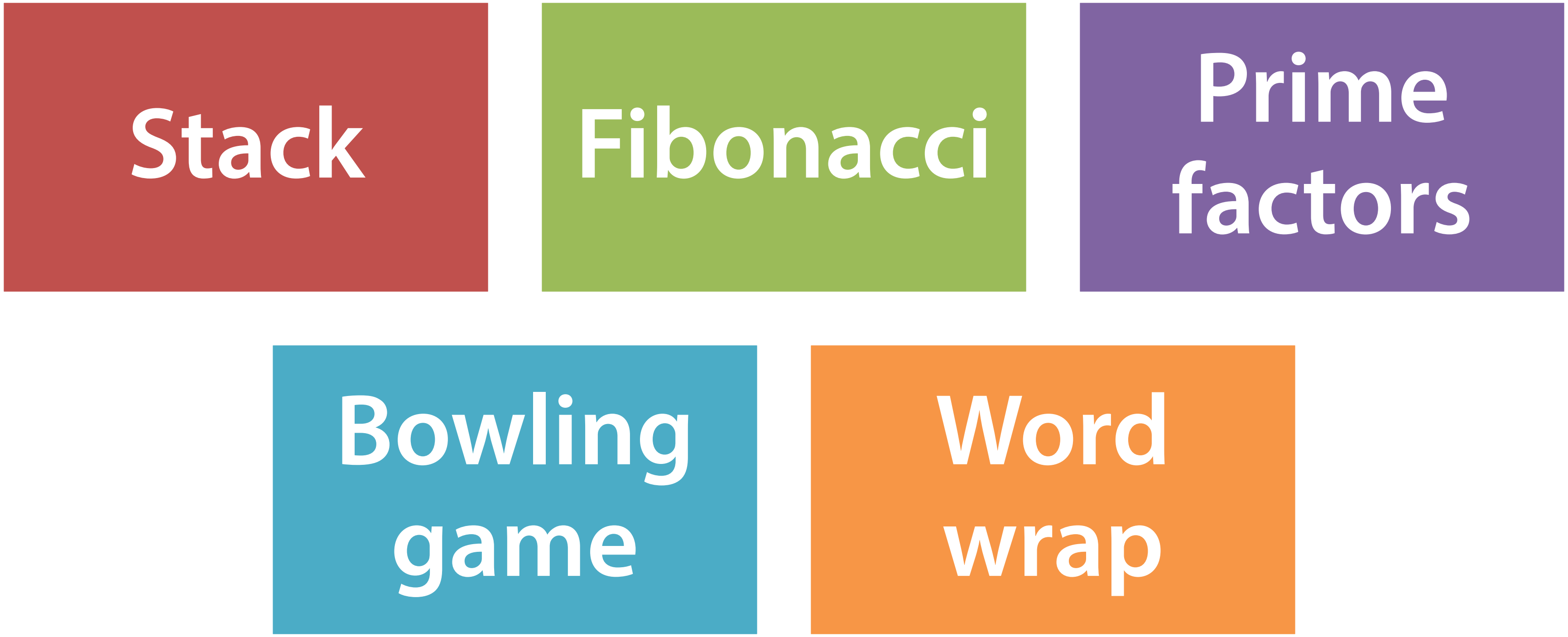
Complex

Intrinsic

Complicated

Extrinsic

Traditional introductions to TDD



Stack

Fibonacci

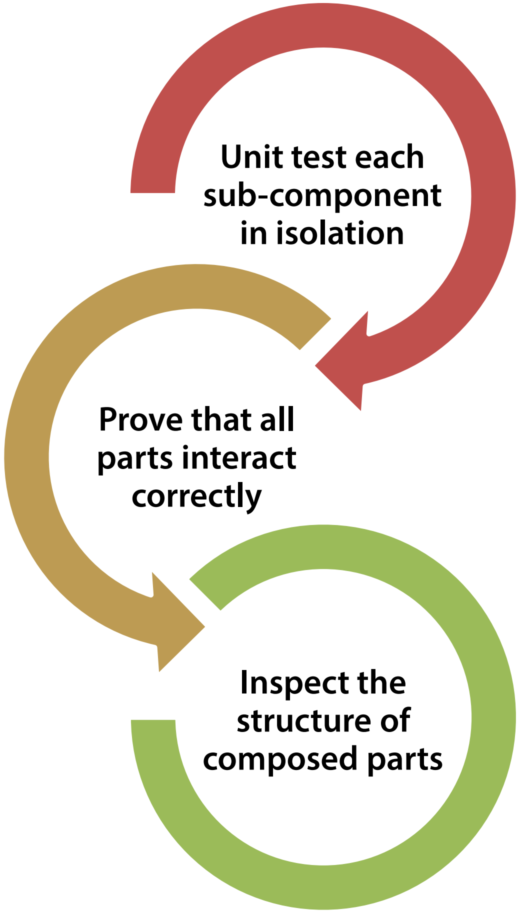
Prime factors

Bowling game

Word wrap

Favor object composition over class inheritance

- **Design Patterns, 1994**

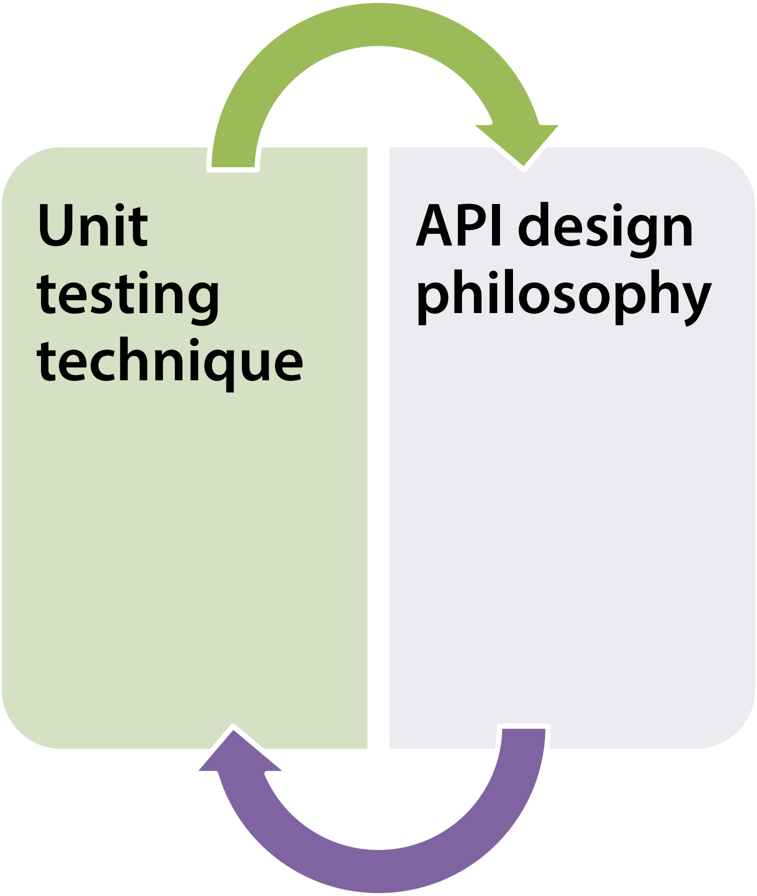


Unit test each sub-component in isolation

Prove that all parts interact correctly

Inspect the structure of composed parts

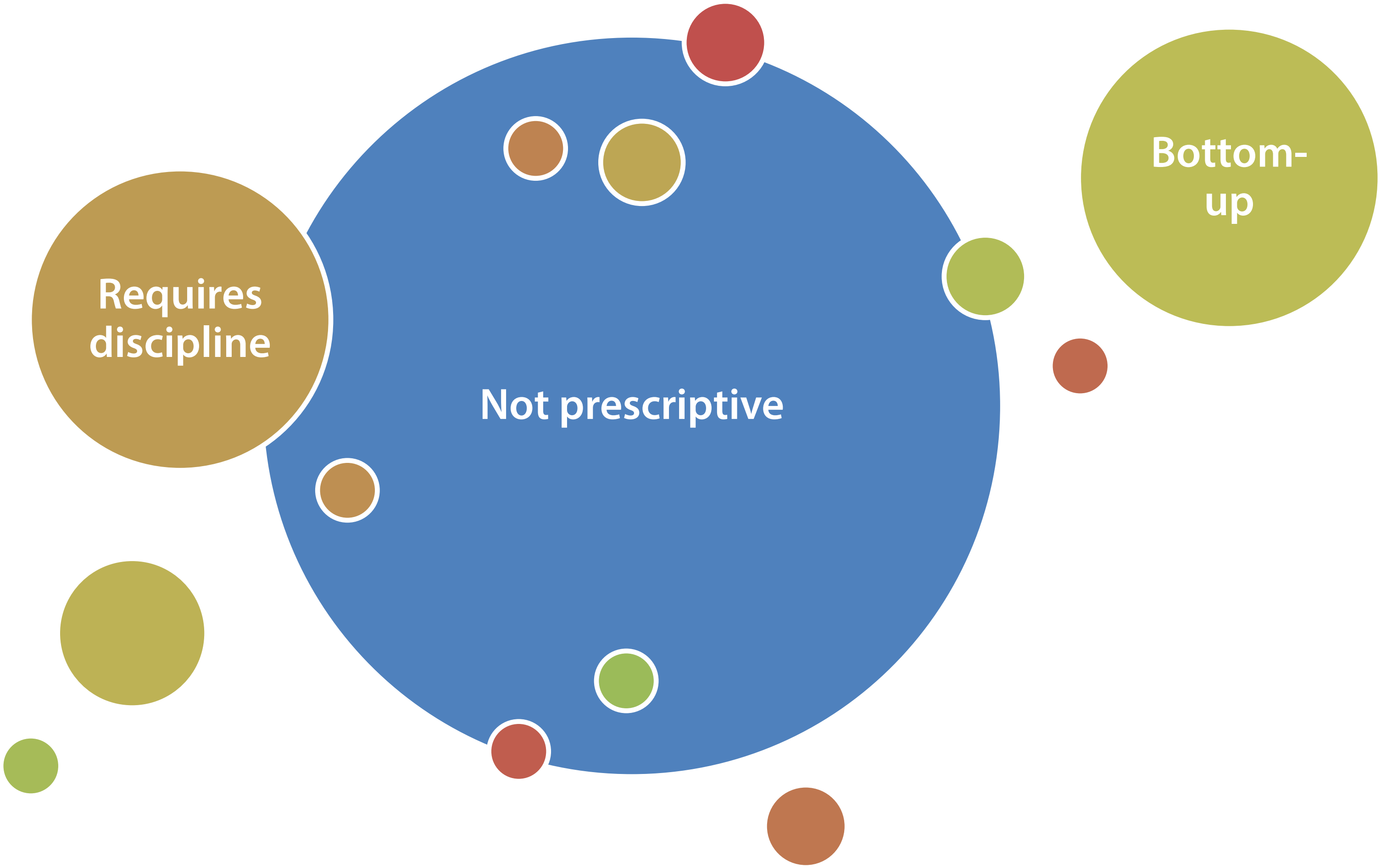
Structural Inspection



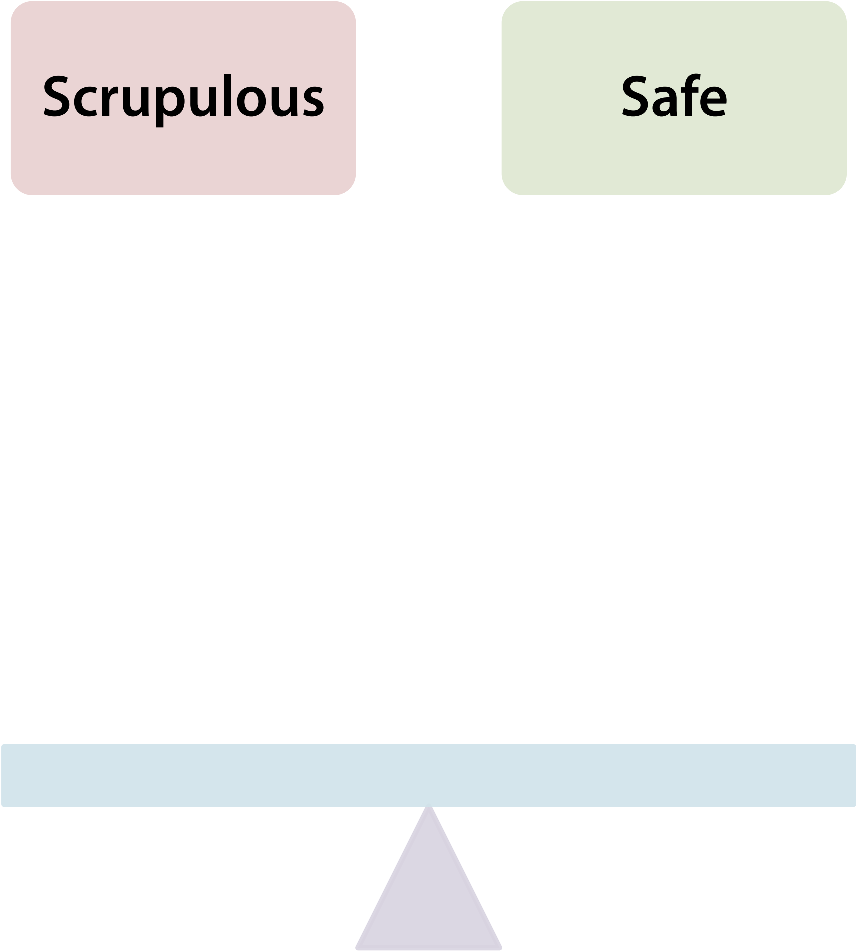
Unit testing technique

API design philosophy

Structural Inspection

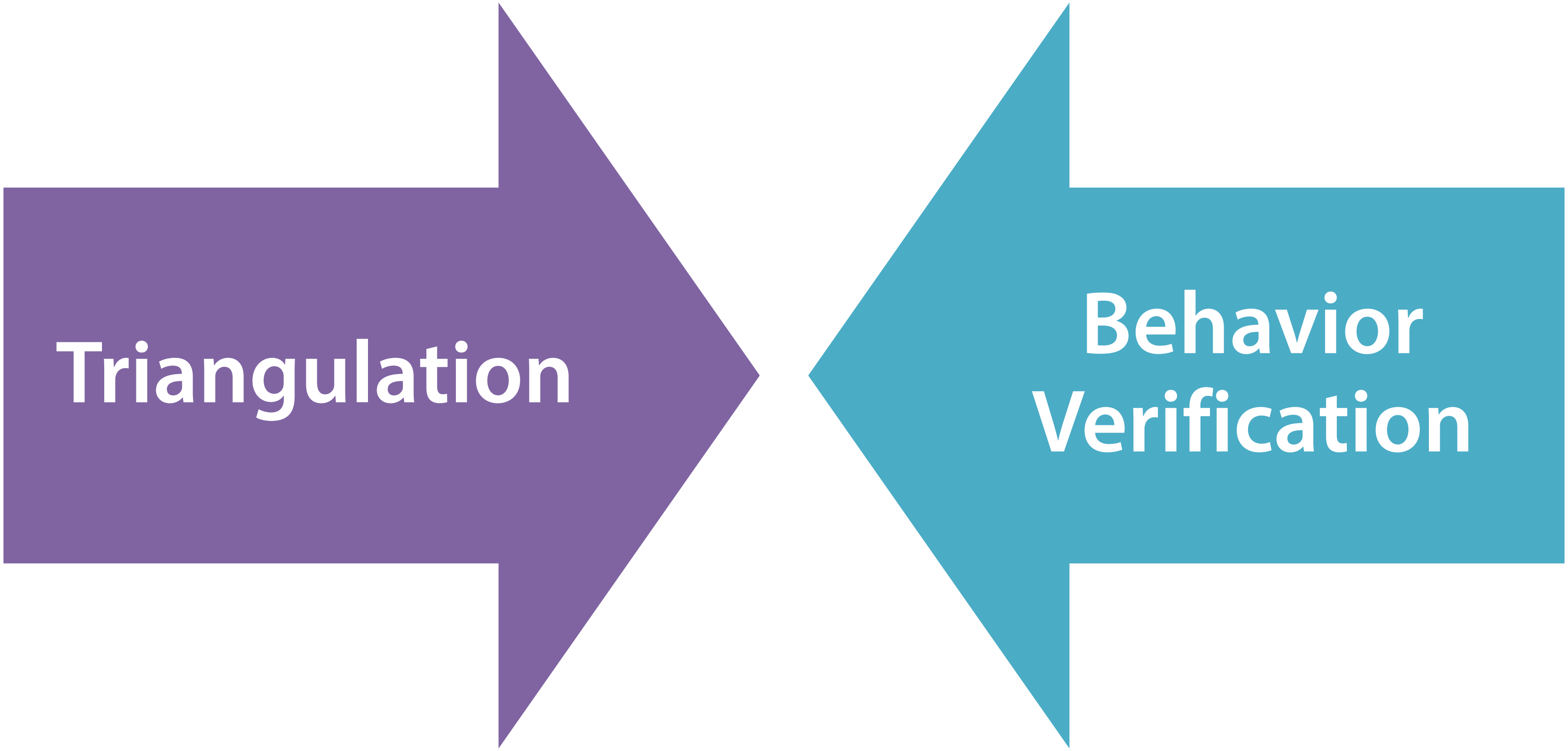


Tradeoff



Scrupulous

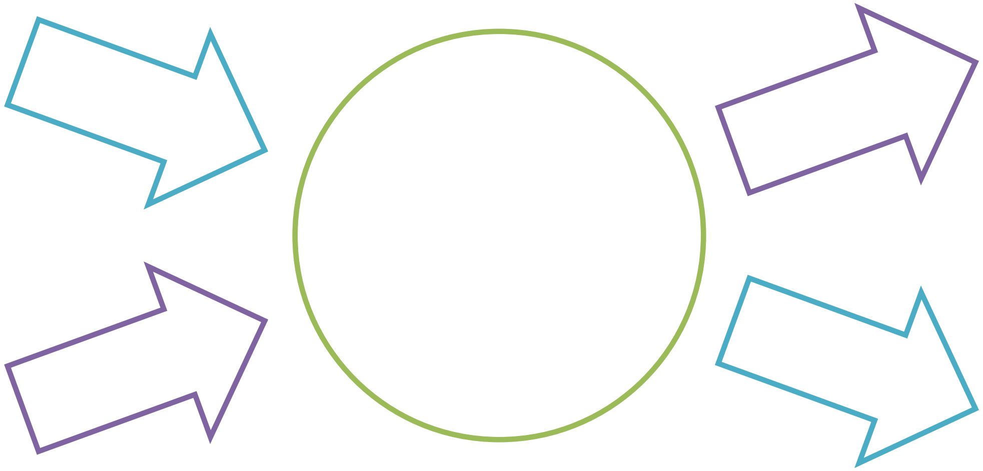
Safe



Triangulation

Behavior Verification

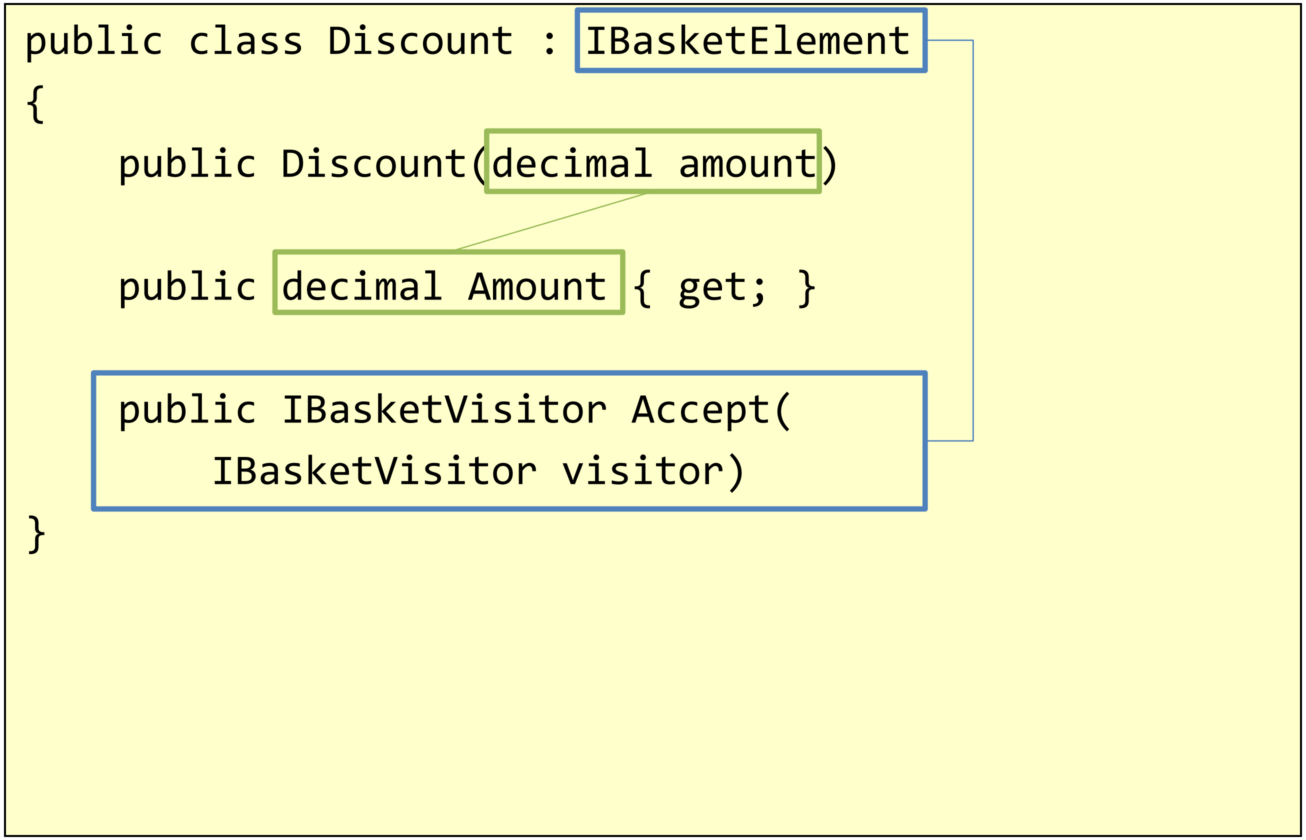
API design philosophy



What you
compose

you can also
expose

API design example



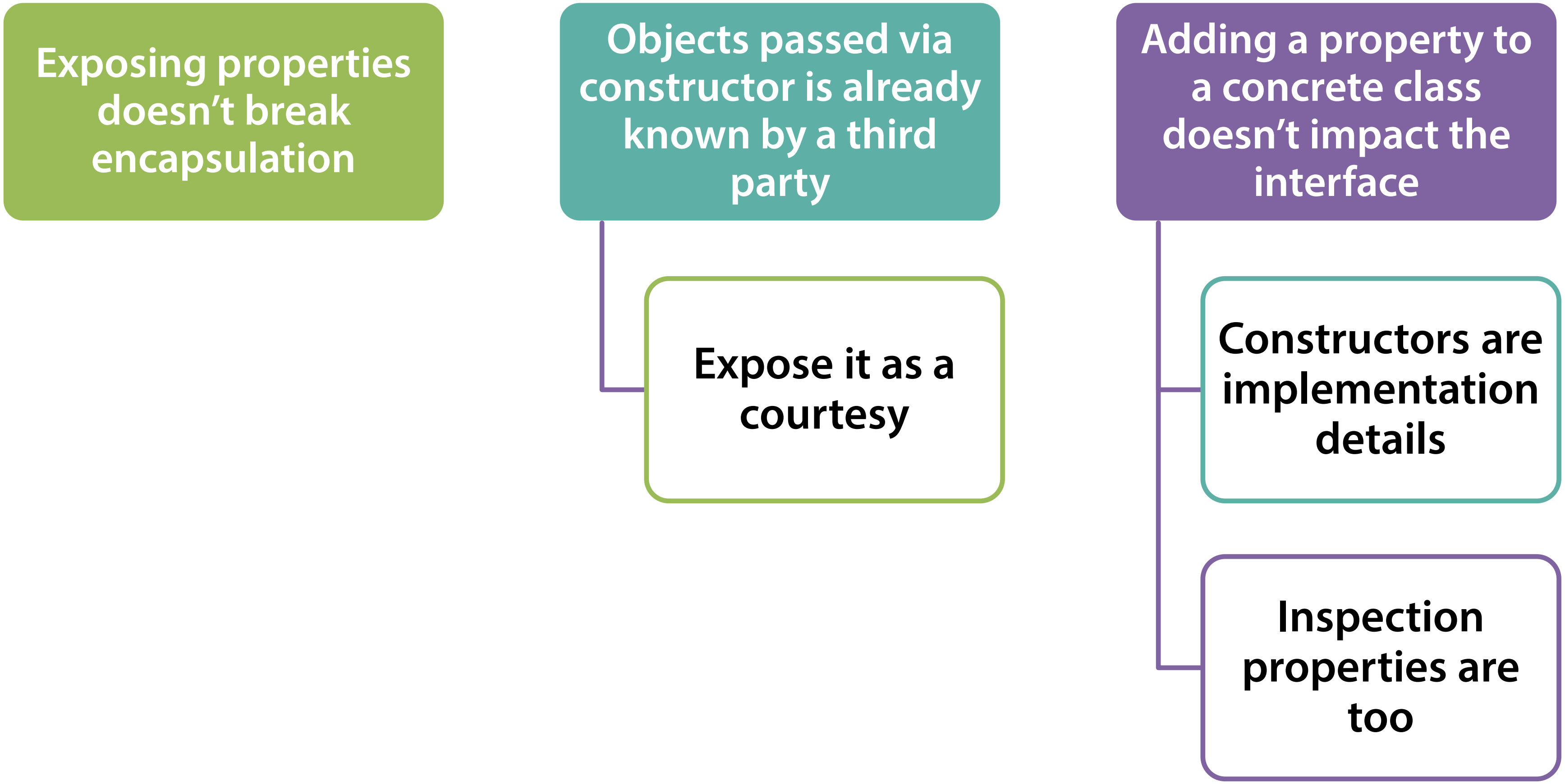
```
public class Discount : IBasketElement
{
    public Discount(decimal amount)

    public decimal Amount { get; }

    public IBasketVisitor Accept(
        IBasketVisitor visitor)
}
```

Typical reactions

I don’t want to add members only for testing purposes!

It breaks encapsulation!

Encapsulation



Exposing properties doesn’t break encapsulation

Objects passed via constructor is already known by a third party

Expose it as a courtesy

Adding a property to a concrete class doesn’t impact the interface

Constructors are implementation details

Inspection properties are too

Unit testing

Test each part in isolation

Prove that parts correctly interact

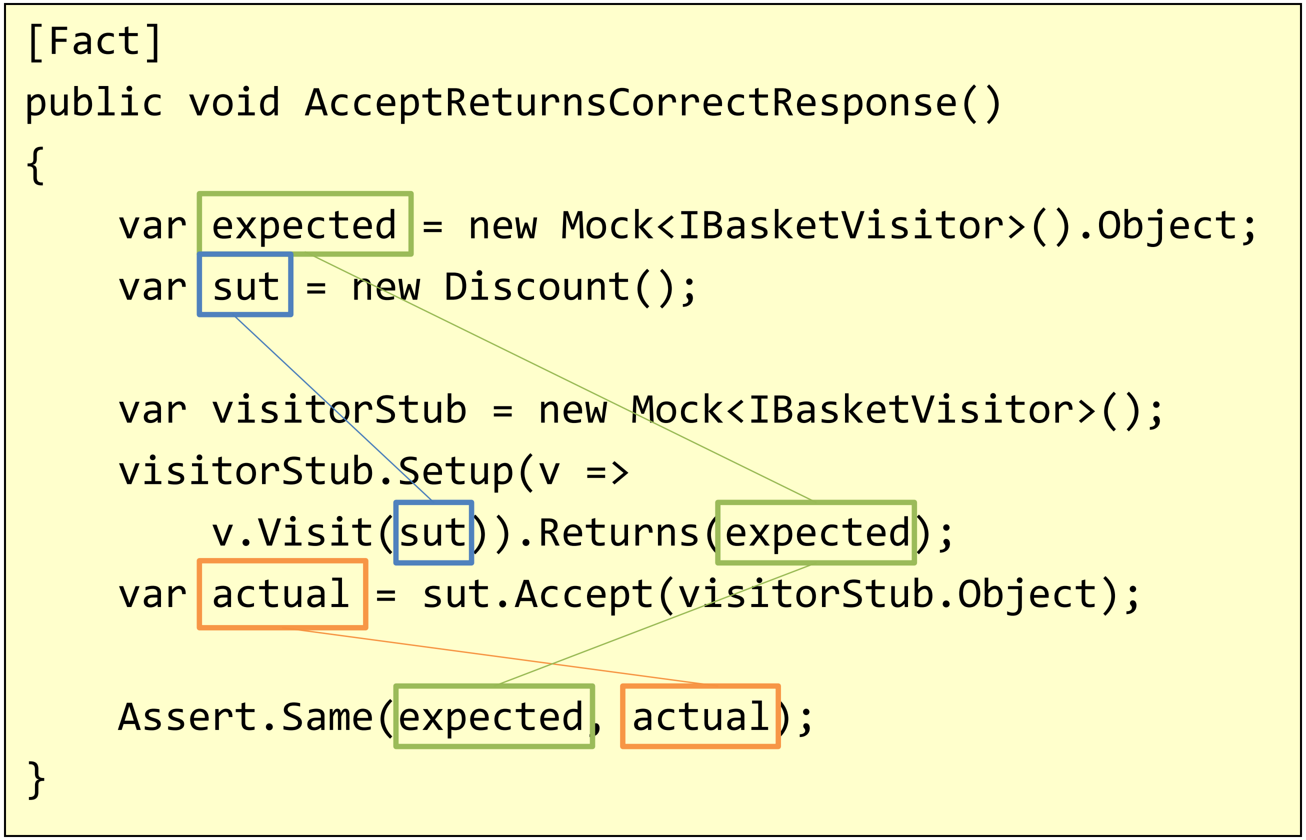
Prove that injected amount is exposed

```
[Theory]
[InlineData(1)]
[InlineData(2)]
public void AmountIsCorrect(int expected)
{
    var sut = new Discount(expected);
    var actual = sut.Amount;
    Assert.Equal(expected, actual);
}
```

Prove that Discount implements IBasketElement

```
[Fact]
public void SutIsBasketElement()
{
    var sut = new Discount();
    Assert.IsAssignableFrom<IBasketElement>(sut);
}
```

Verify that interface is correctly implemented

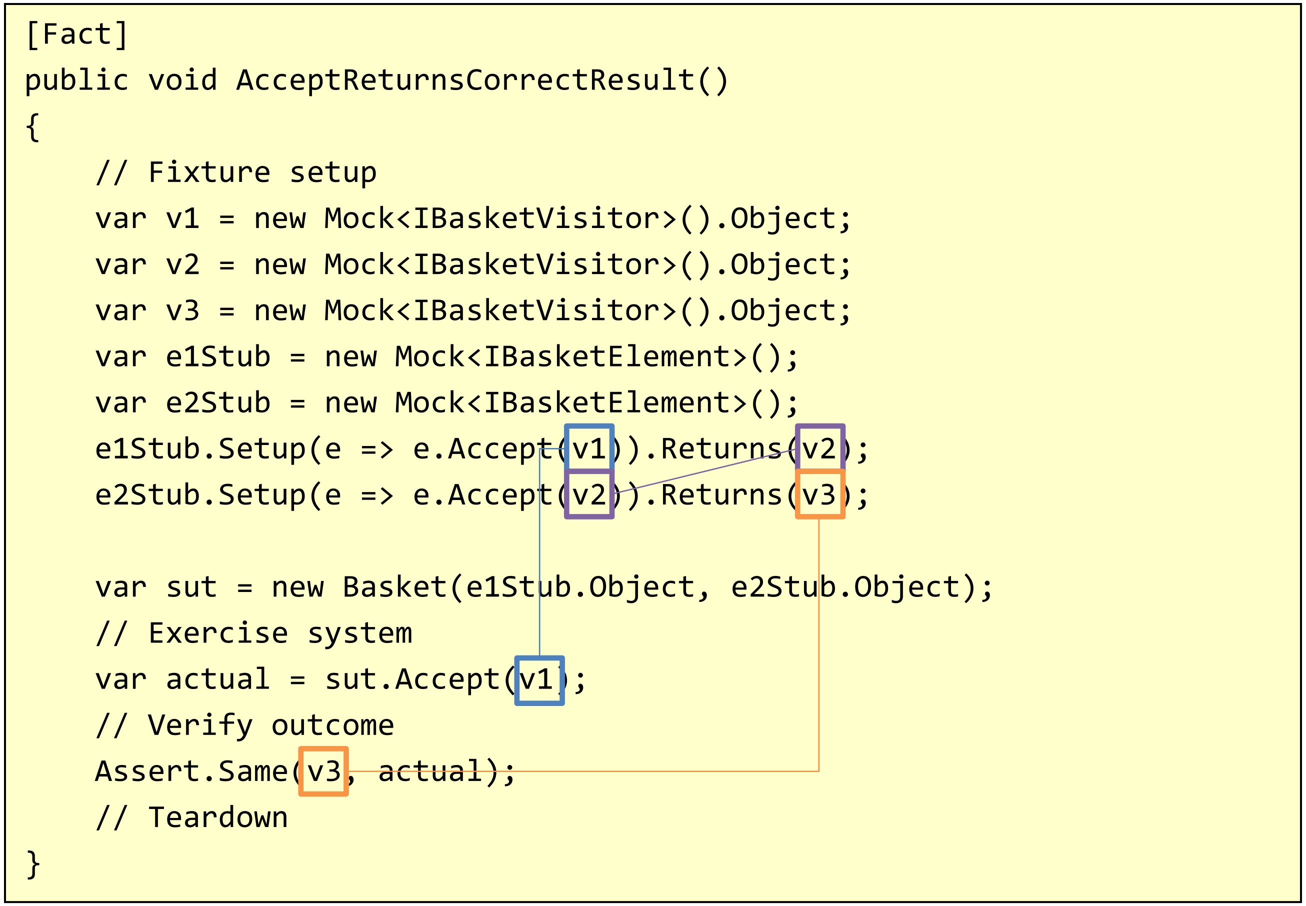


```
[Fact]
public void AcceptReturnsCorrectResponse()
{
    var expected = new Mock<IBasketVisitor>().Object;
    var sut = new Discount();

    var visitorStub = new Mock<IBasketVisitor>();
    visitorStub.Setup(v =>
        v.Visit(sut)).Returns(expected);
    var actual = sut.Accept(visitorStub.Object);

    Assert.Same(expected, actual);
}
```

Discount implementation

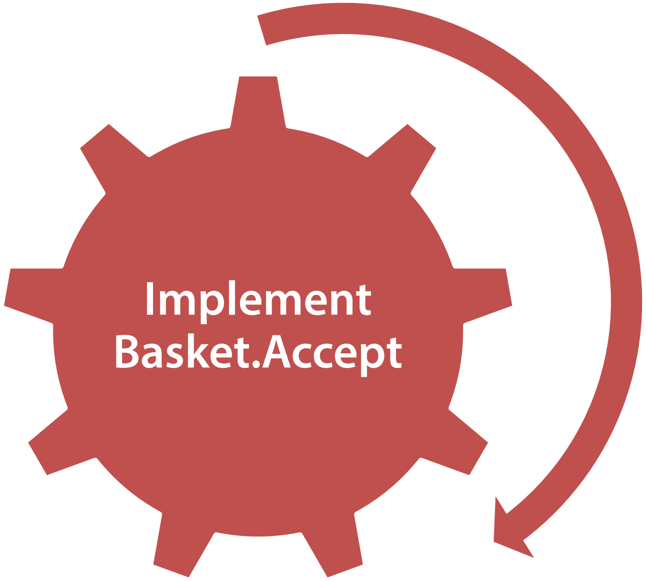
```
public class Discount : IBasketElement
{
    private readonly decimal amount;

    public Discount(decimal amount)
    {
        this.amount = amount;
    }

    public IBasketVisitor Accept(IBasketVisitor visitor)
    {
        return visitor.Visit(this);
    }

    public decimal Amount
    {
        get { return this.amount; }
    }
}
```

Basket Behavior Verification



```
[Fact]
public void AcceptReturnsCorrectResult()
{
    // Fixture setup
    var v1 = new Mock<IBasketVisitor>().Object;
    var v2 = new Mock<IBasketVisitor>().Object;
    var v3 = new Mock<IBasketVisitor>().Object;
    var e1Stub = new Mock<IBasketElement>();
    var e2Stub = new Mock<IBasketElement>();
    e1Stub.Setup(e => e.Accept(v1)).Returns(v2);
    e2Stub.Setup(e => e.Accept(v2)).Returns(v3);

    var sut = new Basket(e1Stub.Object, e2Stub.Object);
    // Exercise system
    var actual = sut.Accept(v1);
    // Verify outcome
    Assert.Same(v3, actual);
    // Teardown
}
```

Demo



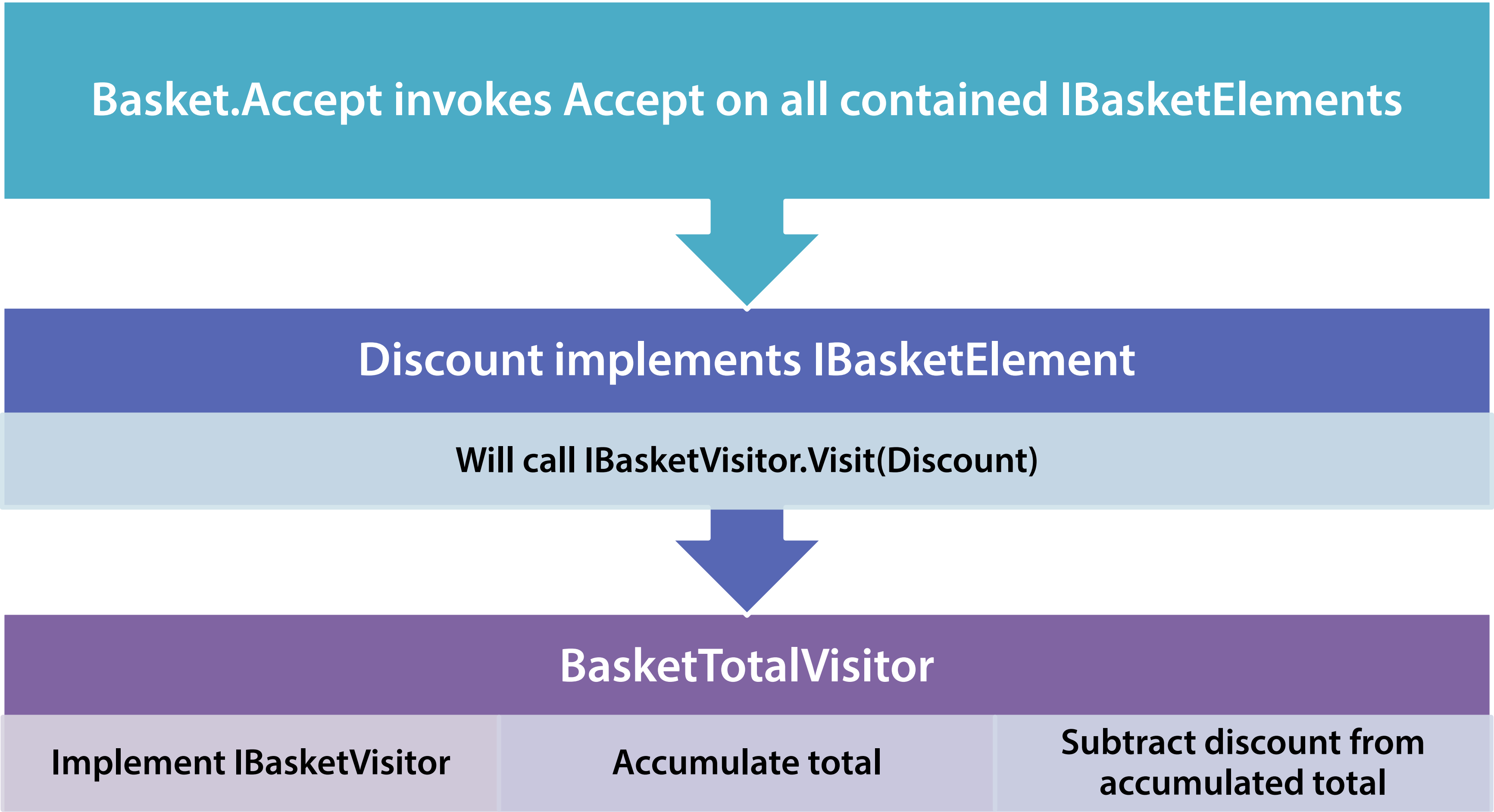
Demo recap

Implemented Basket.Accept

Imperative

Functional

Combining knowledge



Basket.Accept invokes Accept on all contained IBasketElements

Discount implements IBasketElement

Will call IBasketVisitor.Visit(Discount)

BasketTotalVisitor

Implement IBasketVisitor

Accumulate total

Subtract discount from accumulated total

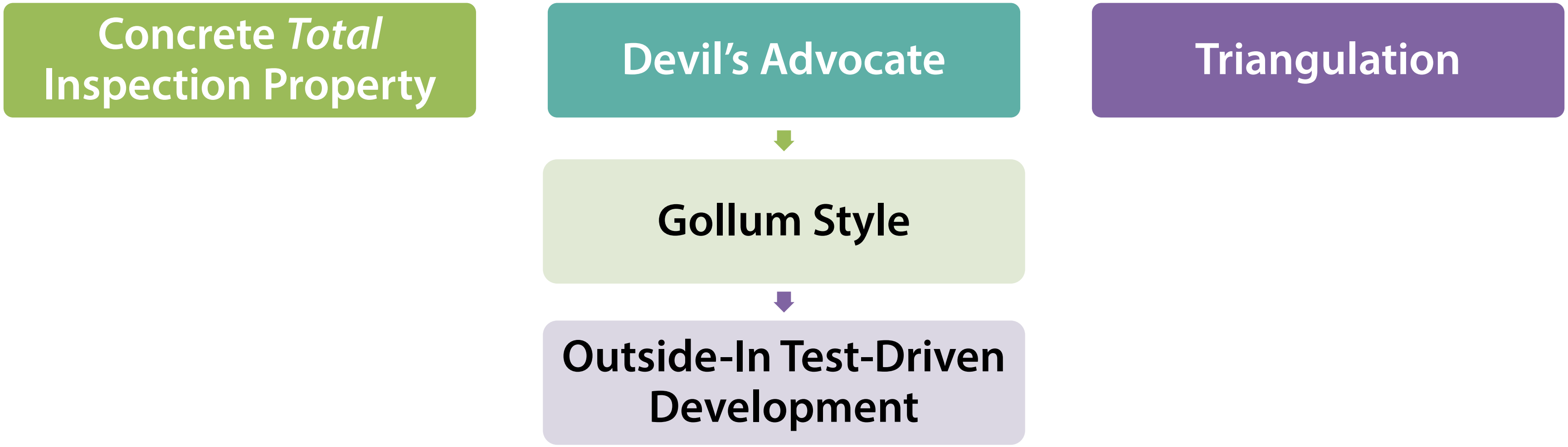
Dealing with discount while calculating the total

```
[Theory]
[InlineData(1, 1)]
[InlineData(2, 1)]
[InlineData(3, 2)]
public void VisitDiscountReturnsCorrectResult(
    int initialTotal,
    int discount)
{
    var sut = new BasketTotalVisitor(initialTotal);

    var actual = sut.Visit(new Discount(discount));

    var btv = Assert.IsAssignableFrom<BasketTotalVisitor>(actual);
    Assert.Equal(initialTotal - discount, btv.Total);
}
```

Demo

Test-Drive the BasketTotalVisitor behavior

Demo recap



Concrete *Total* Inspection Property

Devil's Advocate

Gollum Style

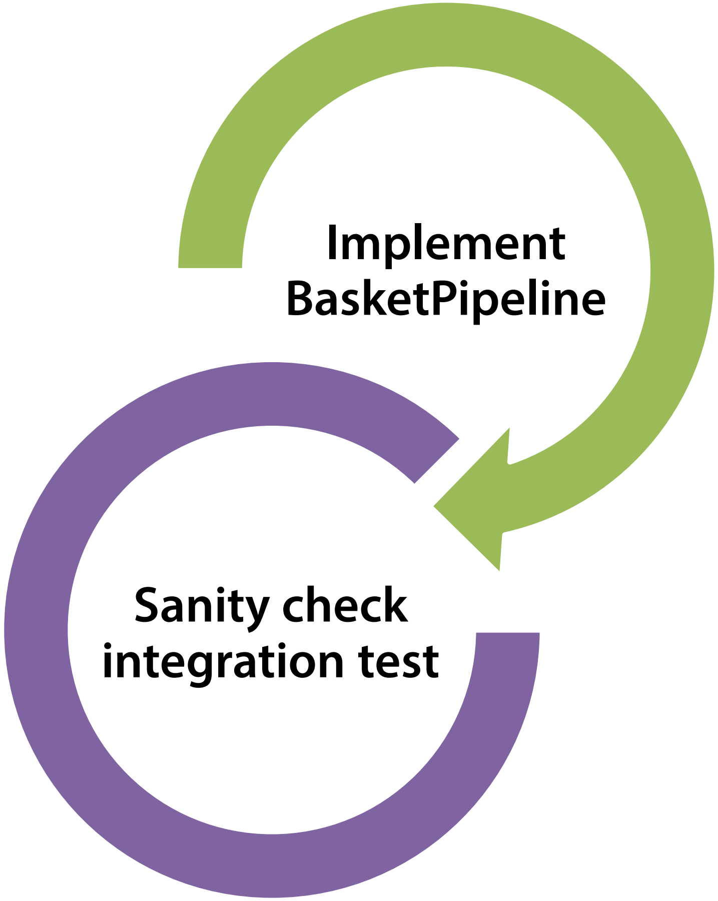
Outside-In Test-Driven Development

Triangulation

Verifying a Facade

```
[Fact]
public void SutCorrectlyConvertsToPipe()
{
    CompositePipe<Basket> sut = new BasketPipeline();

    var visitors = sut
        .OfType<BasketVisitorPipe>()
        .Select(bvp => bvp.Visitor);

    var dv = Assert.IsAssignableFrom<VolumeDiscountVisitor>(visitors.First());
    Assert.Equal(500, dv.Threshold);
    Assert.Equal(.05m, dv.Rate);

    var vv = Assert.IsAssignableFrom<VatVisitor>(visitors.ElementAt(1));
    Assert.Equal(.25m, vv.Rate);

    var btv = Assert.IsAssignableFrom<BasketTotalVisitor>(visitors.Last());
}
```

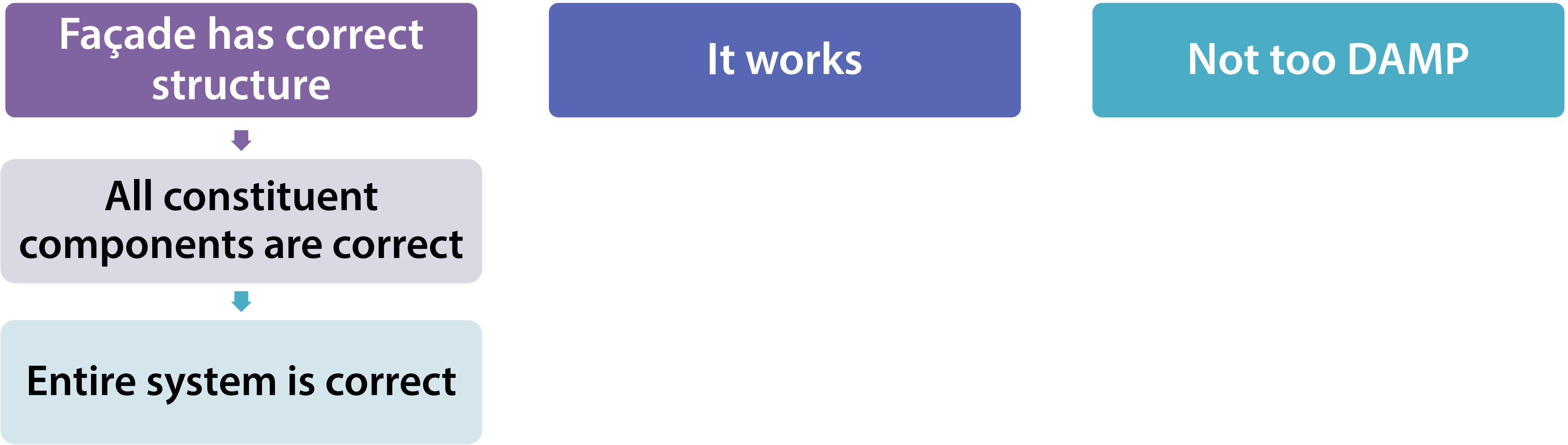
Demo



Implement BasketPipeline

Sanity check integration test

Demo recap



Façade has correct structure

All constituent components are correct

Entire system is correct

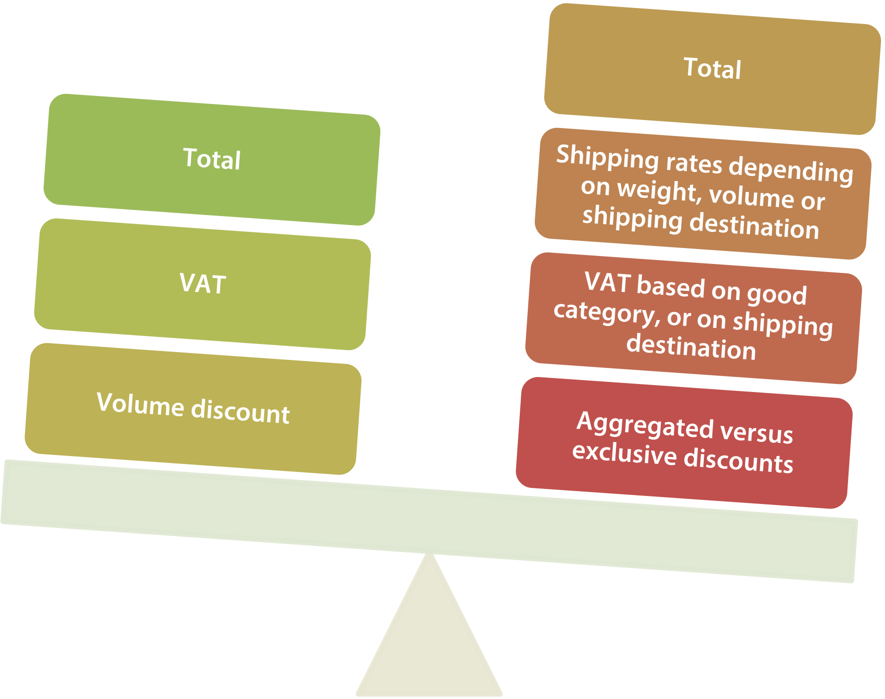
It works

Not too DAMP

Too enterprisey?

Simple basket rules

Complex basket rules



Total

VAT

Volume discount

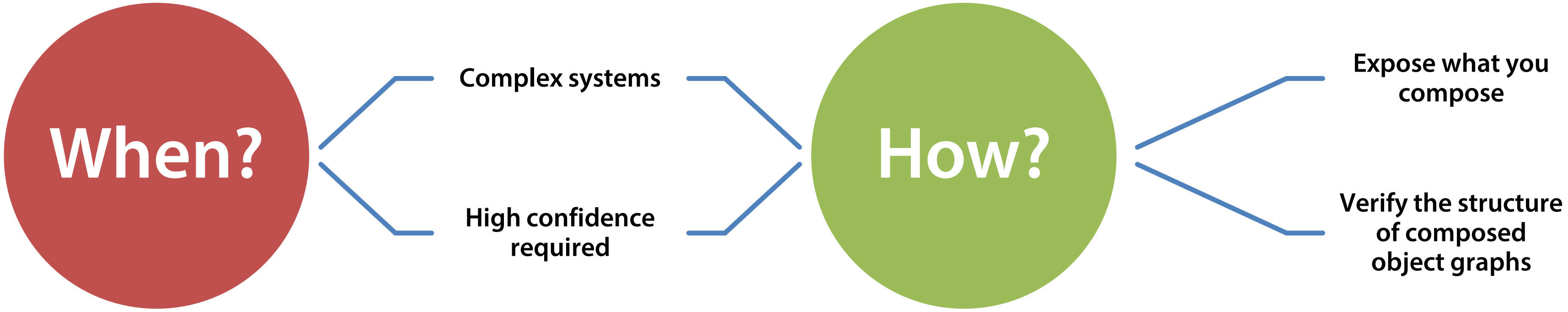
Total

Shipping rates depending on weight, volume or shipping destination

VAT based on good category, or on shipping destination

Aggregated versus exclusive discounts

Summary


When?
Complex systems
High confidence required
How?
Expose what you compose
Verify the structure of composed object graphs