Google™

Tutorial:

How to Write Hard to Test Code

-- Miško Hevery & Cory Smith

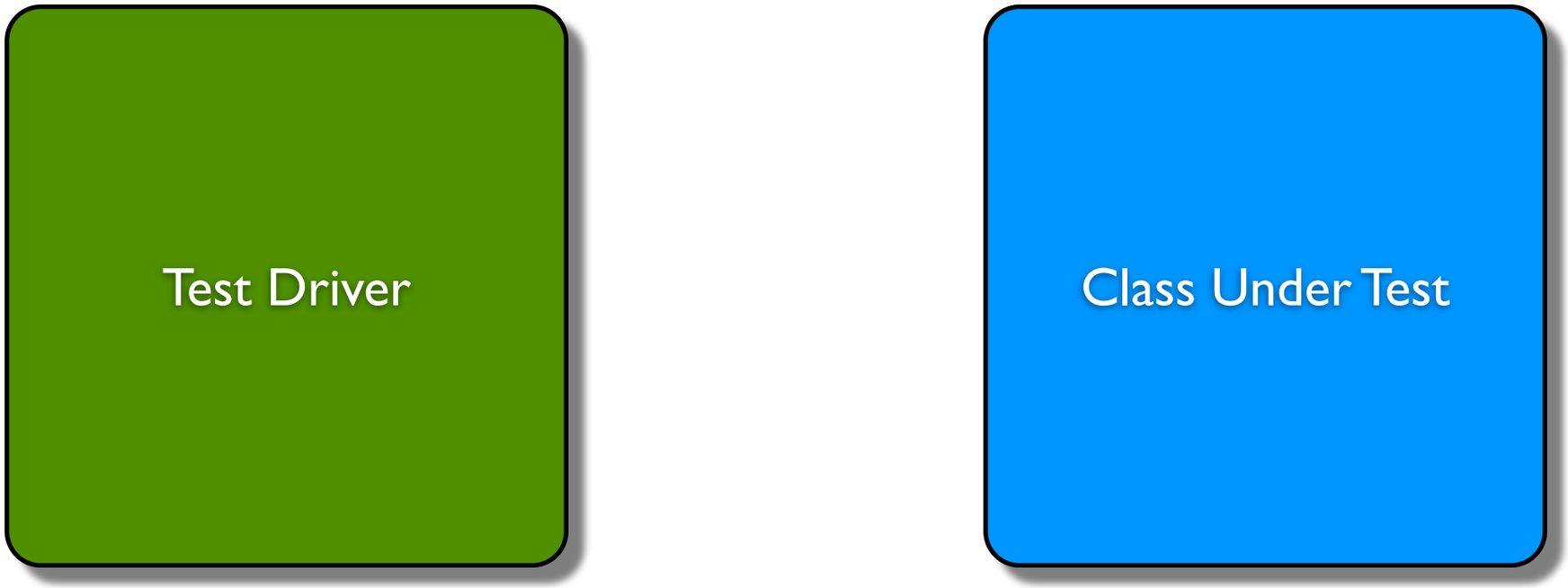
OOPSLA'09, October 25–29, 2009, Orlando, FL, USA.
ACM 09/10.

Work in Constructor

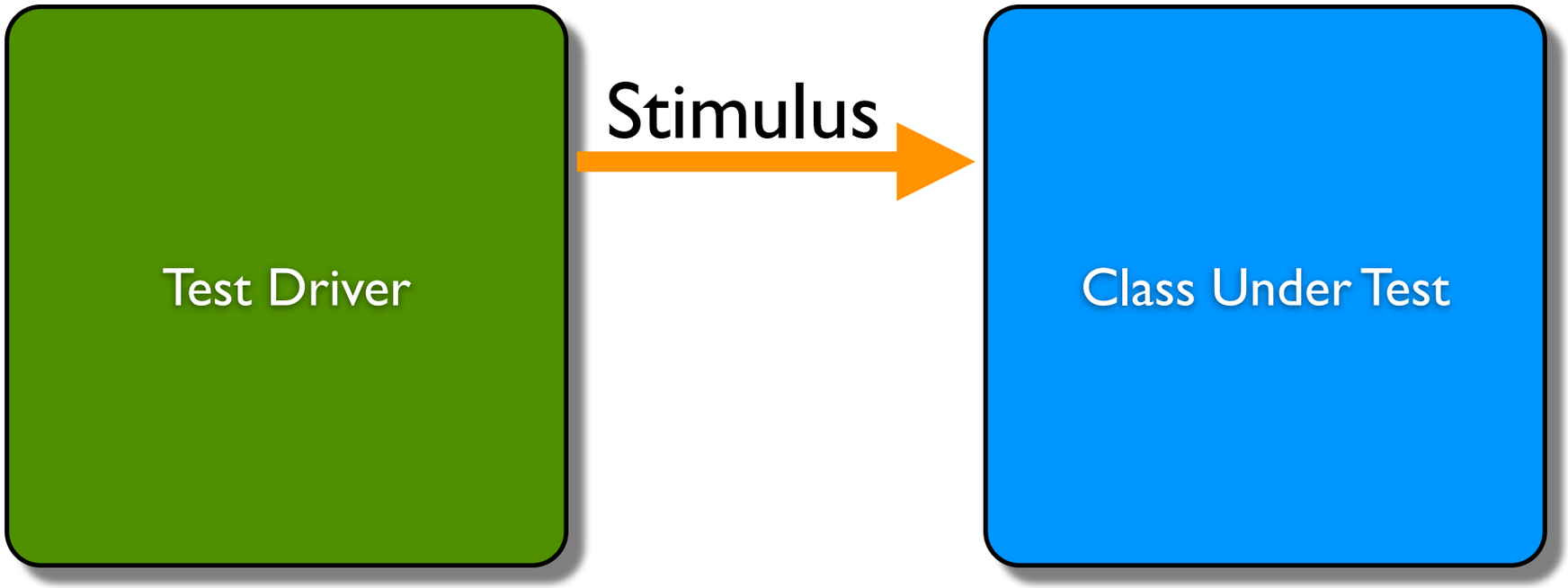
Unit Testing a Class

Unit Testing a Class

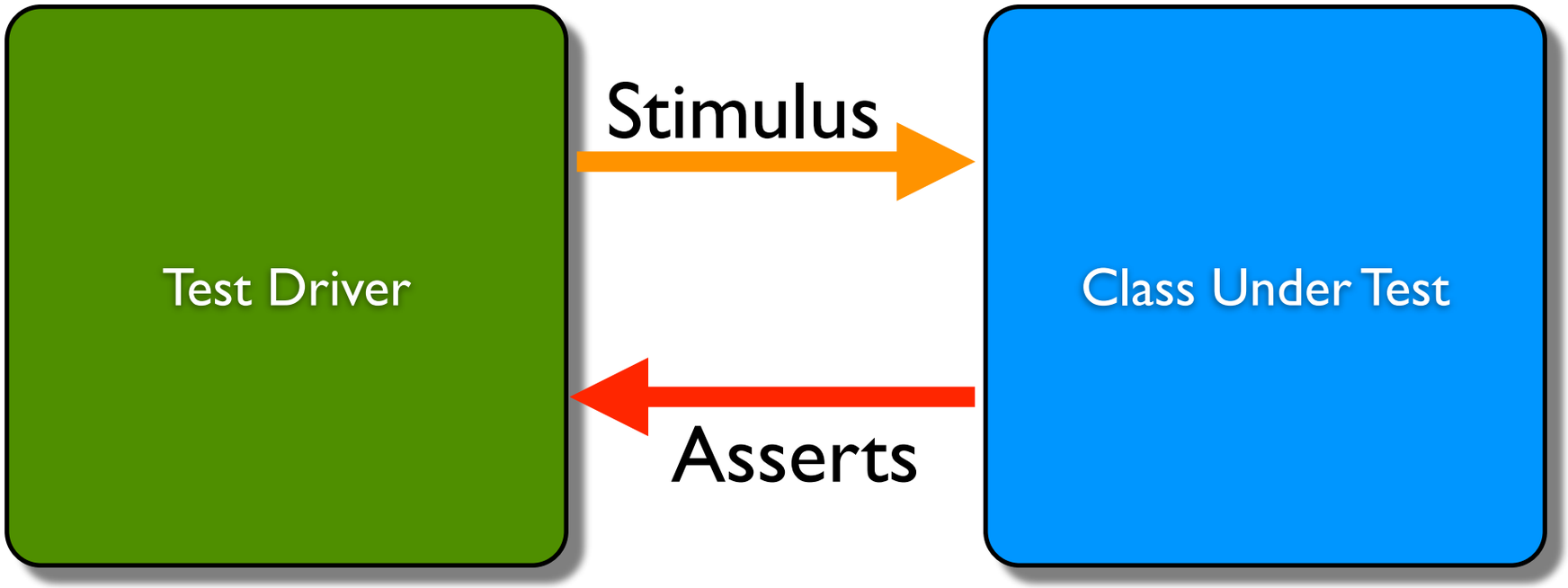
Unit Testing a Class



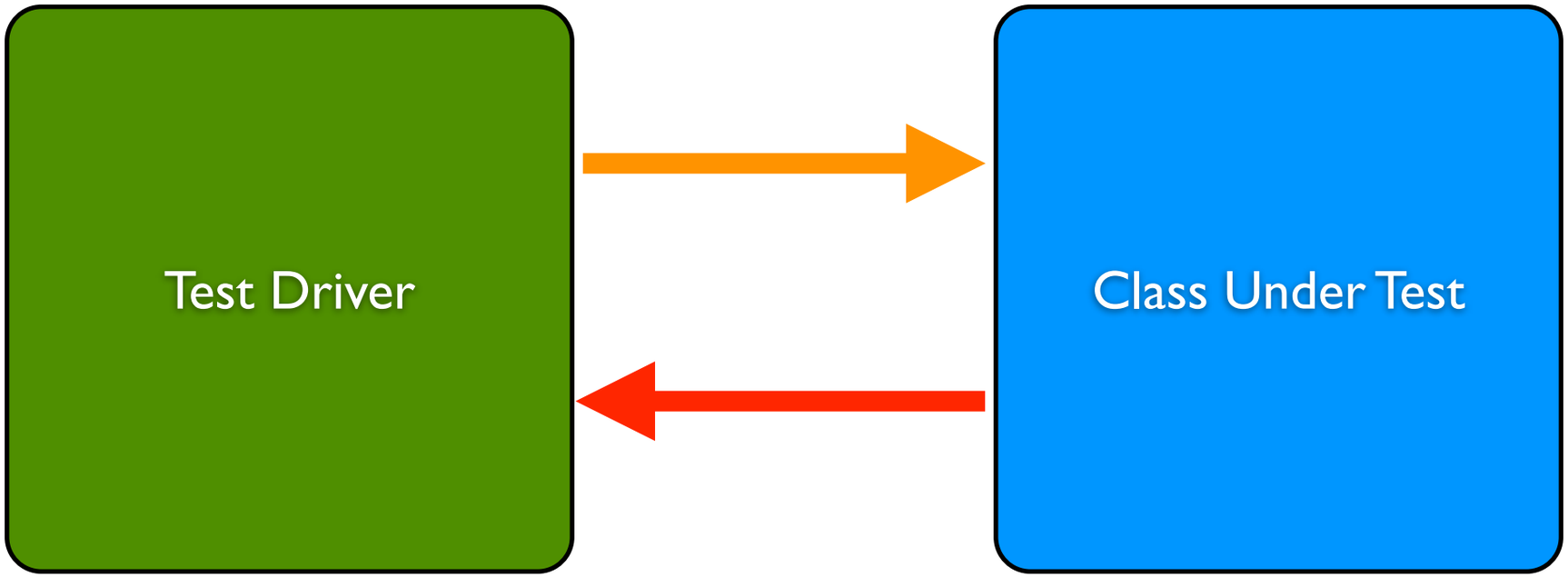
Unit Testing a Class



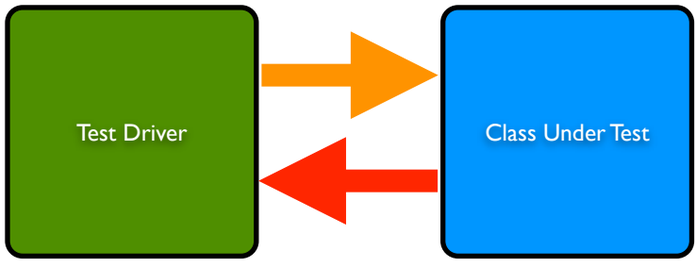
Unit Testing a Class



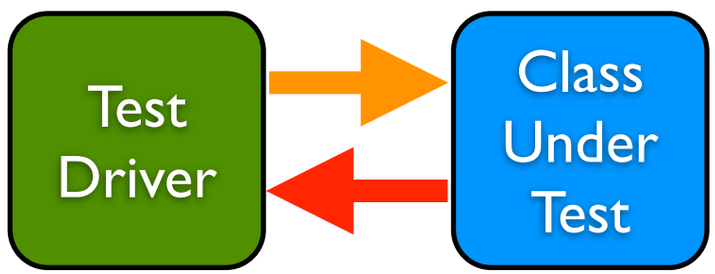
Unit Testing a Class



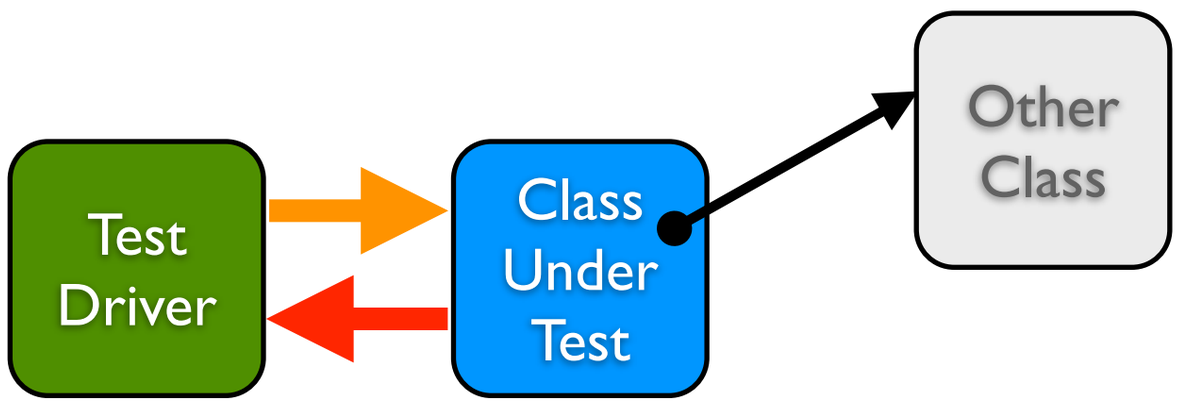
Unit Testing a Class



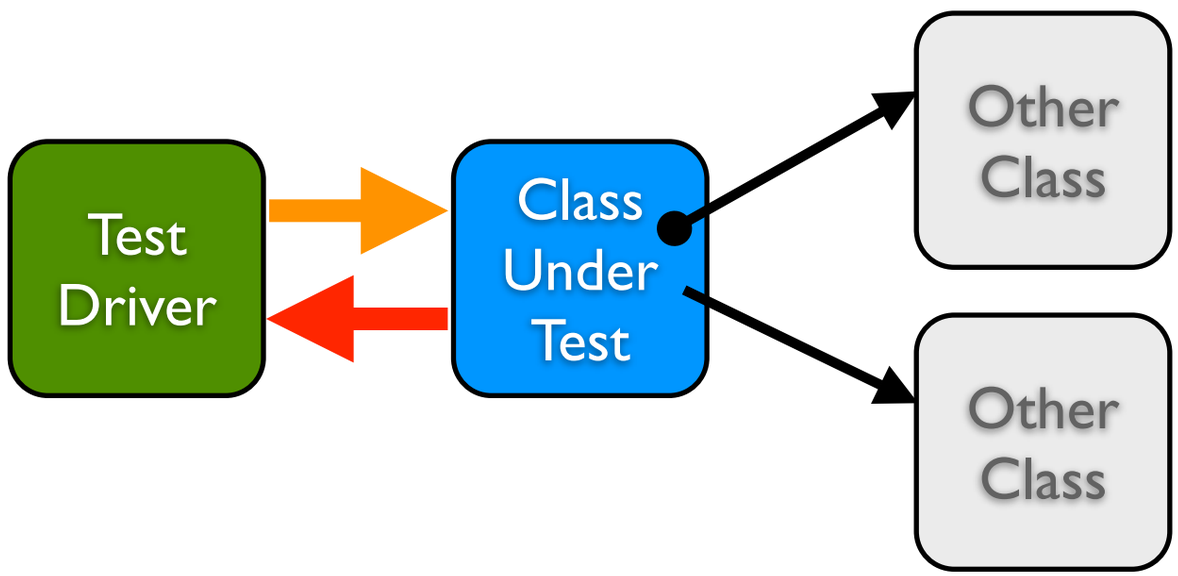
Unit Testing a Class



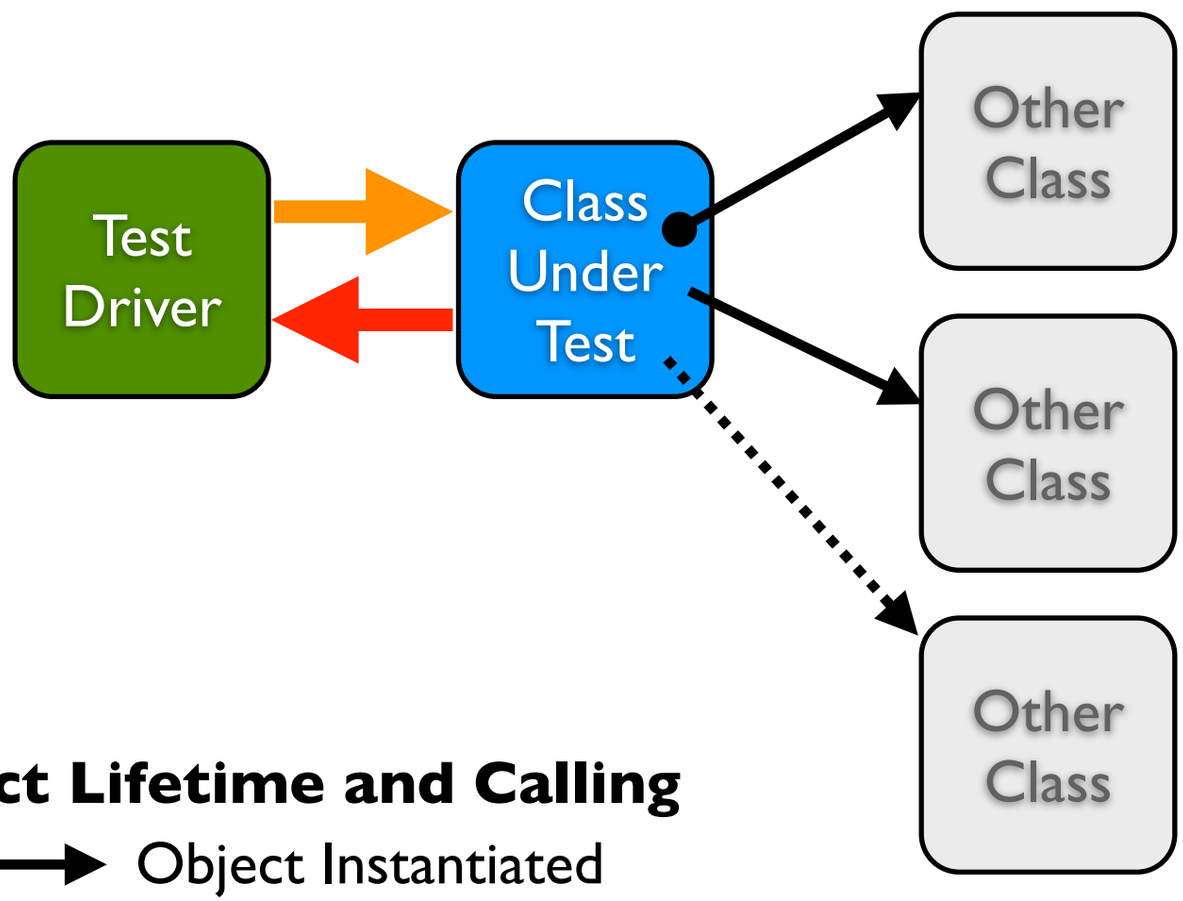
Unit Testing a Class



Unit Testing a Class



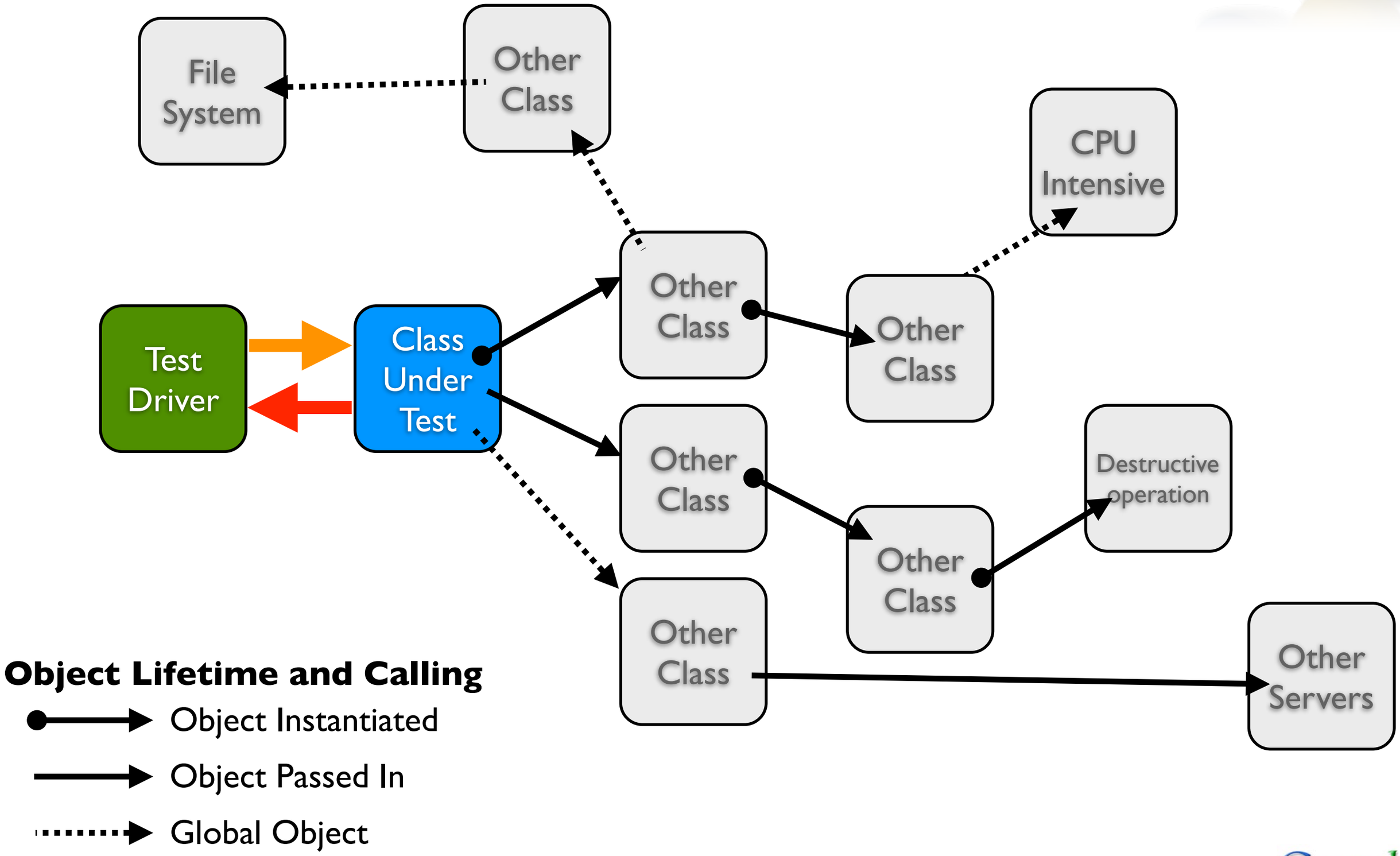
Unit Testing a Class



Object Lifetime and Calling

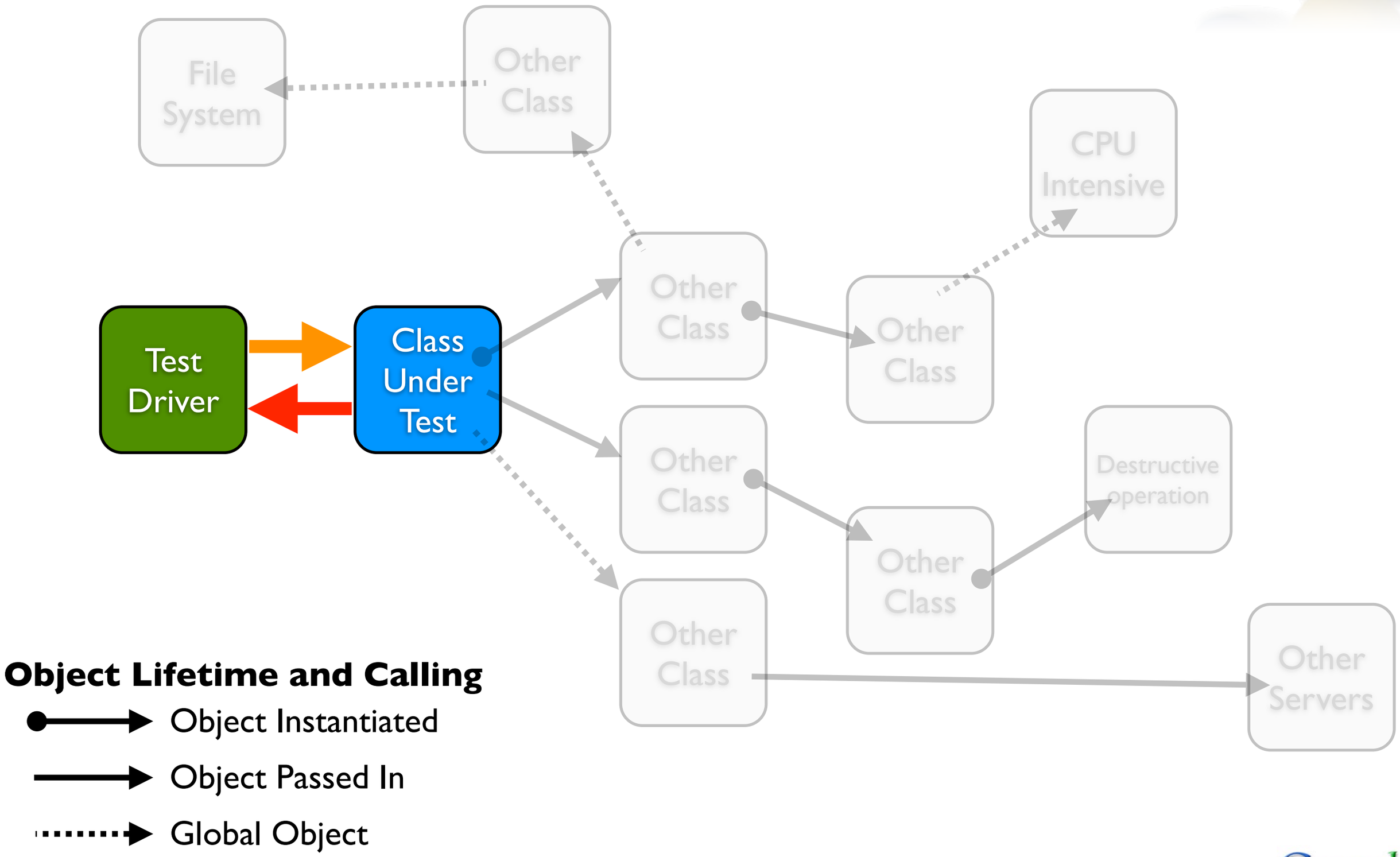
- Object Instantiated
- Object Passed In
- Global Object

Unit Testing a Class



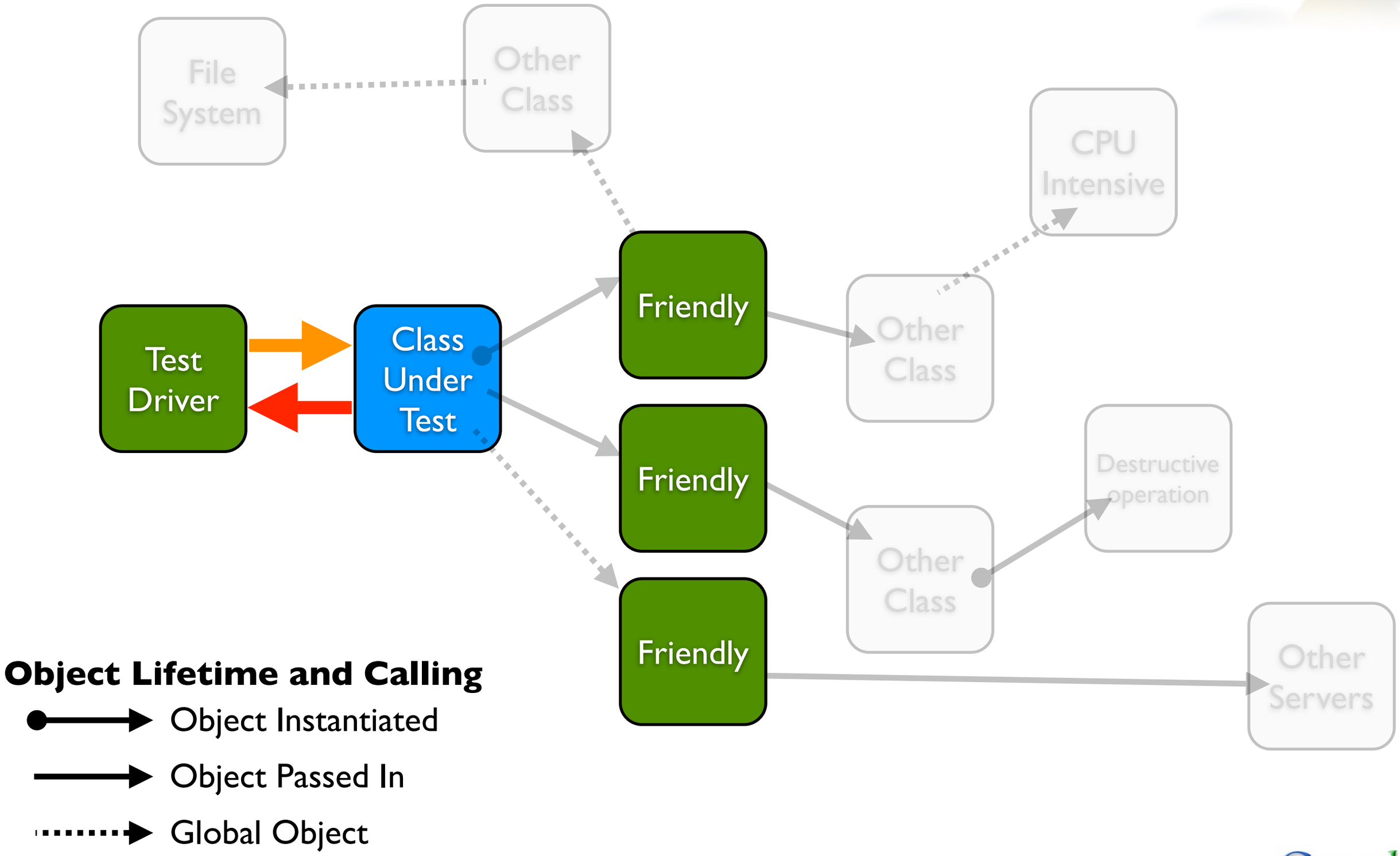
Google

Unit Testing a Class



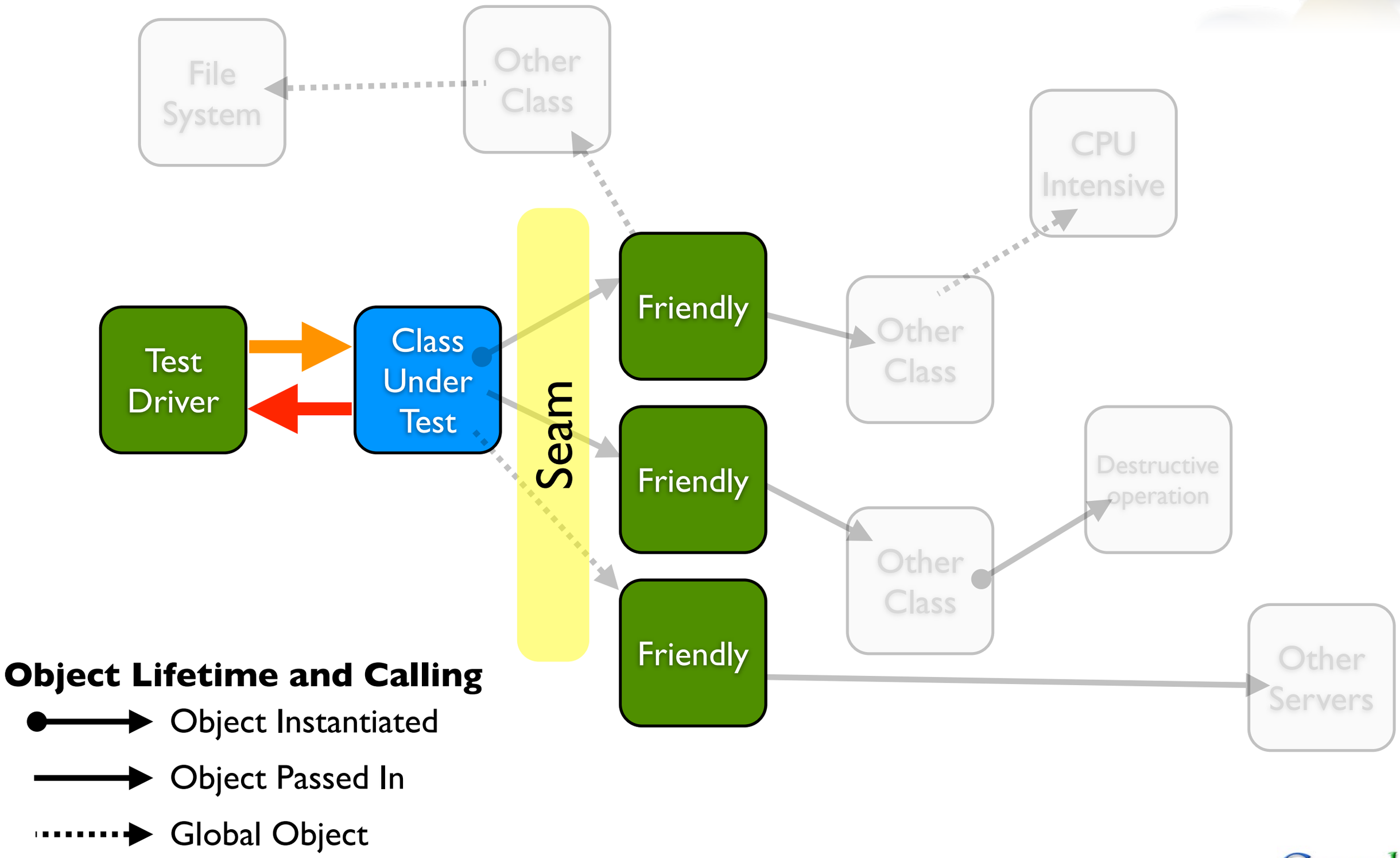
Google

Unit Testing a Class



Google

Unit Testing a Class



Google

Object Graph
Construction
& Lookup

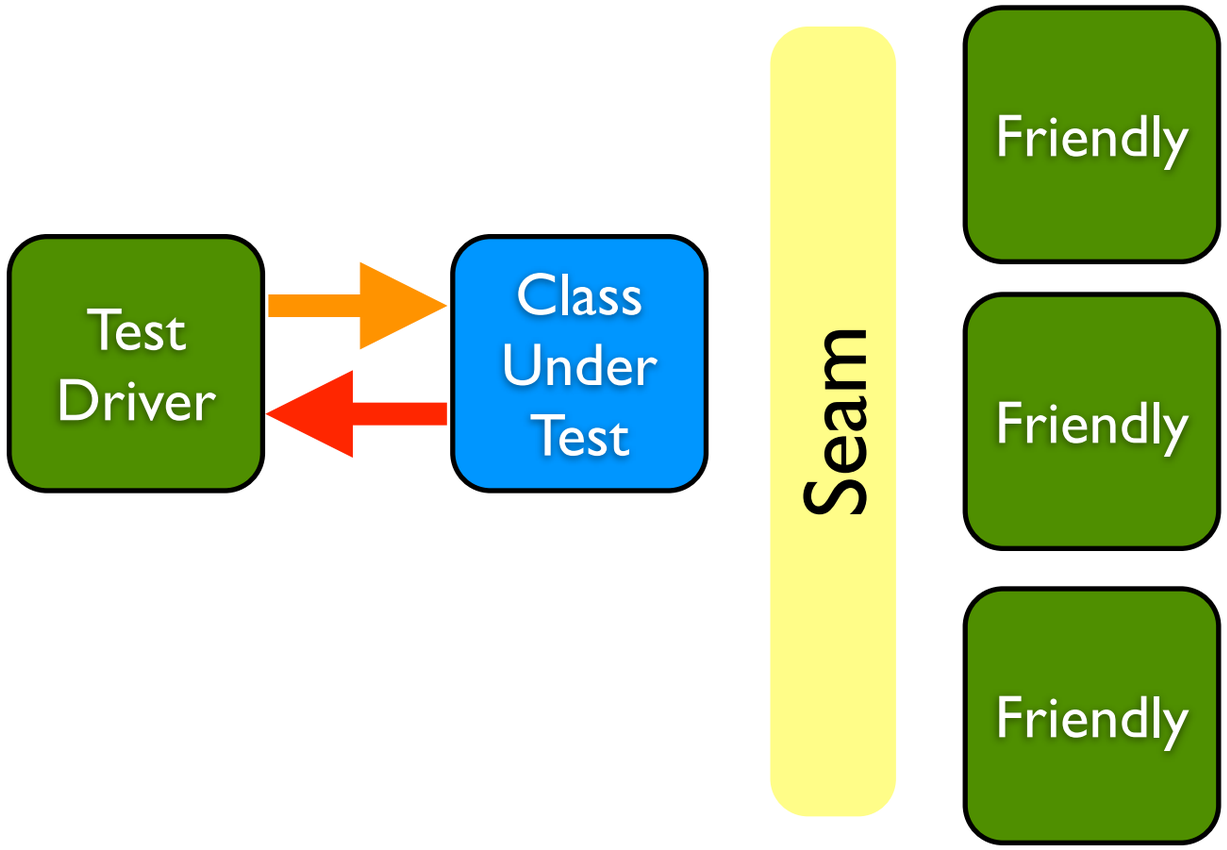
Business
Logic

Object Graph
Construction
& Lookup

Business
Logic

Google

Unit Testing a Class

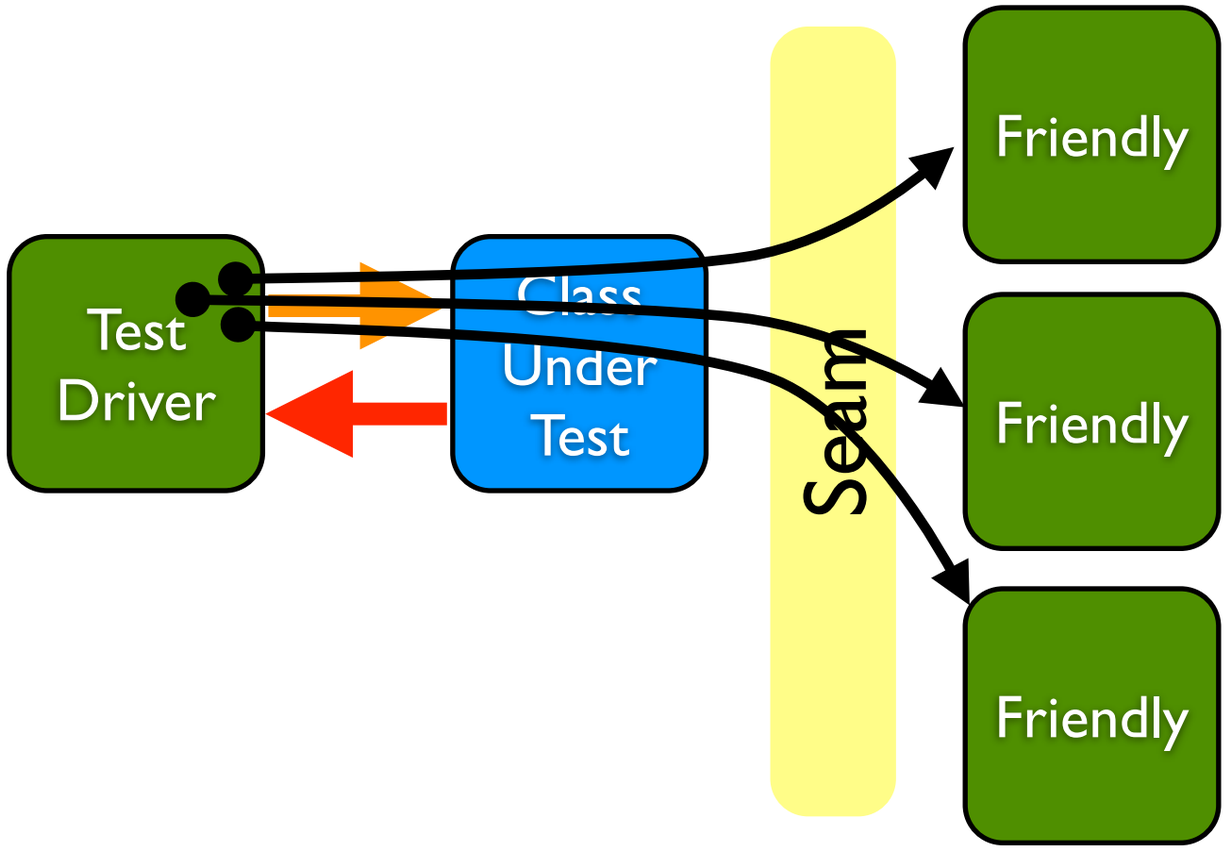


Object Lifetime and Calling

- Object Instantiated
- Object Passed In
- Global Object

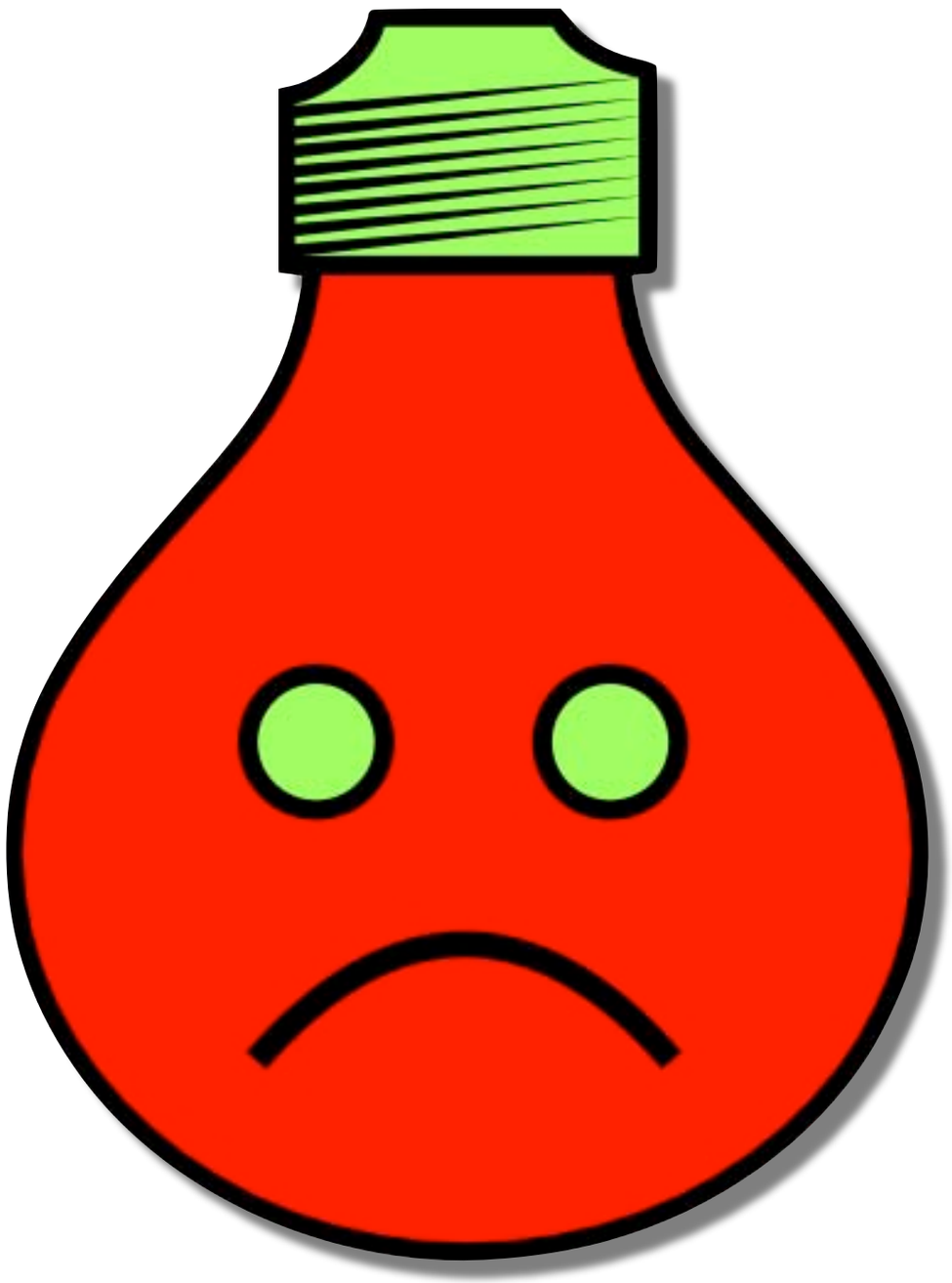
Google

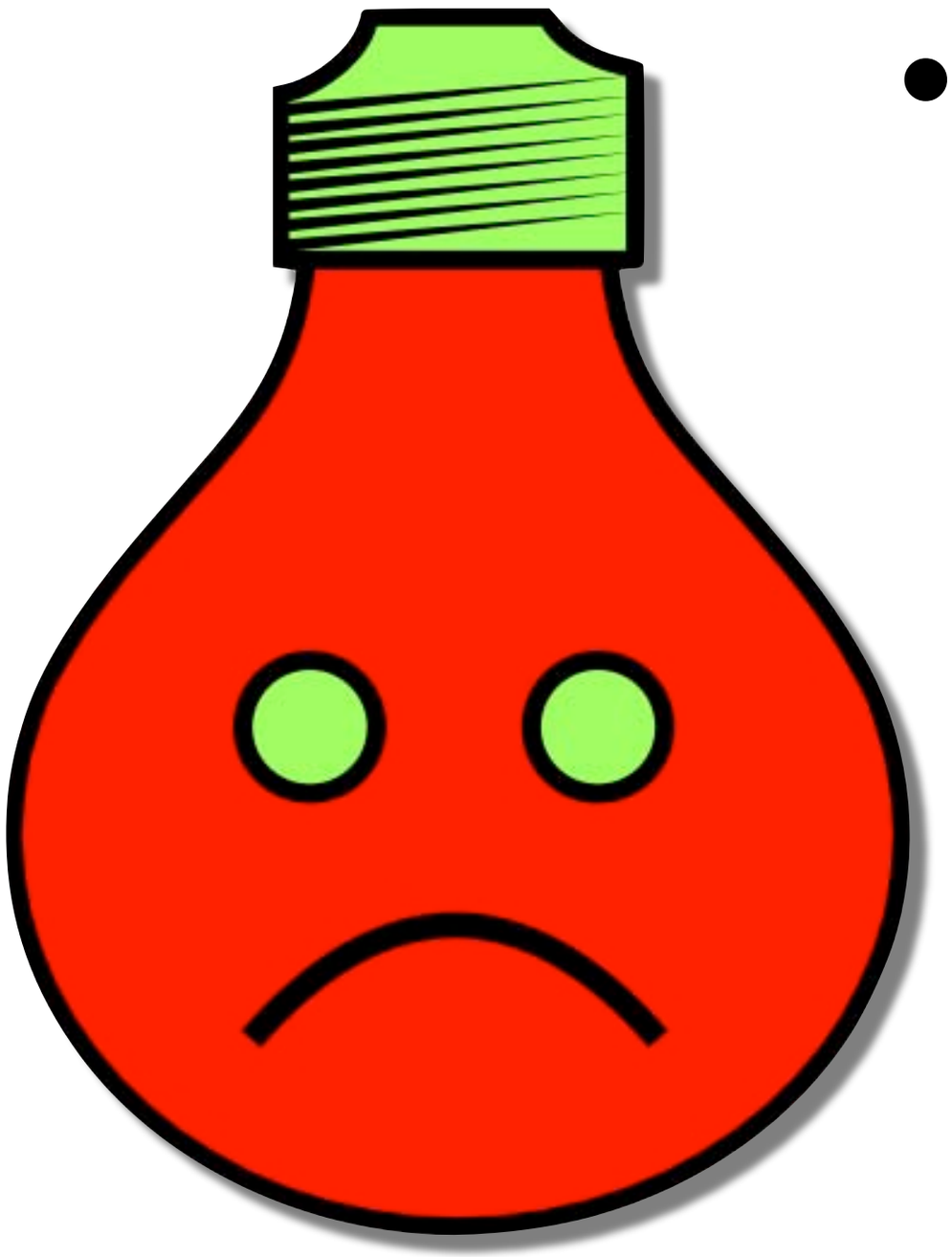
Unit Testing a Class



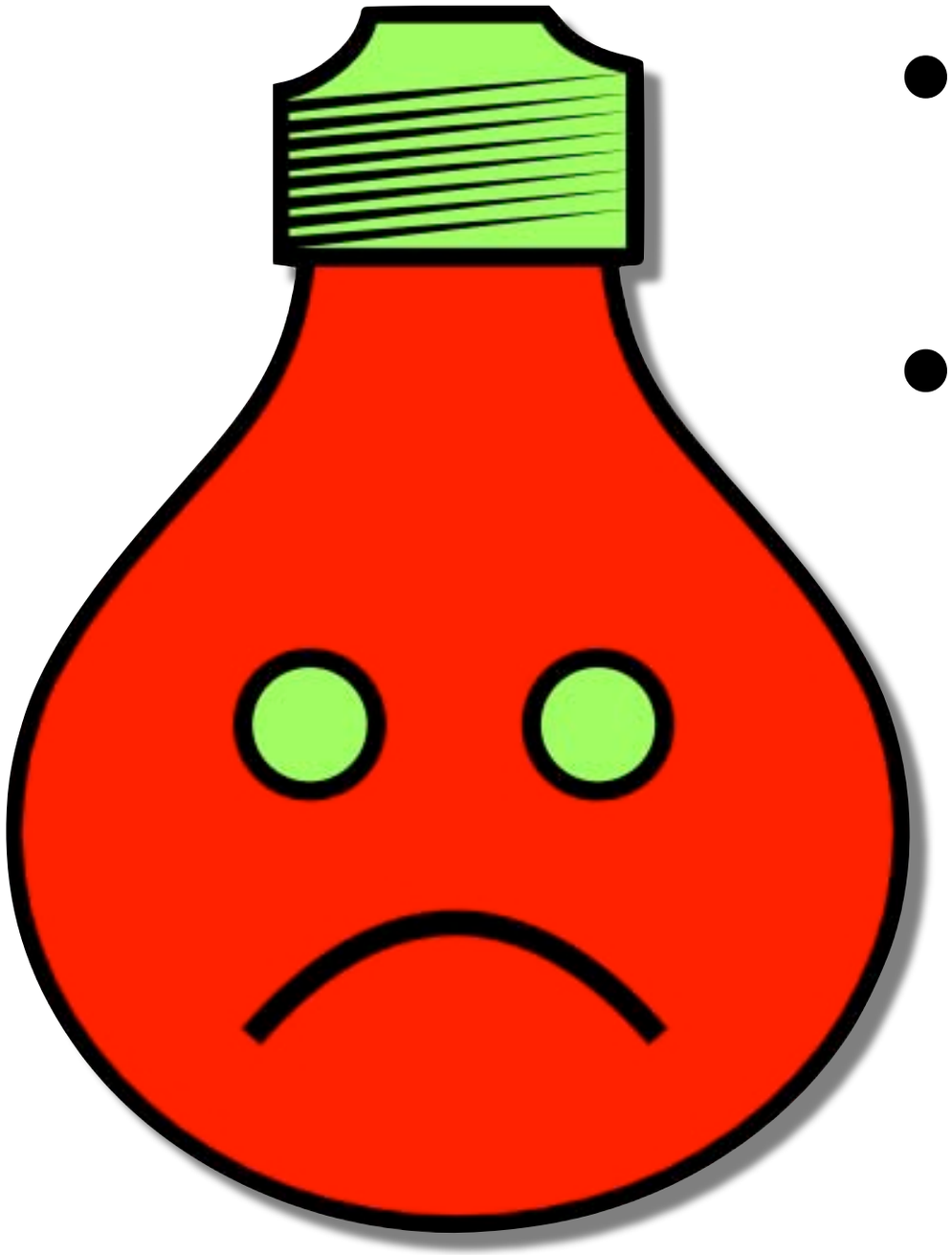
Object Lifetime and Calling

- Object Instantiated
- Object Passed In
- Global Object





- How Do You Write Hard To Test Code?



- How Do You Write Hard To Test Code?
- You mix object creation code with business logic. This will assure that a test can never construct a graph of objects different from production. Hence nothing can be tested in isolation.

Cost of Construction

- To test a method you first need to instantiate an object:
  - Work inside of constructor has no seams
    - Can't override
    - Your test must successfully navigate the constructor maze
- Do as little work in constructor as possible

Cost of Construction

```
class Document {
  String html;

  Document(String url) {
    HtmlClient client = new HtmlClient();
    html = client.get(url);
  }
}
```

Cost of Construction

```
class Document {
  String html;

  Document(String url) {
    HtmlClient client = new HtmlClient();
    html = client.get(url);
  }
}
```

Mixing graph construction with work

Cost of Construction

```
class Document {
  String html;

  Document(String url) {
    HtmlClient client = new HtmlClient();
    html = client.get(url);
  }
}
```

Mixing graph construction with work

Doing work in constructor

Cost of Construction

```
class Document {
  String html;

  Document(HtmlClient client, String url) {
    html = client.get(url);
  }
}
```

Cost of Construction

```
class Document {
  String html;

  Document(HtmlClient client, String url) {
    html = client.get(url);
  }
}
```

Doing work in constructor

Cost of Construction

```
class Document {
  String html;

  Document(HtmlClient client, String url) {
    html = client.get(url);
  }
}
```

Document does not care about client, it cares about what client can produce

Doing work in constructor

Cost of Construction

```
class Document {
  String html;

  Document(HtmlClient client, String url) {
    html = client.get(url);
  }
}
```

Document does not care about client, it cares about what client can produce

Doing work in constructor

Law of the Demeter: Asking for something you don't need directly only to get to what you really want.

Google

Cost of Construction

```
class Car {
  Engine engine;

  Car(String modelNo) {
    EngineFactory factory
                = new EngineFactory();
    engine = factory.get(modelNo);
  }
}
```

Cost of Construction

```
class Car {
  Engine engine;

  Car(String modelNo) {
    EngineFactory factory
                = new EngineFactory();
    engine = factory.get(modelNo);
  }
}
```

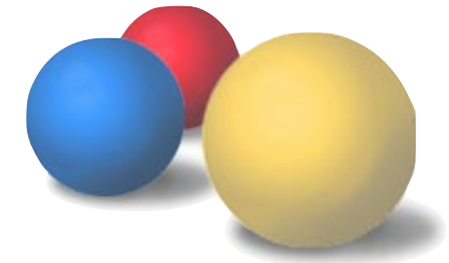
The fact that a Car knows about EngineFactory Seems silly. But somehow if car is a Document it is ok for Document to be its own factory.

Cost of Construction

Cost of Construction

```
class Document {
  String html;

  Document(String html) {
    this.html = html;
  }
}
```

Cost of Construction



```
class Document {
  String html;

  Document(String html) {
    this.html = html;
  }
}
```

```
class DocumentFactory {
  HtmlClient client;

  DocumentFactory(HtmlClient client) {
    this.client = client;
  }

  Document build(String url) {
    return new Document(client.get(url));
  }
}
```

Google

Cost of Construction

```
class Document {
  String html;

  Document(String html) {
    this.html = html;
  }
}
```

```
class Printer {

  void print(Document html) {
    // do some work here.
  }
}
```

```
class DocumentFactory {
  HtmlClient client;

  DocumentFactory(HtmlClient client) {
    this.client = client;
  }

  Document build(String url) {
    return new Document(client.get(url));
  }
}
```

Cost of Construction

```
class Document {
  String html;

  Document(String html) {
    this.html = html;
  }
}
```

```
class Printer {

  void print(Document html) {
    // do some work here.
  }
}
```

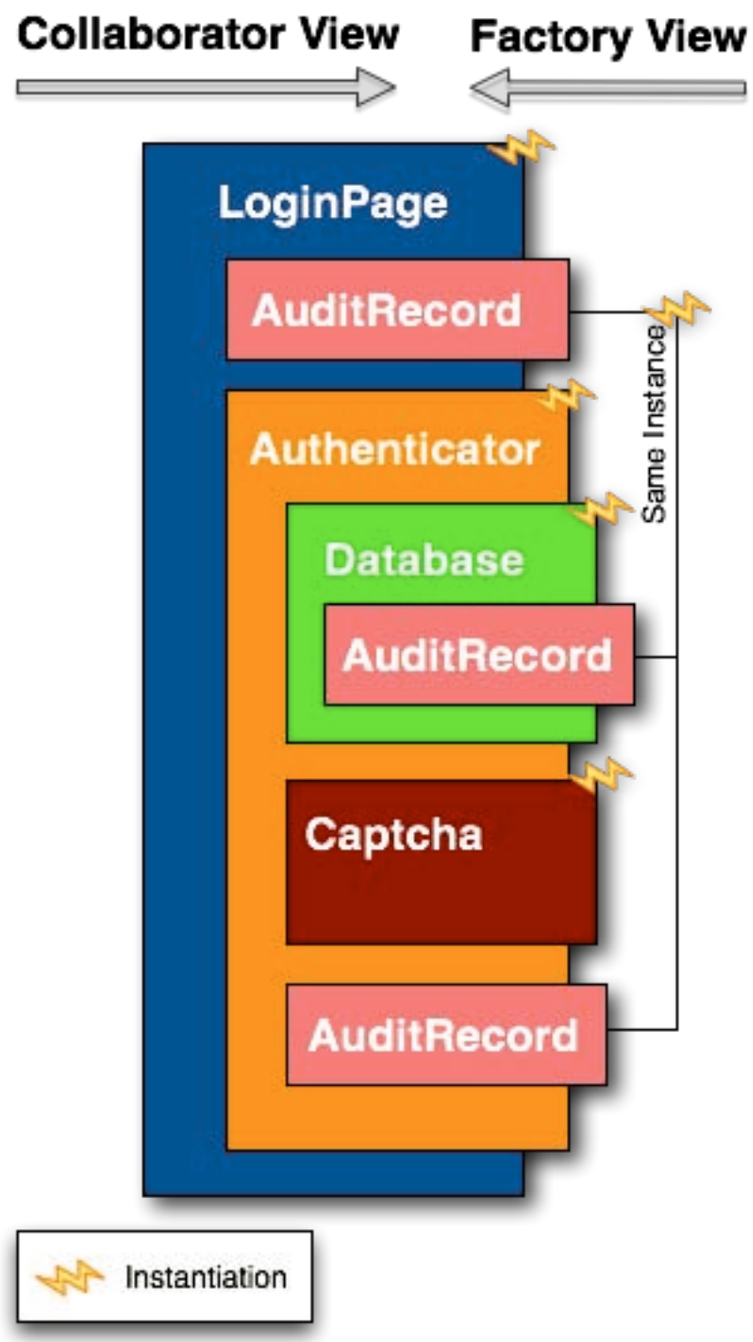
Easy to test
since Document is easy to
construct

```
class DocumentFactory {
  HtmlClient client;

  DocumentFactory(HtmlClient client) {
    this.client = client;
  }

  Document build(String url) {
    return new Document(client.get(url));
  }
}
```

Cost of Construction

- Test has to successfully navigate the constructor each time instance is needed
- Objects require construction often indirectly making hard to construct objects a real pain to test with

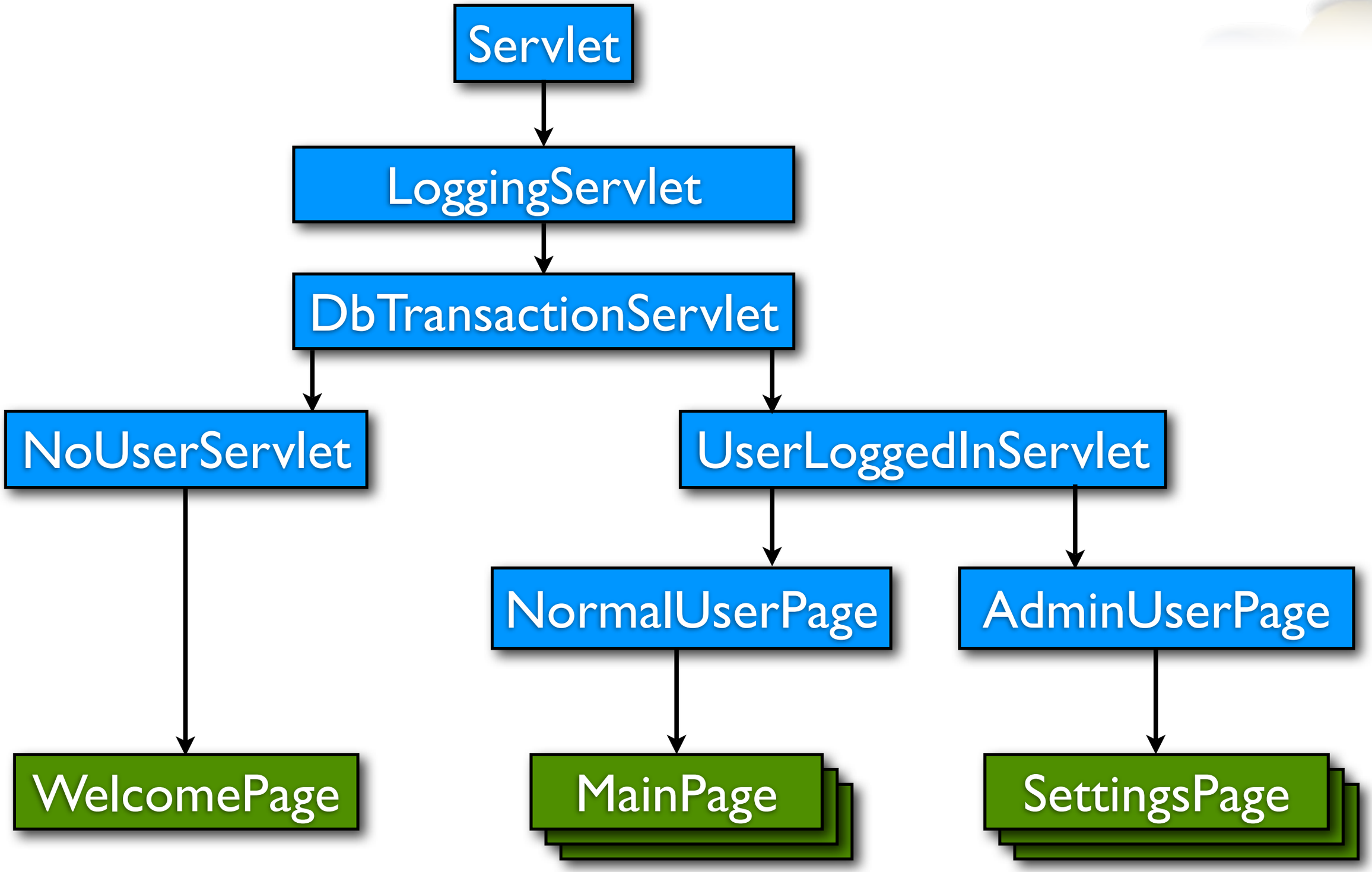
Wiring


Collaborator View
Factory View
LoginPage
AuditRecord
Authenticator
Database
AuditRecord
Captcha
AuditRecord
Same Instance
Instantiation

Composition vs Inheritance

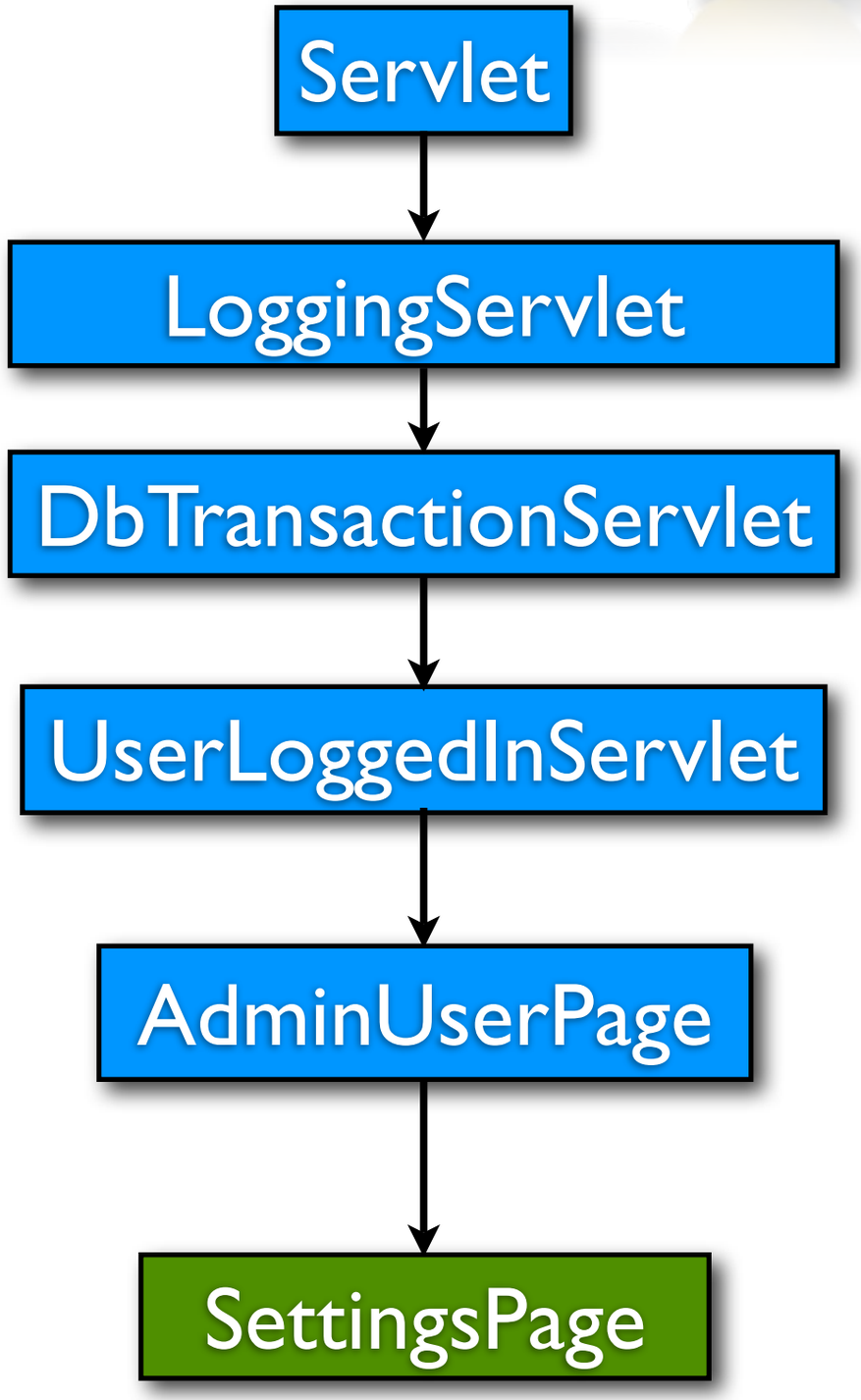
- The purpose of Inheritance is polymorphic behavior
- If you don't take advantage of polymorphism you should reuse code through delegation / composition

Composition vs Inheritance



Composition vs Inheritance

```
class SettingsPageTest extends TestCase {
  public void testAddUser() {
    SettingsPage p = new SettingsPage();
    // What about Logging?
    // What about Database?
    // What about User Verification?
    // What about Admin User Verification?
    // How do I inject mocks into this?
    HttpServletRequest req = ....?
    HttpServletResponse res = ...?
    // What parameters => Add User Action?
    p.doGet(req, resp);
    // What do I assert?

    // This test is not unit test!
    // Failed test => No clue why!
  }
}
```



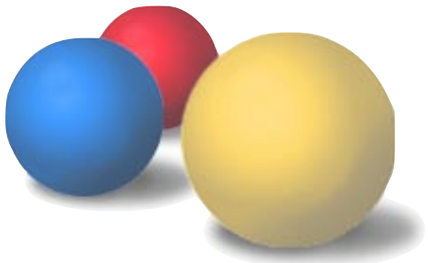
Composition vs Inheritance

- With composition at test time we **can** build different object graphs under tests.

- With inheritance at test time we **can not** build different object inheritance!

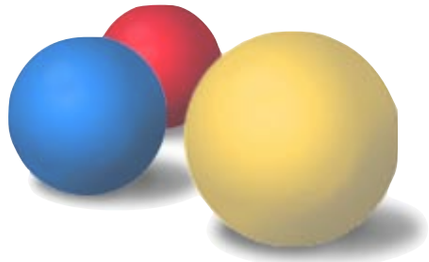
Composition vs Inheritance

Composition vs Inheritance



Servlet	LoggingServlet	DbTransactionServlet	UserLoggedInServlet	AdminUserPage	SettingsPage

Composition vs Inheritance

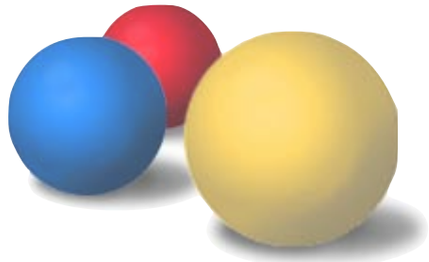


Servlet	LoggingServlet	DbTransactionServlet	UserLoggedInServlet	AdminUserPage	SettingsPage

```
public void testAddUser() {
  SettingsPage p = new SettingsPage();
  HttpServletRequest req = ....?
  HttpServletResponse res = ...?
  // What parameters => Add User Action?
  p.doGet(req, resp);
  // What do I assert?
}
```

Google

Composition vs Inheritance

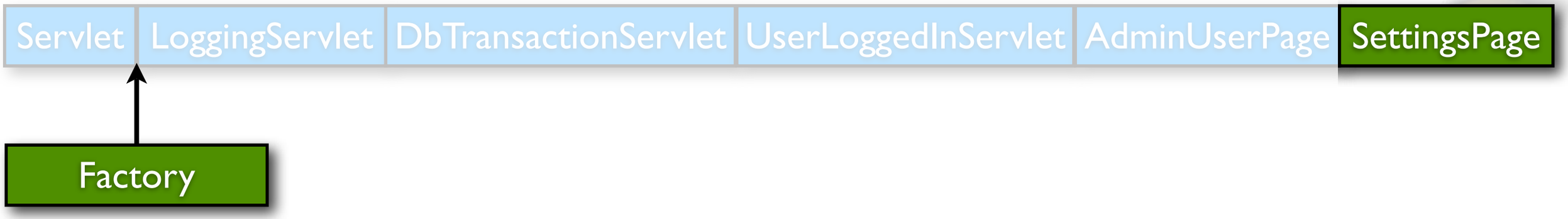


Servlet	LoggingServlet	DbTransactionServlet	UserLoggedInServlet	AdminUserPage	SettingsPage

```
public void testAddUser() {
  SettingsPage p = new SettingsPage();
  HttpServletRequest req = ....?
  HttpServletResponse res = ...?
  // What parameters => Add User Action?
  p.doGet(req, resp);
  // What do I assert?
}

public void testAddUser() {
  UserRepository users = new InMemoryUserRepository();
  SettingsPage page = new SettingsPage(users);
  page.addUser("jon");
  assertNotNull(users.getUser("jon"))
}
```

Google

Composition vs Inheritance



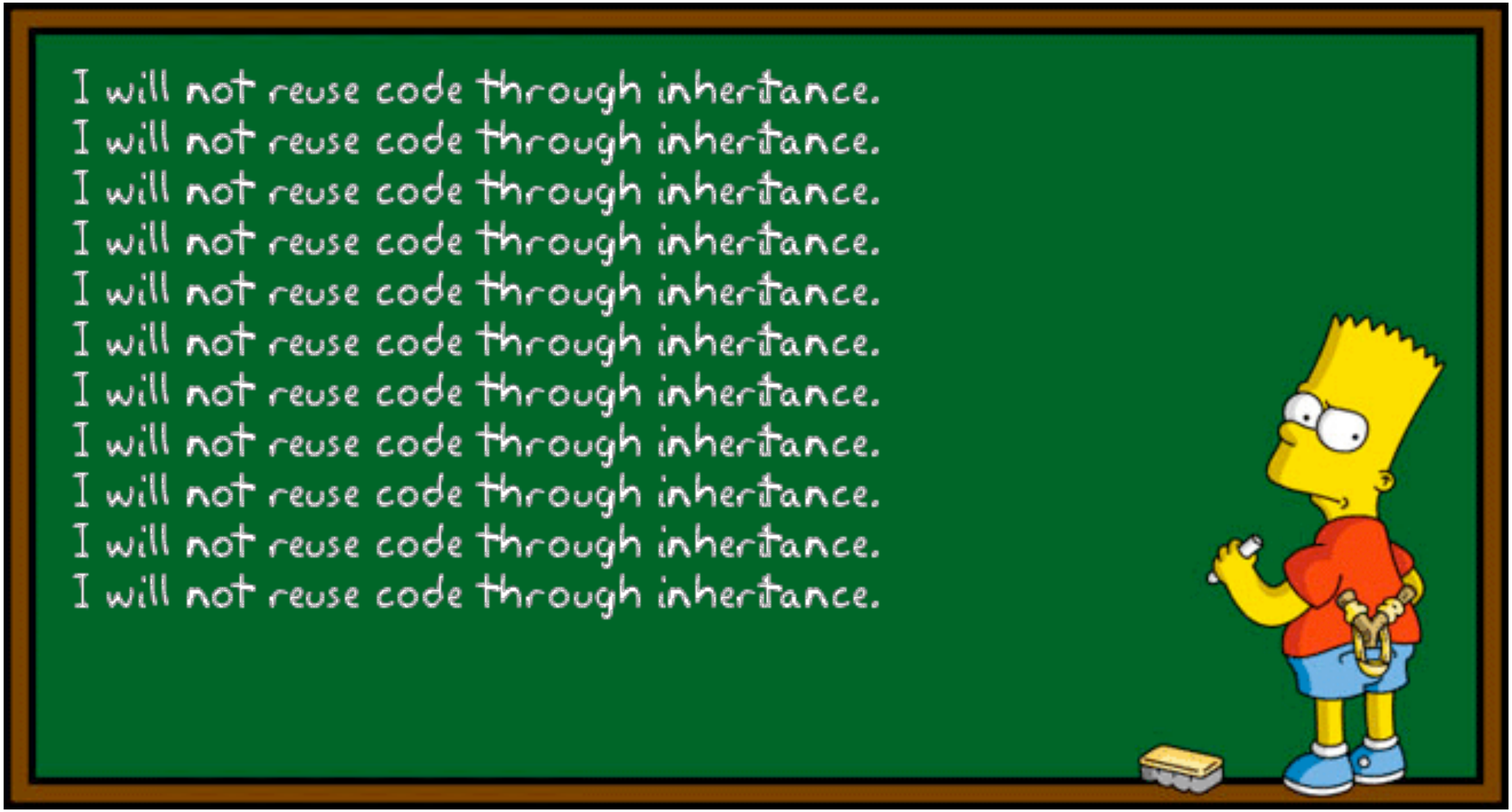

```
public void testAddUser() {
  SettingsPage p = new SettingsPage();
  HttpServletRequest req = ....?
  HttpServletResponse res = ...?
  // What parameters => Add User Action?
  p.doGet(req, resp);
  // What do I assert?
}

public void testAddUser() {
  UserRepository users = new InMemoryUserRepository();
  SettingsPage page = new SettingsPage(users);
  page.addUser("jon");
  assertNotNull(users.getUser("jon"))
}
```

Google

Composition vs Inheritance

- There are no seams in the inheritance hierarchy. It is all or nothing proposition, which makes **Unit** Testing impossible.

Composition vs Inheritance

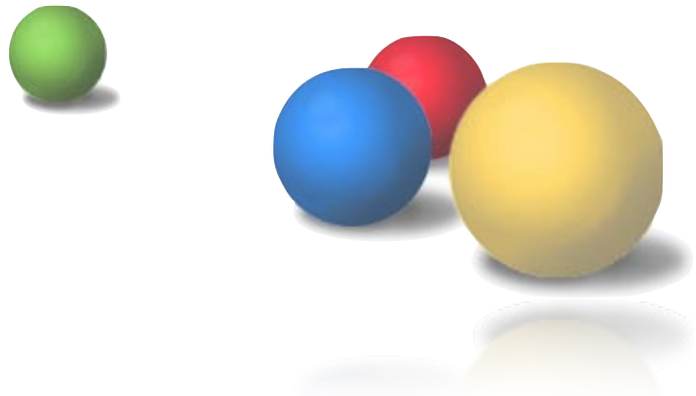


Sub classing in tests

- Anonymous inner subclass and override is the ultimate in swiss army knife of testing.
  - It is a code-smell
- Subclassing for tests, begs for whatever you are subclassing to live in a different object. So that in test you can replace that portion with friendly

Google

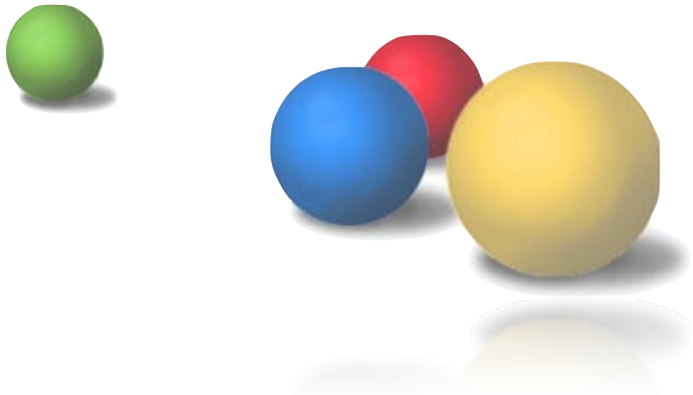
Sub classing in tests



```
class LoginPage {
  public void login(String user, String password){
    User user = loadUser(user);
    if (!user.getPassword.equals(password)) {
      throw new InvalidPassword();
    }
  }
  // protected for test access
  protected User loadUser(String user) { ... }
}

testLogin() {
  final User u = new User("joe", "pwd");
  LoginPage lp = new LoginPage() {
    protected User loadUser(String user) {
      return u;
    }
  }
  lp.login("joe", "pwd");
}
```

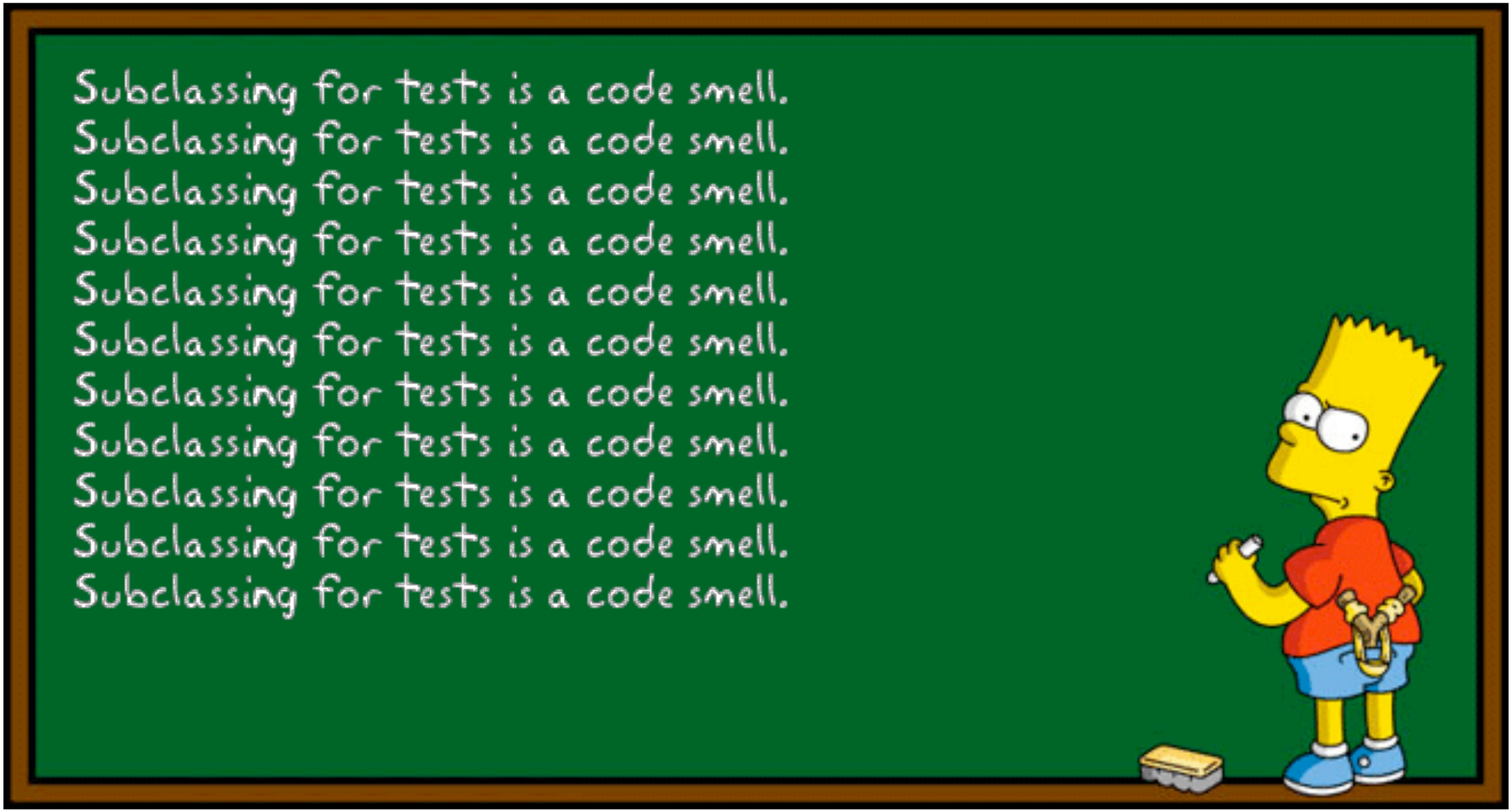
Google

Sub classing in tests



```
class LoginPage {
  LoginPage(UserRepo userRepo){...}
  public void login(String user, String password){
    User user = userRepo.getUserByName(user);
    if (!user.getPassword.equals(password)) {
      throw new InvalidPassword();
    }
  }
}

testLogin() {
  User u = new User(“joe”, “pwd”);
  UserRepo repo = new InMemoryUserRepo()
  repo.addUser(u);
  LoginPage lp = new LoginPage(repo);
  lp.login(“joe”, “pwd”);
}
```

Sub classing in tests



Law of Demeter

Law of Demeter

- Imagine your are in a store and the item you are purchasing is $25.

Law of Demeter

- Imagine your are in a store and the item you are purchasing is $25.
- Do you give the clerk $25?

Law of Demeter

- Imagine your are in a store and the item you are purchasing is $25.
- Do you give the clerk $25?
- Or do you give the clerk your wallet and let him retrieve the $25?

Law of Demeter

```
class Goods {
  AccountsReceivable ar;
  void purchase(Customer c) {
    Money m = c.getWallet().getMoney();
    ar.recordSale(this, m);
  }
}
```

Law of Demeter

```
class Goods {
  AccountsReceivable ar;
  void purchase(Customer c) {
    Money m = c.getWallet().getMoney();
    ar.recordSale(this, m);
  }
}
```

To test this we need to create a valid Customer with a valid Wallet which contains the real item of interest. (Money)

Law of Demeter

```
class GoodsTest {
  void testPurchase() {
    AccountsReceivable ar = new MockAR();
    Goods g = new Goods(ar);
    Money m = new Money(25, USD);
    Wallet v = new Wallet(m);
    Customer c = new Customer(v);
    g.purchase(c);
    assertEquasl(25, ar.getSales());
  }
}
```

Law of Demeter

```
class Goods {
  AccountsReceivable ar;
  void purchase(Money m) {
    ar.recordSale(this, m);
  }
}

class GoodsTest {
  void testPurchase() {
    AccountsReceivable ar = new MockAR();
    Goods g = new Goods(ar);
    g.purchase(new Money(25, USD));
    assertEquasl(25, ar.getSales());
  }
}
```

Law of Demeter

Law of Demeter

- You only ask for objects which you directly need (operate on)

Law of Demeter

- You only ask for objects which you directly need (operate on)
- a.getX().getY().... is a dead givaway

Law of Demeter

- You only ask for objects which you directly need (operate on)
- a.getX().getY().... is a dead givaway
- serviceLocator.getService() is breaking the Law of Demeter

Global State

Lets test a class

```
class A {
    private HardDisk d;
    public A() {
        d = new HardDisk();
    }

    public void doWork() {
        d.open();
        d.format(FAT);
        d.close();
    }
}
```

We can't test this

```
class A {
    private HardDisk d;
    public A() {
        d = new HardDisk();
    }

    public void doWork() {
        d.open();
        d.format(FAT);
        d.close();
    }
}
```

- There is no way to intercept the calls to HardDisk
- The only way to test this is to actually format your hard disk and have a post condition

There is only one HardDisk

```
class B {
    private HardDisk d;
    public B() {
        d = HardDisk.get();
    }

    public void doWork() {
        d.open();
        d.format(FAT);
        d.close();
    }
}
```

- Since there is only one HardDisk we might need to enforce “Singleton-ess”
- Still can’t test this as we can’t intercept calls to HardDisk

Lets try to write a test

```
class B {
   private HardDisk d;
   public B() {
      d = HardDisk.get();
   }

   public void doWork() {
      d.open();
      d.format(FAT);
      d.close();
   }
}
```

```
testBFormatsHD() {
   HardDisk.init("/temp");
   new B().doWork();
   assert???(...);
}

testBFormatsHD2() {
   // we forgot to init
   // guess the default is
   // /dev/hd0 :-)

   new B().doWork();
   // Your machine is gone
}
```

Calling init() multiple times on a Singleton is a bit weird

Configurable Singleton

```
class B {
   private HardDisk d;
   public B() {
      d = HardDisk.get();
   }

   public void doWork() {
      d.open();
      d.format(FAT);
      d.close();
   }
}
```

```
testBFormatsHD() {
   MockHD hd = new MockHD();
   HardDisk.set(hd);

   new C().doWork();

   assertTrue(hd.verify());
}
```

Its not a singleton if you can change it!

What about Service Locator

- To test our singleton we need
  - Ability to change the “instance”
  - Ability to configure the singleton multiple times
- Both of the above go against what a singleton is
- Maybe Service Locator / Registry is better

We need to intercept the HD

```
class C {
    private HardDisk d;
    public C() {
        d = ServiceLocator.get(HardDisk.class);
    }

    public void doWork() {
        d.open();
        d.format(FAT);
        d.close();
    }
}
```

Now we can intercept the calls

```
class C {
  private HardDisk d;
  public C() {
    d = ServiceLocator
    .get(HardDisk.class);
  }

  public void doWork() {
    d.open();
    d.format(FAT);
    d.close();
  }
}
```

```
testCFormatsHD() {
  MockHD hd = new MockHD();
  ServiceLocator
    .registerHardDisk(hd);

  new C().doWork();

  assertTrue(hd.verify());
}
```

There is something weird

```
testCFormatsHD() {
   MockHD hd = new MockHD();
   ServiceLocator
      .RegisterHardDisk(hd);

   new C().doWork();

   assertTrue(hd.verify());
}
```

- Red and green areas share no common variables!
  - How do they talk?
  - Are they related?
  - Does order matter?
  - What if the red is in setUp() in super-class?

API lies

- Class API hides true dependencies
  - Constructor mentions nothing of HardDisk
  - Methods mention nothing of HardDisk
- Attempting to use class C outside of HardDisk will fail.
- Attempting to reuse the component in a different project will fail because it will drag dependencies with it.
- Dependencies are there but are **HIDDEN**!

Making the dependencies explicit

```
class D {
    private HardDisk hd;
    public D(HardDisk hd) {
        this.hd = hd;
    }

    public void doWork() {
        hd.open();
        hd.format(FAT);
        hd.close();
    }
}
```

Testing explicit dependencies is easier

```
class D {
    private HardDisk d;
    public D(HardDisk d) {
        this.d = d;
    }

    public void doWork() {
        d.open();
        d.format(FAT);
        d.close();
    }
}
```

```
testDFormatsHD() {
    MockHD hd = new MockHD();

    new D(hd).doWork();

    assertTrue(hd.verify());
}
```

- Dependencies are **EXPLICIT**!

Explicit Dependencies

- Also known as:
  - Dependency Injection
  - Inversion of Control

- Make the order of initialization clear
- Don’t pretend to be cleaner then they are
- WY(Declare)IWY(Need) -- No hiding

Difficulty of Testing

```
class A {
   private HardDisk d;
   public A() {
      d = new HardDisk();
   }

   public void doWork() {
      d.open();
      d.format(FAT);
      d.close();
   }
}
```

Construction

```
class B {
   private HardDisk d;
   public B() {
      d = HardDisk.get();
   }

   public void doWork() {
      d.open();
      d.format(FAT);
      d.close();
   }
}
```

Singleton

```
class C {
  private HardDisk d;
  public C() {
    d = ServiceLocator
    .get(HardDisk.class);
  }

  public void doWork() {
    d.open();
    d.format(FAT);
    d.close();
  }
}
```

Service

```
class D {
   private HardDisk d;
   public D(HardDisk d) {
      this.d = d;
   }

   public void doWork() {
      d.open();
      d.format(FAT);
      d.close();
   }
}
```

Injection

Testable == Understandable

Easy to Test --> Easy to Understand

Hard to Test --> Hard to Understand

Why?

What is Testability?

“A testability is directly proportional to the number of locations where we can intercept the normal flow of the code”

Interception == Polymorphism

Would that mean that static methods are harder to tests then instance methods because they can not be intercepted?

Interception possibilities

```
class A {
    private HardDisk d;
    public A() {
        d = new HardDisk();
    }

    public void doWork() {
        d.open();
        d.format(FAT);
        d.close();
    }
}
```

```
class D {
    private HardDisk d;
    public D(HardDisk d) {
        this.d = d;
    }

    public void doWork() {
        d.open();
        d.format(FAT);
        d.close();
    }
}
```

Constructors are static

- Static constructor prevents interception
- Statically constructed object is equivalent to non-intercept-able objects
  - Why?
    - Because interception involves
      - sub-classing / alternate-implementation

Interception possibilities

```
class A {
    private HardDisk d;
    public A() {
        d = new HardDisk();
    }

    public void doWork() {
        d.open();
        d.format(FAT);
        d.close();
    }
}
```

```
class D {
    private HardDisk d;
    public D(HardDisk d) {
        this.d = d;
    }

    public void doWork() {
        d.open();
        d.format(FAT);
        d.close();
    }
}
```

Service Locator

- So if new is static (prevents interception)
  - THEN
    - ServiceLocator is good as it allows easy object substitution and interception

- True
  - BUT
    - Hides dependencies

Service Locator

- aka Context
- Better then a Singleton
  - If you had static look up of services this is an improvement. It is testable but it is not pretty
- Hides true dependencies

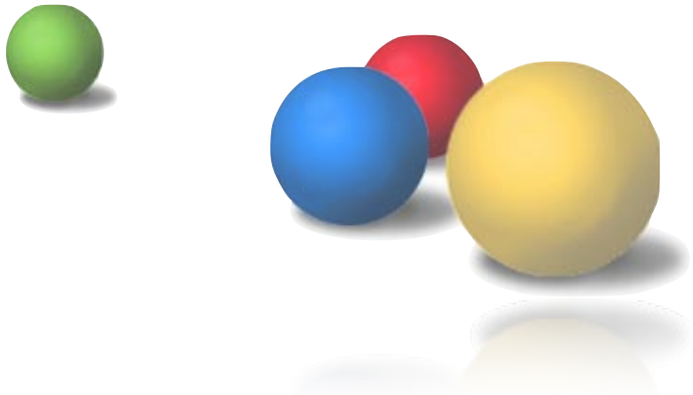
Service Locator

```
class House {

  House(Locator locator){

    }
}
```

What needs to be mocked out in test?

Google

Service Locator



```
class House {
  Door door;
  Window window;
  Roof roof;

  House(Locator locator){
    door = locator.getDoor();
    window = locator.getWindow();
    roof = locator.getRoof();
  }
}
```

Service Locator

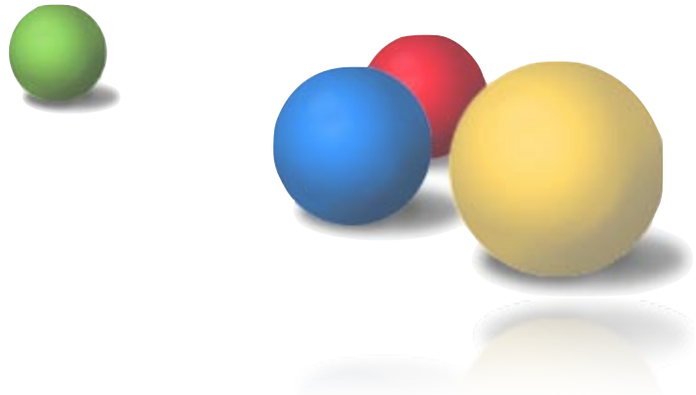
```
class House {
  Door door;
  Window window;
  Roof roof;

  House(Locator locator){
    door = locator.getDoor();
    window = locator.getWindow();
    roof = locator.getRoof();
  }
}
```

The API lies about its true dependencies. Only after examining or running the code can we determine what is actually needed

Service Locator

```
class House {
  Door door;
  Window window;
  Roof roof;

  House(Door d, Window w, Roof r){
    door = d;
    window = w;
    roof = r;
  }
}
```

Service Locator



```
class HouseTest {
  pulic void testServiceLocator() {
    Door d = new Door(...);
    Roof r = new Roof(...);
    Window w = new Window(...);
    House h = new House(d, r, w);
  }
}
```

Google

Service Locator

- Mixing Responsibilities
  - Lookup
  - Factory
- Need to have an interface for testing
- Anything which depends on Service Locator now depends on everything else.

Global State aka Singletons

- API that lies about what it needs
- Spooky action at a distance

Deceptive API

```
testCharge() {
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

Deceptive API

```
testCharge() {
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

- At the end of the month I got my Statement!

Deceptive API

```
testCharge() {
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

- At the end of the month I got my Statement!
- I was out $100!

Deceptive API

```
testCharge() {
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

- At the end of the month I got my Statement!
- I was out $100!
- Spooky action at a distance!

Deceptive API

```
testCharge() {
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

- At the end of the month I got my Statement!
- I was out $100!
- Spooky action at a distance!
- It never passed in isolation

Deceptive API

```
testCharge() {
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

Deceptive API

```
testCharge() {
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

```
java.lang.NullPointerException
   at talk3.CreditCard.charge(CreditCard.java:48)
```

Deceptive API

```
testCharge() {
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

Deceptive API

```
testCharge() {
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

```
java.lang.NullPointerException
  at talk3.CreditCartProcessor.init(CreditCardProcessor.java:146)
```

Deceptive API

```
testCharge() {
  OfflineQueue.start();
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

Deceptive API

```
testCharge() {
  OfflineQueue.start();
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

```
java.lang.NullPointerException
  at talk3.OfflineQueue.start(OfflineQueue.java:16)
```

Deceptive API

```
testCharge() {
  Database.connect(...);
  OfflineQueue.start();
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

Deceptive API

```
testCharge() {
  Database.connect(...);
  OfflineQueue.start();
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

- CreditCard API lies
  - It pretends to not need the CreditCardProcessor even thought in reality it does.

Deceptive API

```
testCharge() {
  Database.connect(...);
  OfflineQueue.start();
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

- CreditCard API lies
  - It pretends to not need the CreditCardProcessor even thought in reality it does.

Deceptive API

```
testCharge() {
  Database.connect(...);
  OfflineQueue.start();
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

- CreditCard API lies
  - It pretends to not need the CreditCardProcessor even thought in reality it does.

Deceptive API

```
testCharge() {
  Database.connect(...);
  OfflineQueue.start();
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```

- CreditCard API lies
  - It pretends to not need the CreditCardProcessor even thought in reality it does.

Better API

```
testCharge() {

  CreditCard cc;
  cc = new CreditCard("12..34", ccProc);
  cc.charge(100);
}
```

Better API

```
testCharge() {


  ccProc = new CCProcessor(queue);
  CreditCard cc;
  cc = new CreditCard(“12..34”, ccProc);
  cc.charge(100);
}
```

Better API

```
testCharge() {

  queue = new OfflineQueue(db);
  ccProc = new CCProcessor(queue);
  CreditCard cc;
  cc = new CreditCard("12..34", ccProc);
  cc.charge(100);
}
```

Better API

```
testCharge() {
  db = new Database(...);
  queue = new OfflineQueue(db);
  ccProc = new CCProcessor(queue);
  CreditCard cc;
  cc = new CreditCard("12..34", ccProc);
  cc.charge(100);
}
```

Better API

```
testCharge() {
  db = new Database(...);
  queue = new OfflineQueue(db);
  ccProc = new CCProcessor(queue);
  CreditCard cc;
  cc = new CreditCard("12..34", ccProc);
  cc.charge(100);
}
```

- Dependency injection enforces the order of initialization at compile time.

Code Review

My advice to you...

My advice to you...

- **Global state is evil!**
  - *Stay away at all costs!*

My advice to you...

- **Global state is evil!**
  - *Stay away at all costs!*
- `new` **operator is like kryptonite!**
  - *Handle with extreme care!*

Ideal Interface

Ideal Interface

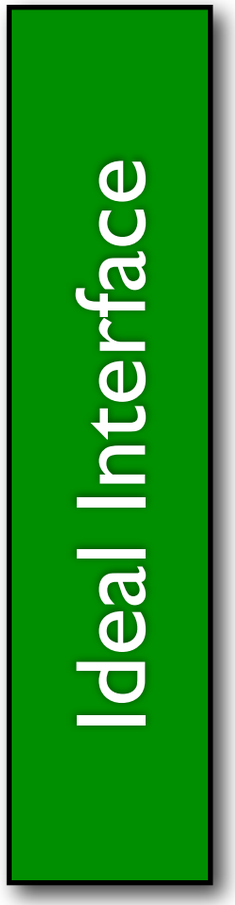
Complex Interface

Ideal Interface

Complex Third Party API; lots of unneeded methods which return objects which are not quite what we want and need to be marshaled

Ideal Interface

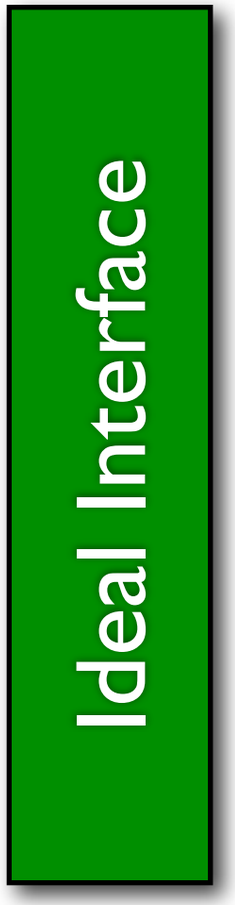
Complex Third Party API; lots of unneeded methods which return objects which are not quite what we want and need to be marshaled



Ideal Interface

Simplified API for your application which returns the application value objects. This interface becomes great place to insert a fake implementation for scenario testing.

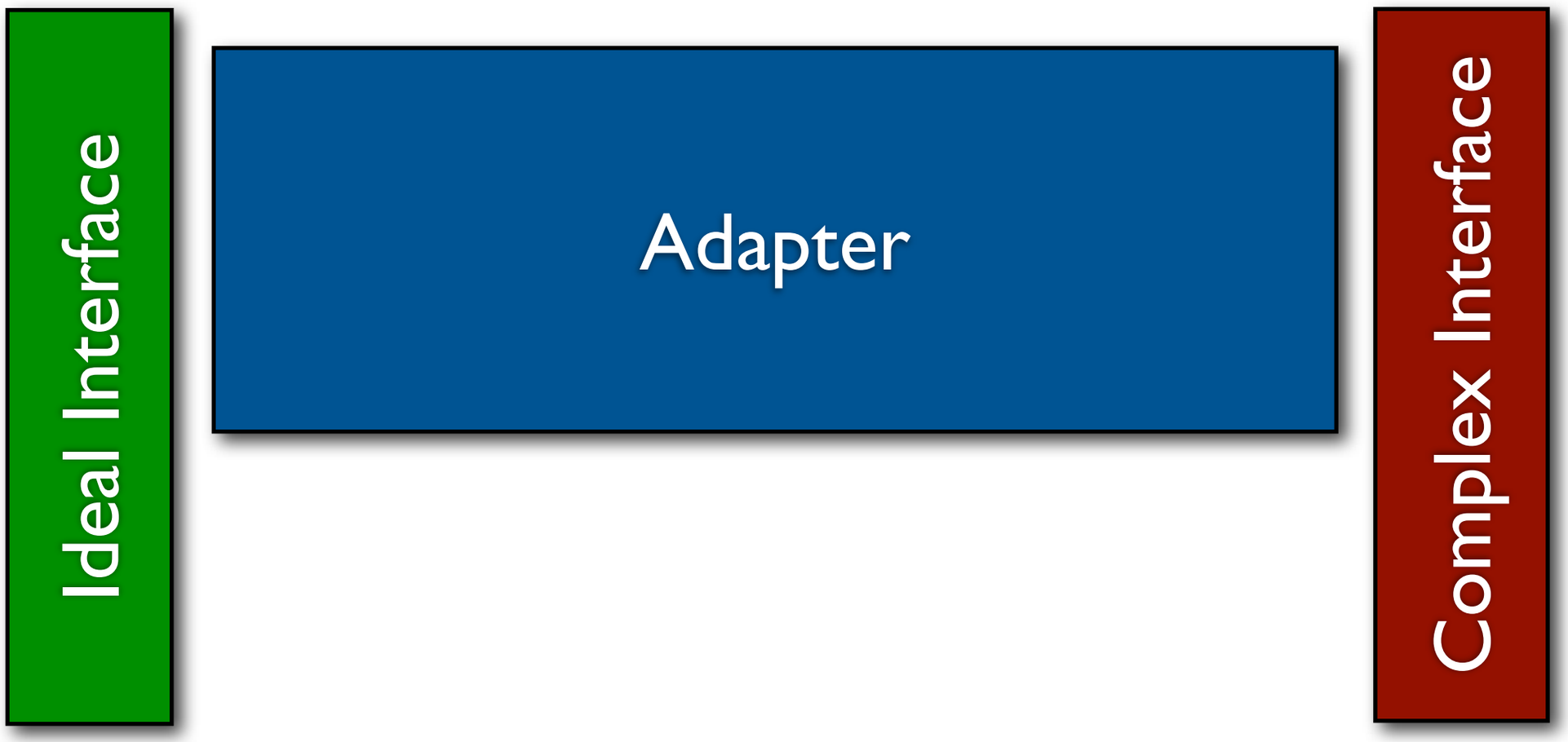
Complex Third Party API; lots of unneeded methods which return objects which are not quite what we want and need to be marshaled



Ideal Interface

Simplified API for your application which returns the application value objects. This interface becomes great place to insert a fake implementation for scenario testing.

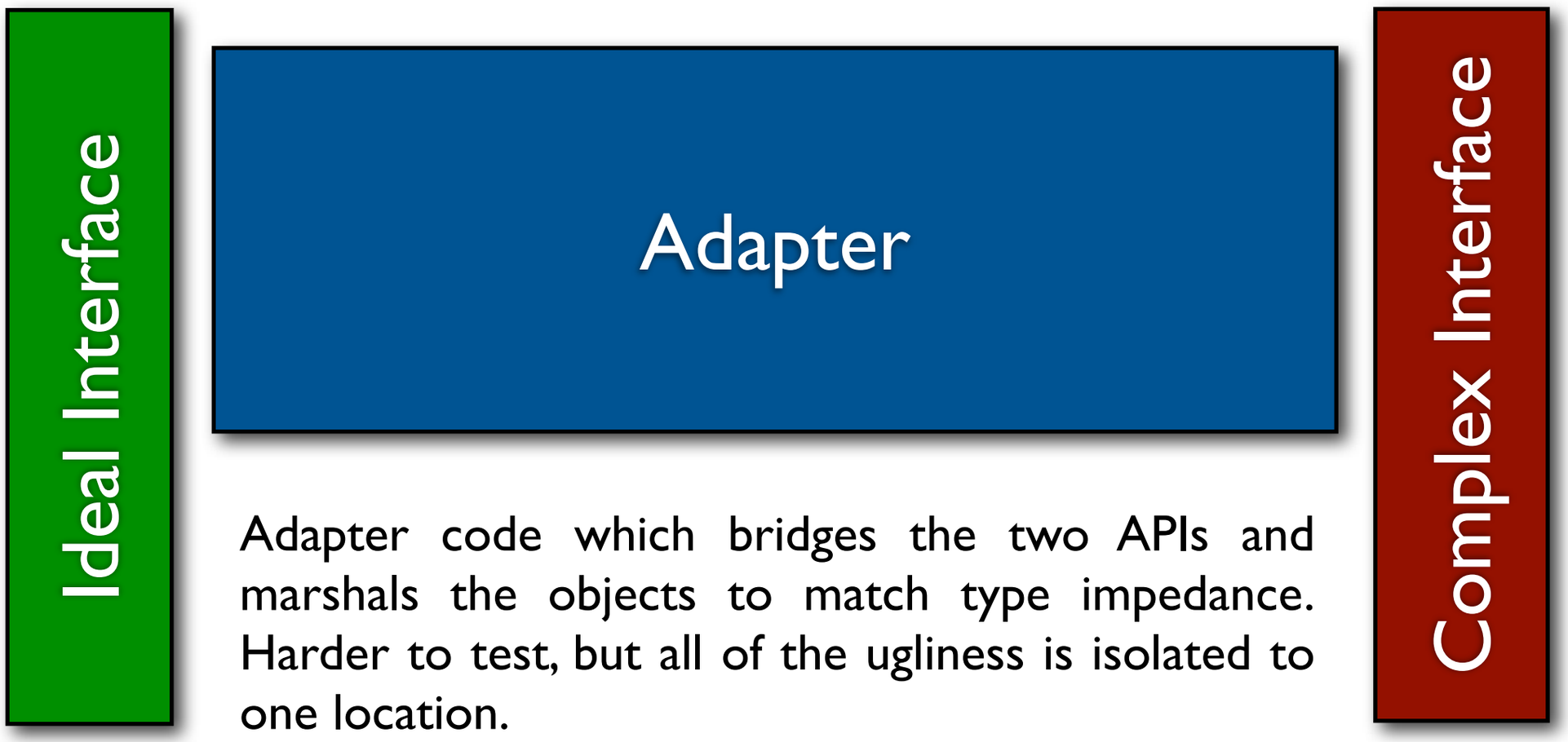
Complex Third Party API; lots of unneeded methods which return objects which are not quite what we want and need to be marshaled



Ideal Interface

Simplified API for your application which returns the application value objects. This interface becomes great place to insert a fake implementation for scenario testing.

Complex Third Party API; lots of unneeded methods which return objects which are not quite what we want and need to be marshaled



Adapter code which bridges the two APIs and marshals the objects to match type impedance. Harder to test, but all of the ugliness is isolated to one location.