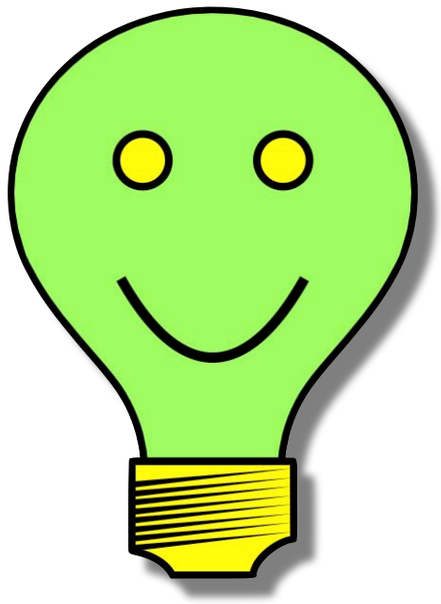
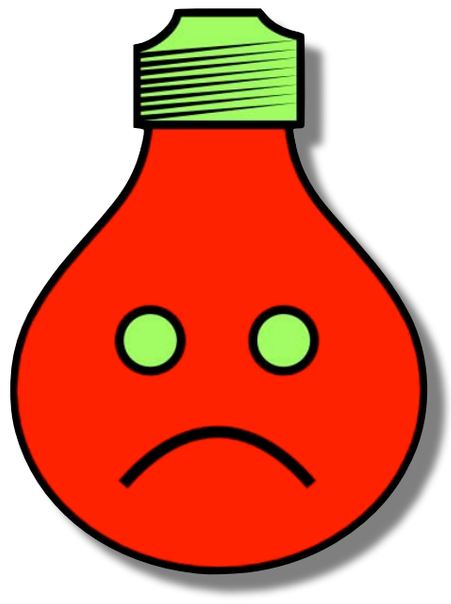
# Unit Testing 101

by Miško Hevery
misko@hevery.com
http://www.TestabilityExplorer.org

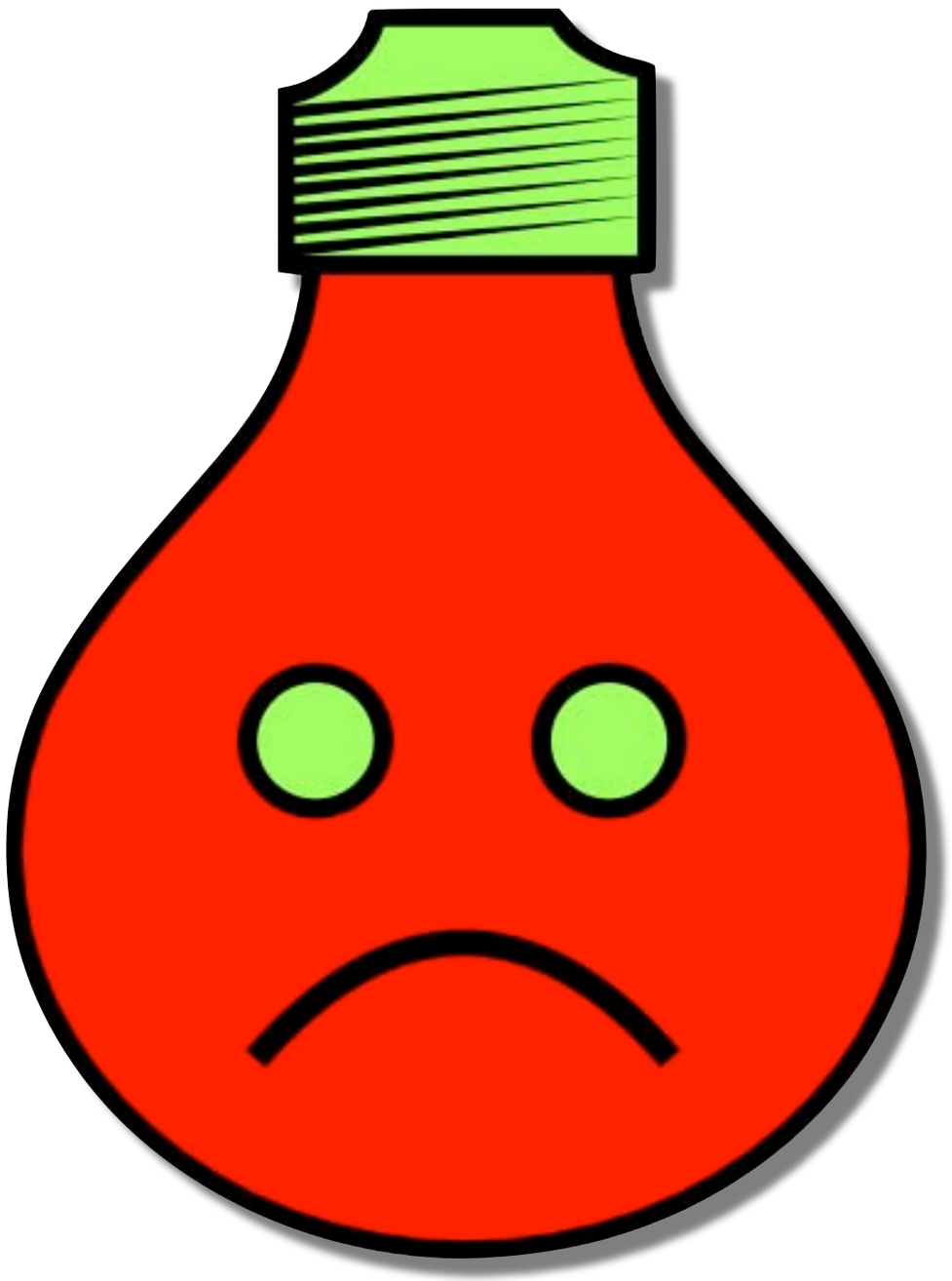
# Mr. Testabulb
# vs.
# Mr. Untestabulb

Testing a Flashlight
Writing Untestable Code





by Miško Hevery



- How Do You Write Hard To Test Code?

Google

# What I know about...

writing tests

(nothing)

Google

# What I know about... writing testable code

- Good OO
- Dependency Injection
- Test Driven Development
- A whole lot about un-testable code

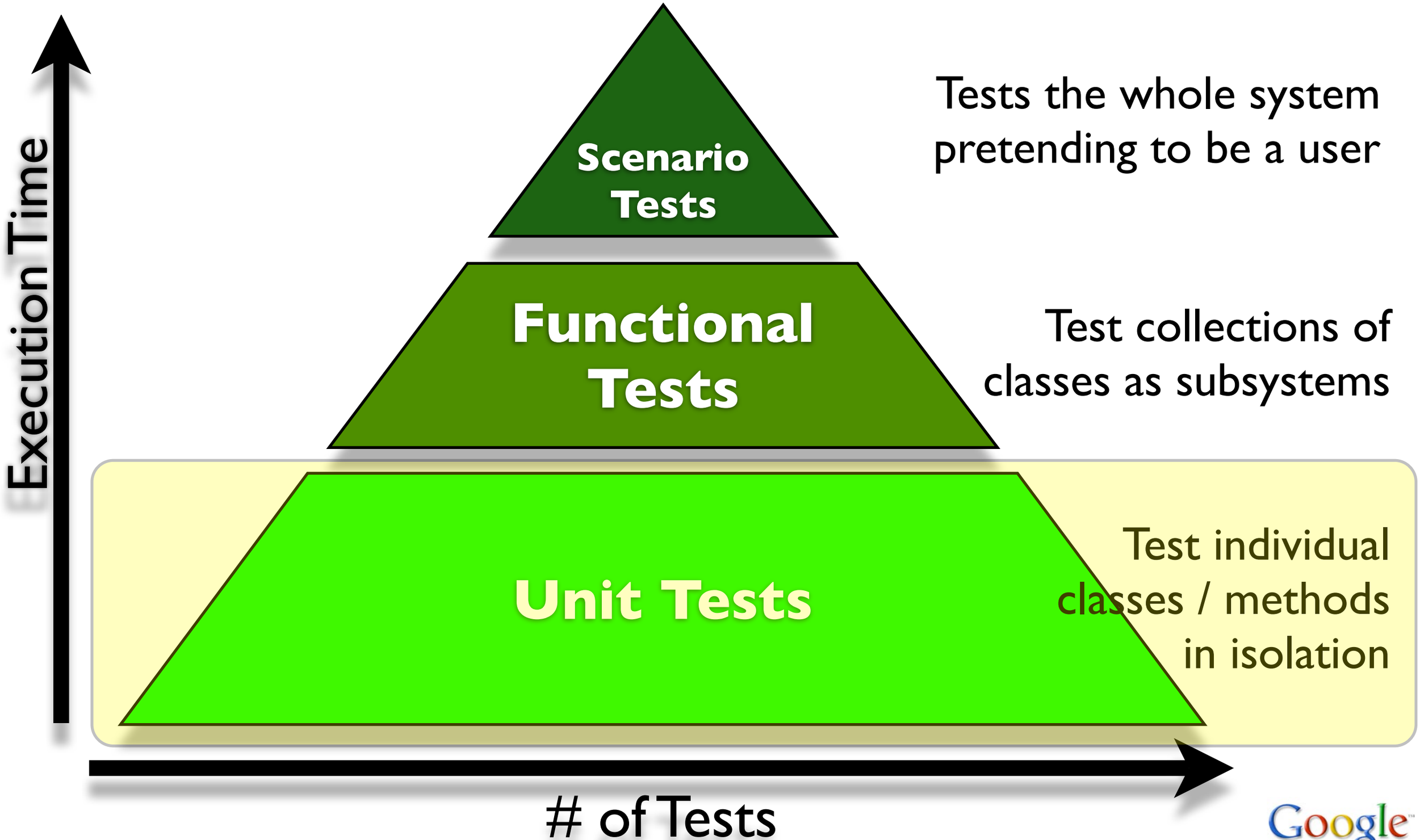
Google

There is no secret to writing tests...

... there are only secrets to writing testable code!

Google

# Different Kinds of Tests



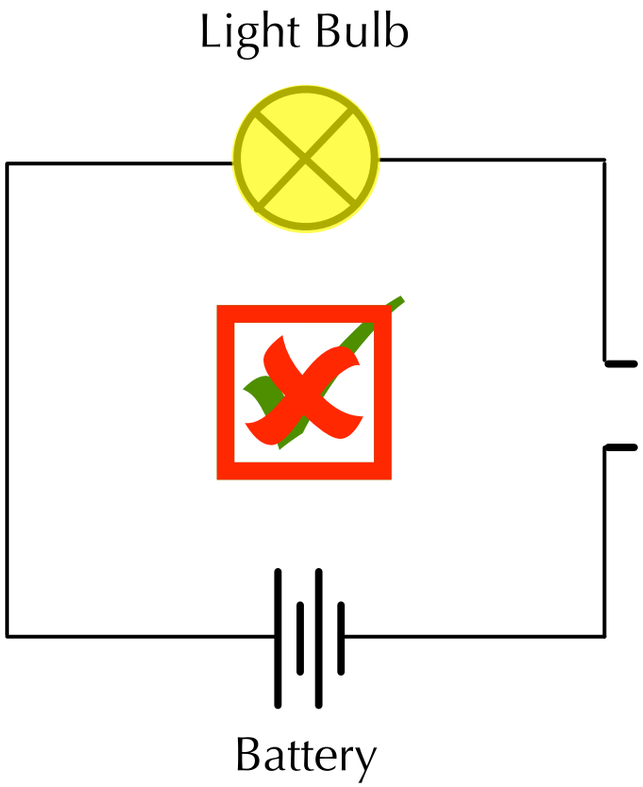

Tests the whole system pretending to be a user

Test collections of classes as subsystems

Test individual classes / methods in isolation

Google

# Testing a Flashlight



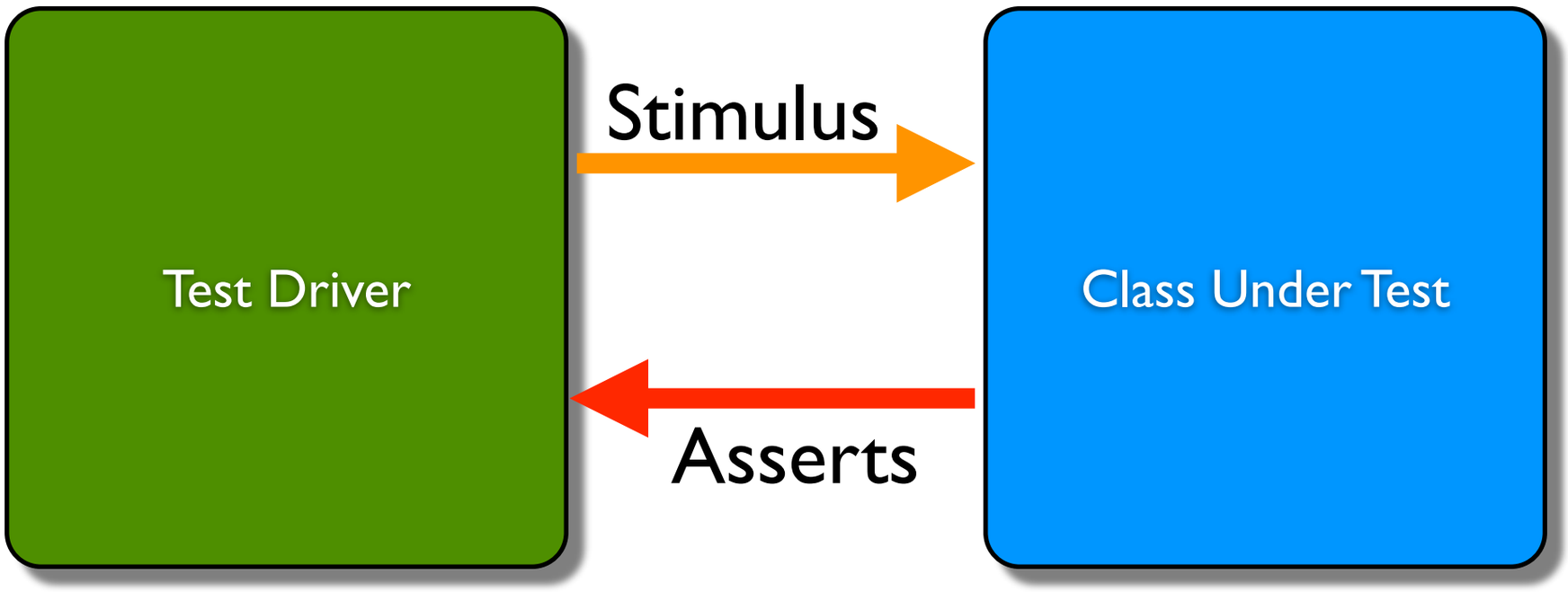

| Test | Probability at fault % | Cost To Run | Action At Fault |
|---|---|---|---|
| Flashlight | 100% | 50 | ??? |
| Light Bulb | 50% | 1 | Replace |
| Battery | 45% | 1 | Replace |
| Switch | 4% | 1 | Replace |
| Connectors | 1% | 2 | Replace |

Google

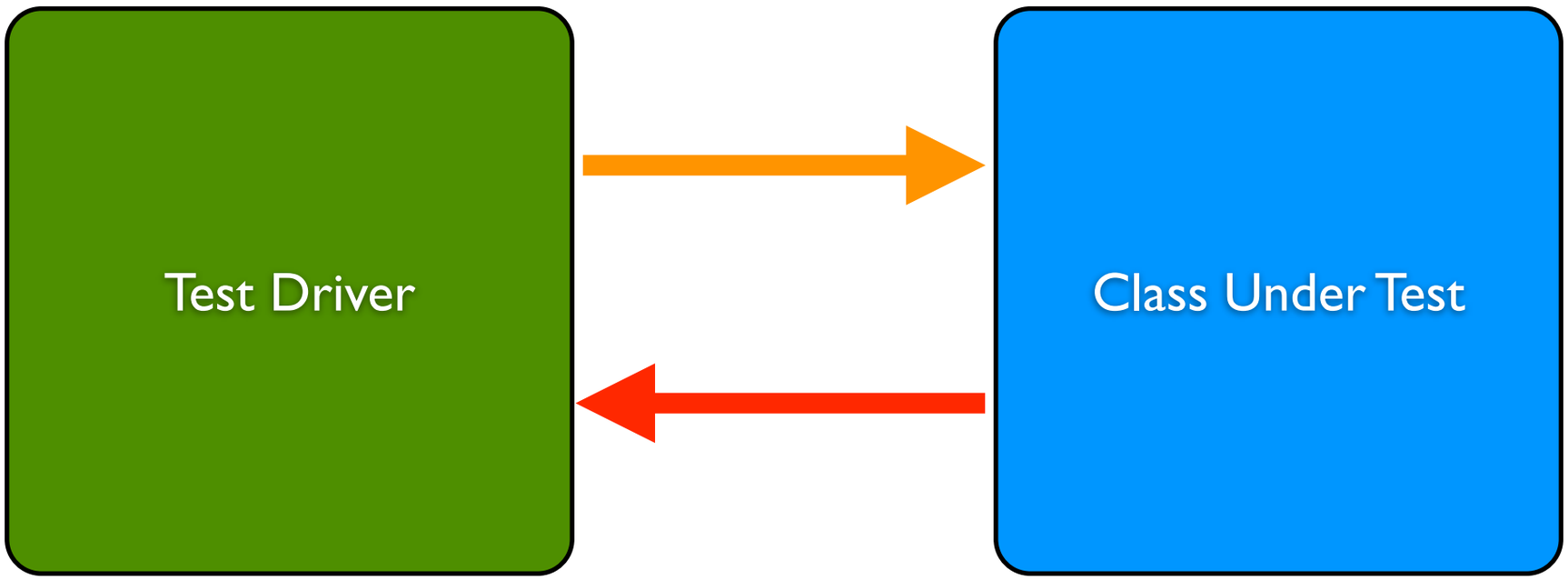
# Unit Tests >>> # Large Tests

Google

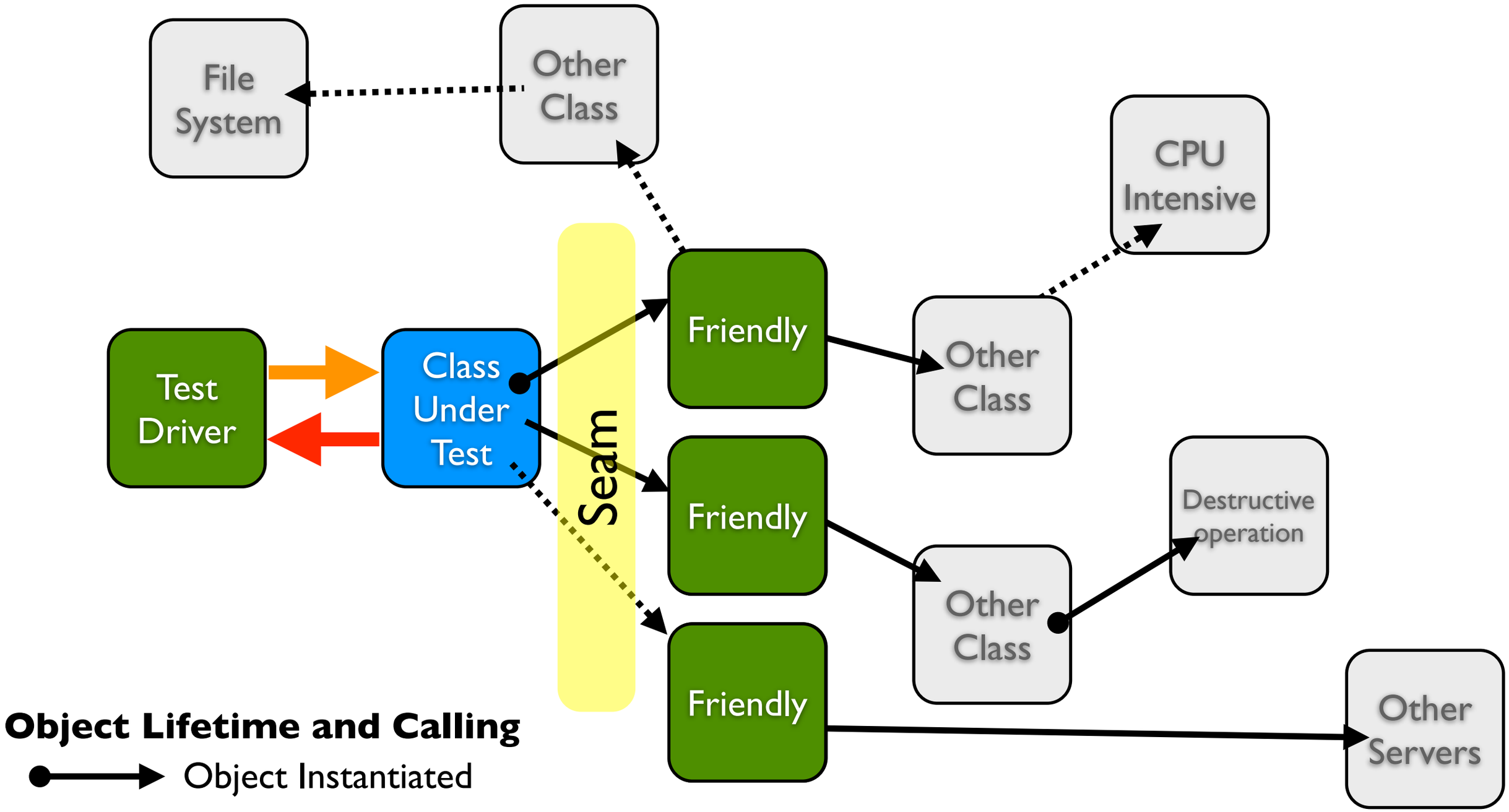
# Unit Testing a Class



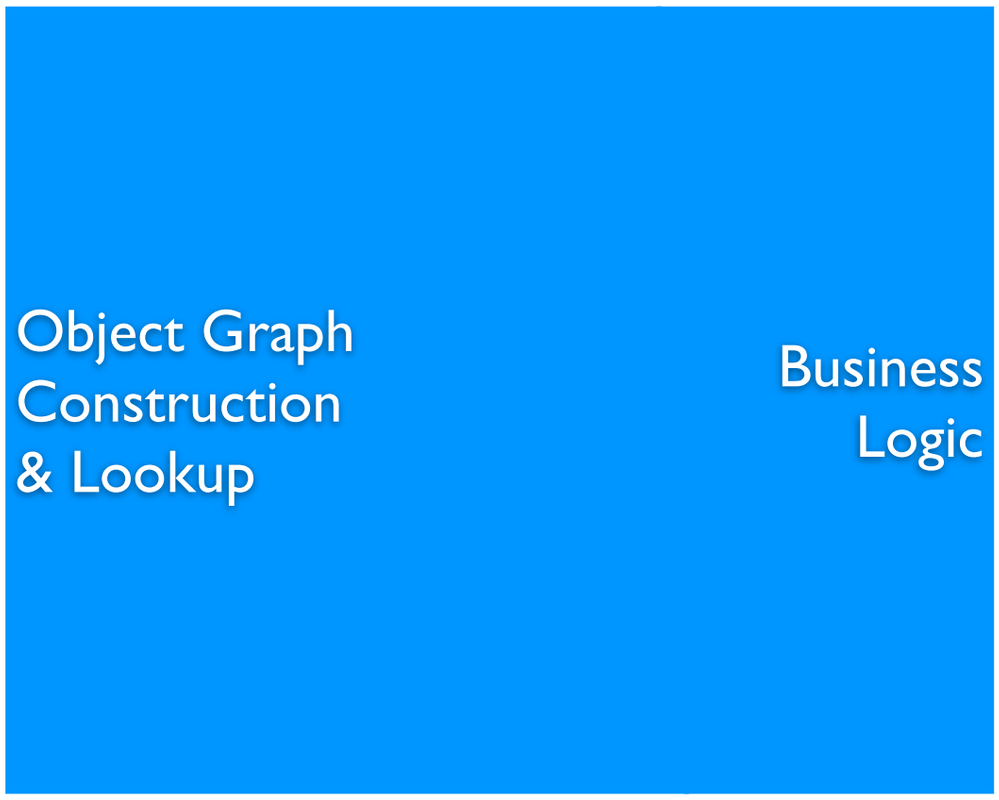
# Unit Testing a Class



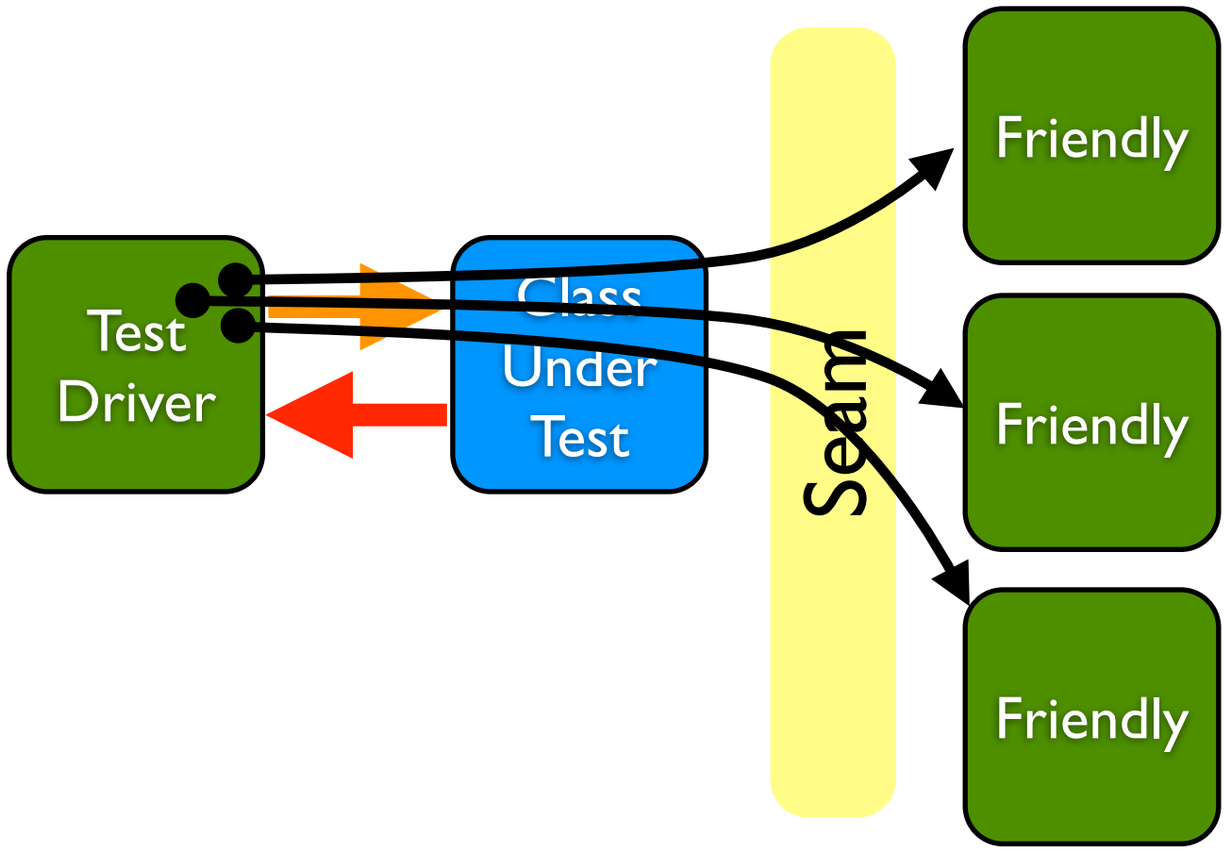
# Unit Testing a Class




Google

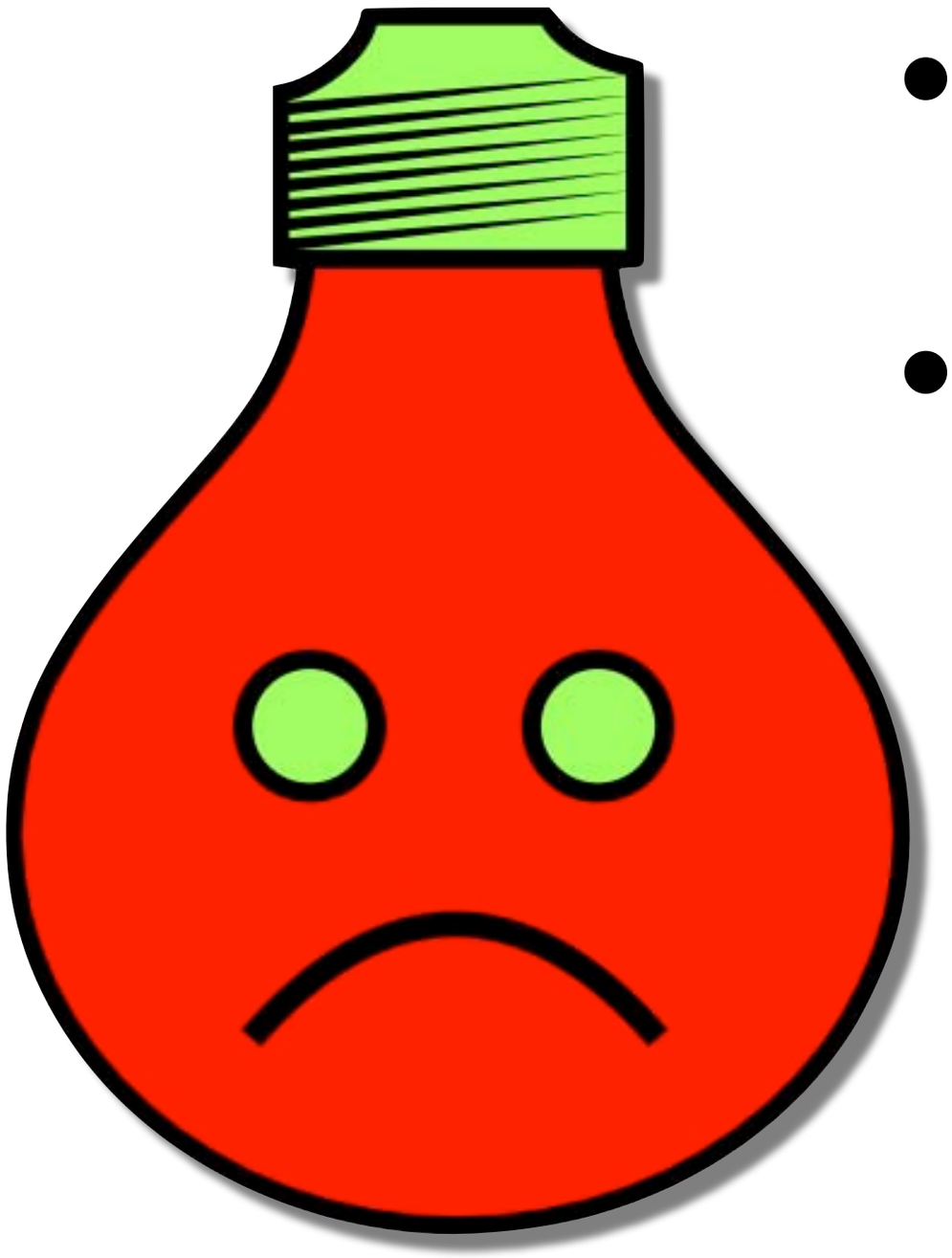


# Unit Testing a Class



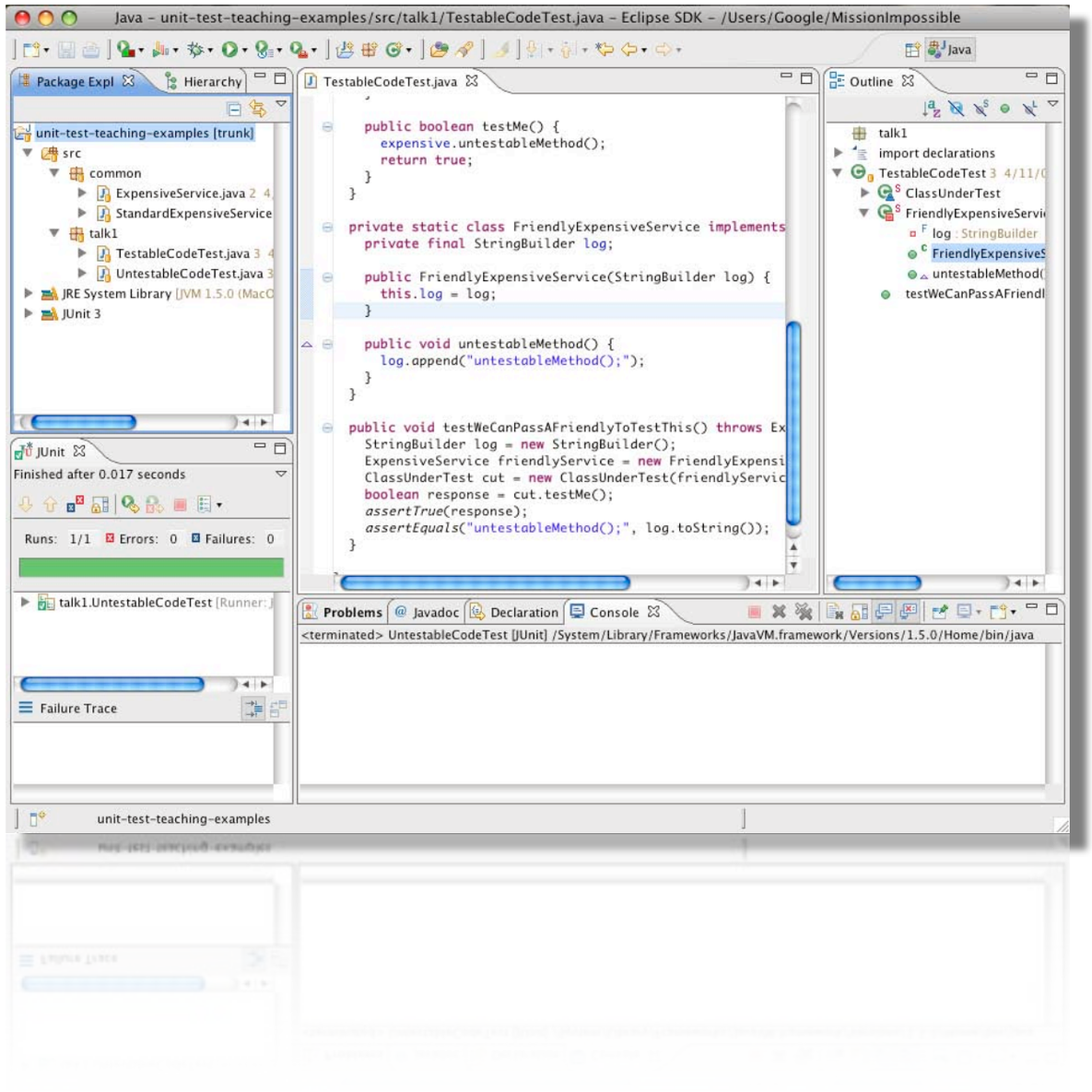

**Object Lifetime and Calling**

- Object Instantiated
- Object Passed In
- Global Object



- 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.

# Code Example




Google

# Take Away

- Unit Tests are a preferred way of testing
- Unit Tests require separation of Instantiation of Object Graph from Business Logic under test

Google

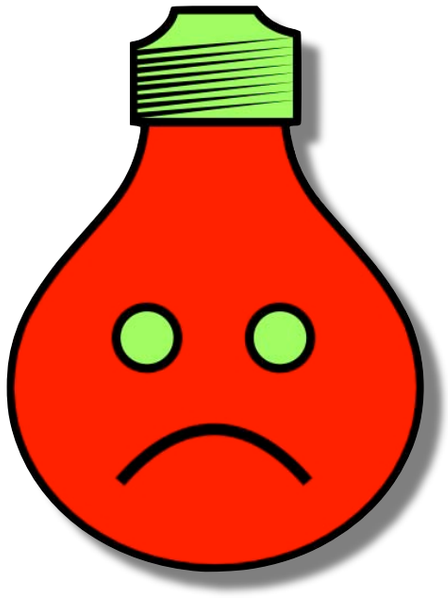
# Fight Mr. Untestabulb

- Trick #1: Ask for things
  - Avoid direct instantiation
  - Avoid work in constructor
  - Prefer passing in dependencies
- Trick #2: Avoid Global Mutable State
  - Bad Singleton vs Good Singleton
  - Deceptive API
- Trick #3: Avoid Deep Inheritance Hierarchies
  - Prefer Composition over Inheritance
  - Prefer Polymorphism over conditionals

Google

# Fighting Mr. Untestable Trick#1
# Ask for Things!
# Don't look for Things!

Cost of Constructor
Always ask for Things
Law of Demeter
Where have all the "`new`" operators gone?



by Miško Hevery

# 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);
   }
}
```

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);
  }
}
```

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);
  }
}
```

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

```
class Document {
  String html;

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

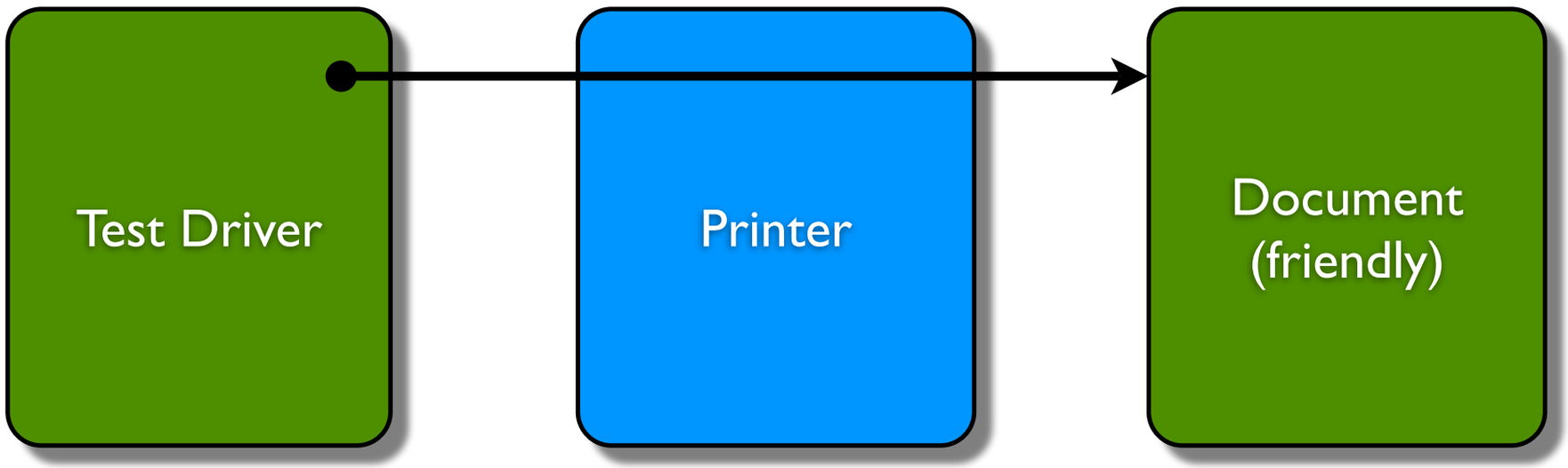
```
class Printer {

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

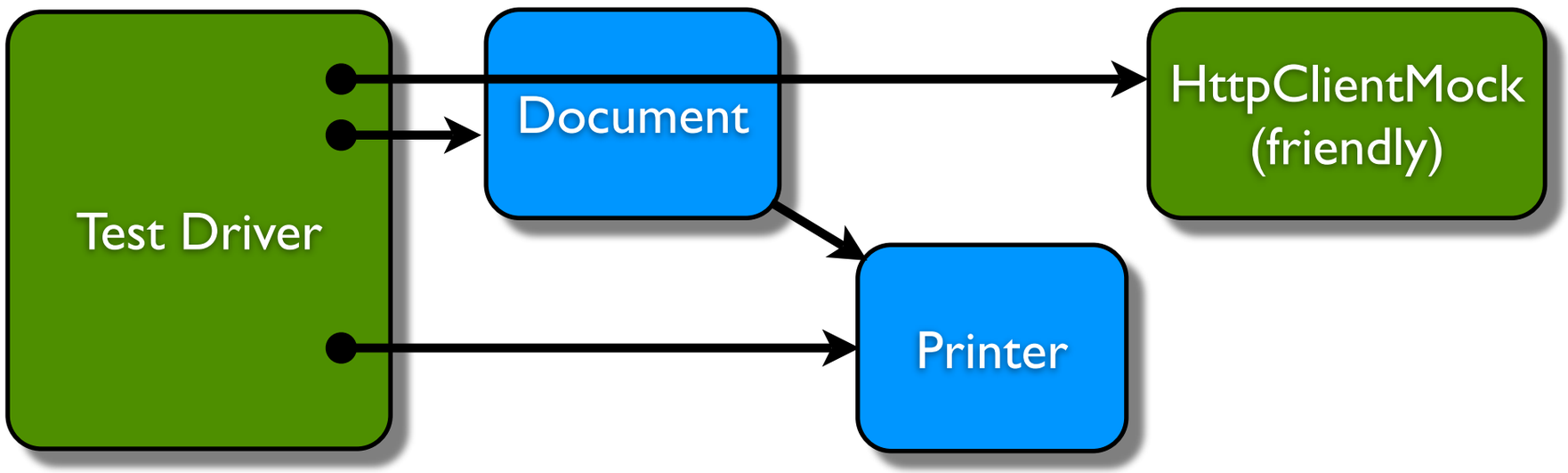
Easy to test
since Document is easy to
construct

```
class DocumentFactory {
  HtmlClient client;

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

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

Google

# Cost of Construction



# 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

# Service Locator

Google

# 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){
    }
}
```

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

**What needs to be mocked out in test?**

# 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);
  }
}
```

# Service Locator

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

# 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);
  }
}
```

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());
  }
}
```

Google

# 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());
  }
}
```

Google

# 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

# Myth about DI

- DI makes refactoring hard
  - If a child object needs a new parameter then I have to pass it through all of its parents
  - Because of Law of Demeter you should never ask for anything you don't directly need.
  - Parent object does not make a child it asks for child. So if child needs a new dependency the parent is not aware

# Myth about DI

```
class House {
  House(Door door) {...}
}

class Door {
  Door(DoorKnob knob){...}
}

class DoorKnob {
  DoorKnob() {...}
}
```

# Myth about DI

```
class HouseFactory {
  House build(){
    DoorKnob knob = new DoorKnob();
    Door door = new Door(knob);
    return new House(door);
  }
}
```

# Myth about DI

```
class House {
  House(Door door) {...}
}

class Door {
  Door(DoorKnob knob, Window window){...}
}

class DoorKnob {
  DoorKnob(Color color) {...}
}
```

# Myth about DI

```
class HouseFactory {
  House build(){
    Color color = Color.RED;
    DoorKnob knob = new DoorKnob(color);
    Window window = new SmallWindow();
    Door door = new Door(knob, window);
    return new House(door);
  }
}
```

# Object Lifetime

- Constructor
  - Only objects whose lifetime is equal to or greater
- Method Parameters
  - Objects whose lifetime is shorter

# Test vs. Production

| | Production | Tests |
|---|---|---|
| `null` | ✗ | ✓ |
| `new` | ✗ | ✓ |

# Myth about DI

```
class House {
  House(Door door) {
    checkNotNull(door);
    this.door = door;
  }
  void paint(Color color){...}
}
```

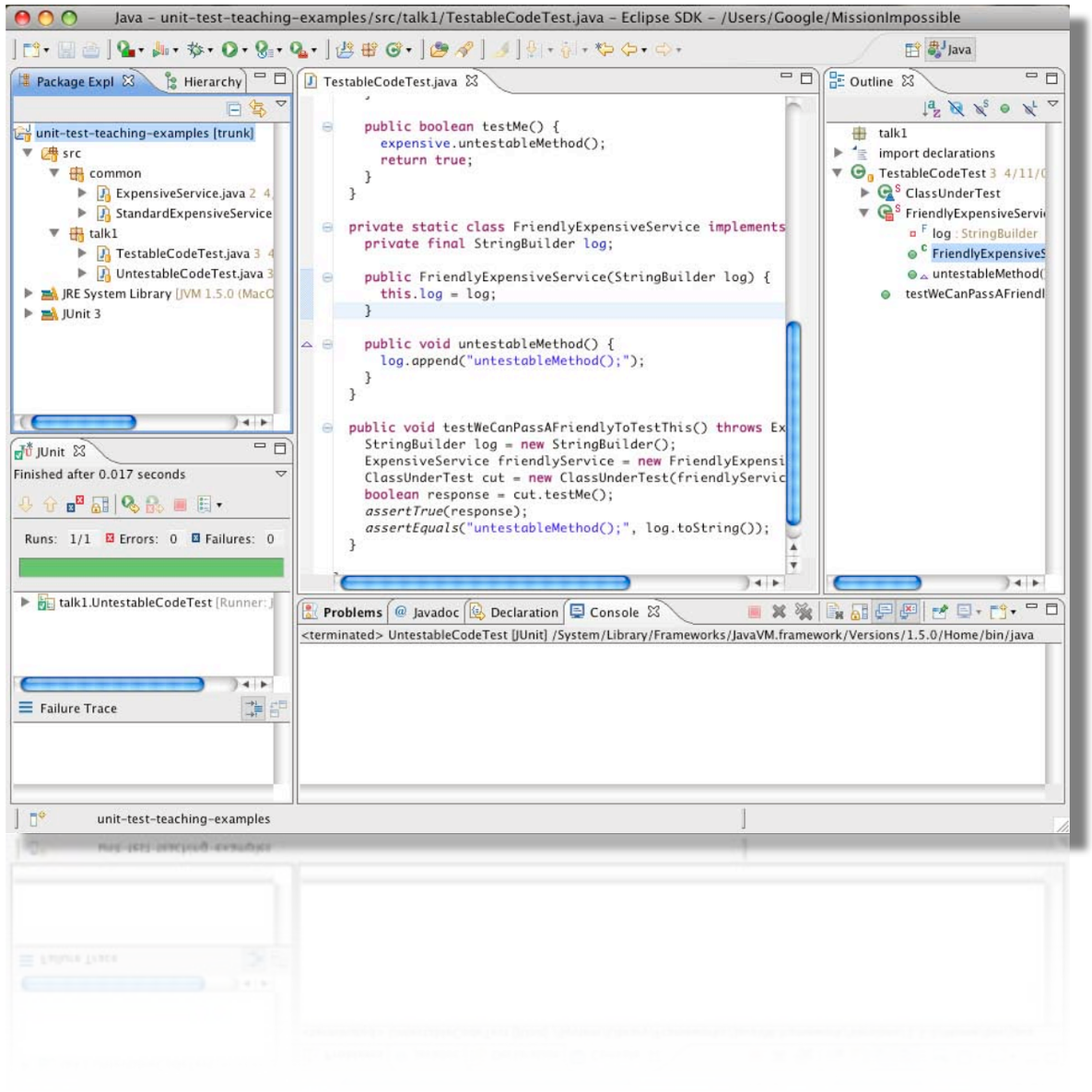
Passing a `null` in test says "This is of no interest" And it is great way to write clean tests. Preconditions make it harder to write tests.

A passing test asserts a lot more about the quality of a code than a precondition can. Getting rid of preconditions in favor of tests is a worthwhile trade.

```
testHousePainting(){
  House house = new House(null);
  house.paint(Color.RED);
  assertEquals(Color.RED, house.getColor())
}
```

Google

# Code Example




Google

# Always Ask for Things

- abandon the `new` operator
  - All of the `new` operators end up with application configuration object (which either works or not)
- Since you are never constructing anything there is no need to know about objects you don't directly use

# Myth about DI

```
class HouseFactory {
  House build(){
    Color color = Color.RED;
    DoorKnob knob = new DoorKnob(color);
    Window window = new SmallWindow();
    Door door = new Door(knob, window);
    return new House(door);
  }
}
```

- Everything works
  - GREAT
- Wiring is wrong
  - Catastrophic failure
- System fails in some small way
  - Bug with specific class, missing unit test.

Scenario tests are needed to ensure that objects are wired together properly.

Google

# Fight Mr. Untestable Trick#2
# Avoid Global Mutable State

Good Singleton vs. Bad Singleton
Encapsulation vs Deceptive API
Order of tests should not matter

by Miško Hevery

# Global State

insanity |inˈsanitē|

noun

*repeating the same thing and expecting a different result.*

Google

# Global State

```
class X {

  ... come code / fields ...

  X() { ... }

  public int doSomething(){ ... }
}

int a = new X().doSomething();
int b = new X().doSomething();
```

**a == b** **?** **a != b**

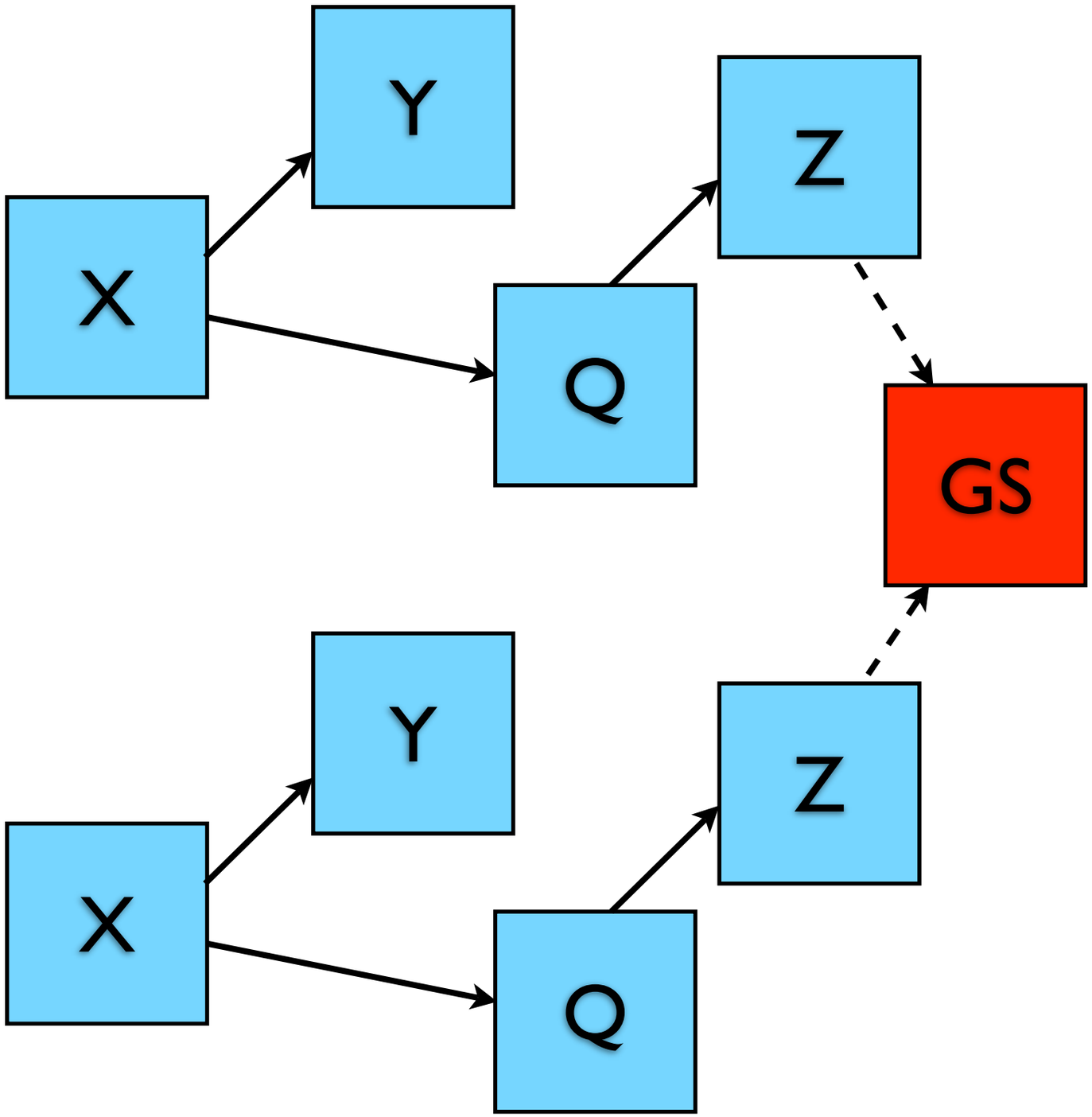
Google

# Global State

- Object state is **transient** and subject to garbage collection
- Class state is **persistent** to lifetime of JVM

# Global State

```
a = new X() -->
a.doSomething()
```






```
b = new X() -->
b.doSomething()
```




Google™

# Global State

- Multiple Executions can produce different results
  - Flakiness
  - Order of tests matters
  - Can not run test in parallel
- Unbounded location of state

# Global State

- Hidden Global State in JVM
  - `System.currentTime()`
  - `new Date()`
  - `Math.random()`
- **Testing above code is hard**

# Singleton: Good vs Bad

- Application Global vs. JVM Global
- Usually we have One application per JVM
  - Hence we incorrectly assume that:
    Application Global == JVM Global
- Each test is a different configuration of a portion of our application
  - For tests it is very important that
  - Application Global != JVM Global

# Singleton: Good vs Bad

```
class AppSettings {
  private static AppSettings instance
    = new AppSettings();

  Object state1;
  Object state2;
  Object state3;

  private AppSettings() { ... }

  public static AppSettings getInstance(){
    return instance;
  }
}
```

Test can never access internal state of the singleton.

Internal state becomes globally accessible. Global state is transitive. "Static cling"

Google

# Singleton: Good vs Bad

```
class Class {
  int method() {
    return AppSettings.getInstance().doX();
  }
}
```

```
void testClass() {
  ????
}
```

# Singleton: Good vs Bad

```
class AppSettings {
  private Object state1;
  private Object state2;
  private Object state3;

  public AppSettings() { ... }
}
```

- Class no longer enforces its own singletonsess
- Application code only creates one.

# Singleton: Good vs Bad

```
class Class {
  AppSettings settings;
  Class(AppSettings settings){
    this.settings = settings;
  }
  int method() {
    return settings.doX();
  }
}

void testClass(){
  new Class(new AppSettings(...)).method();
}
```

Google

# Deceptive API

- API that lies about what it needs
- 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);
}
```

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

# Deceptive API

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

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

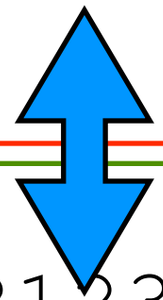
# Deceptive API

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

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

# Deceptive API

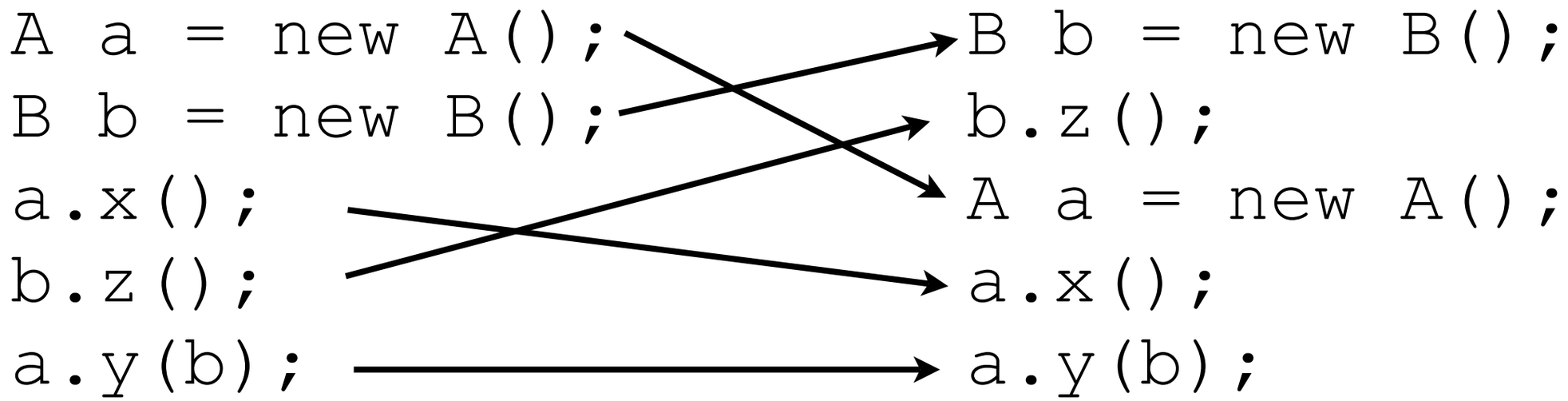
```
testCharge() {
  Database.connect(...);
  OfflineQueue.start();
  CreditCardProcessor.init(...);
  CreditCard cc;
  cc = new CreditCard("1234567890121234");
  cc.charge(100);
}
```



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

Google

# Deceptive API




- Code should have a commutative property
- The above code will fail if there is Global State!!!
- Dependency injection orders code naturally!!!

Google

# Deceptive API

```
testCharge() {
  CreditCardProcessor.init(...);
  OfflineQueue.start();
  Database.connect(...);

  CreditCard cc;
  cc = new CreditCard("12                234");
  cc.charge(100);
}
```

Singletons do not enforce the order of initialization. Often time global state initialization is the messiest code in your application. In tests you just need to know what the correct order is and you need to clean up after yourself.

# 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.

# Review

- Global state is the root of all test problems
- Global state can not be controlled by tests
- Singleton is a common form of encapsulating global state
- Only the Singletons which enforce their own Singleton-ness are a problem
  - “static instance” has transitive property of globallity

# Fighting Mr. Untestable Trick#3: Avoid deep inheritance hierarchies with conditionals

Favor Polymorphism over Conditionals
Favor Composition over Inheritance
When is Inheritance Appropriate

by Miško Hevery

# Polymorphism vs Conditionals

## Premise

Most ifs can be replaced by polymorphism (subclassing)

This is desirable because:

- Functions without ifs are easier to read than with ifs
- Functions without ifs are easier to test
- Related branches of ifs end up in the same subclass

# Polymorphism vs Conditionals

- Subclass => Polymorphism
    - If you are checking an object should behave differently based on its state
    - If you have to check the same if condition in multiple places

- Use if
    - bounds checking  primitive objects (>,<, ==, !=)

- To make your code IF free:
    - Never return a null
        - Null behavior object
        - Empty list
        - Throw on errors don't return error codes.

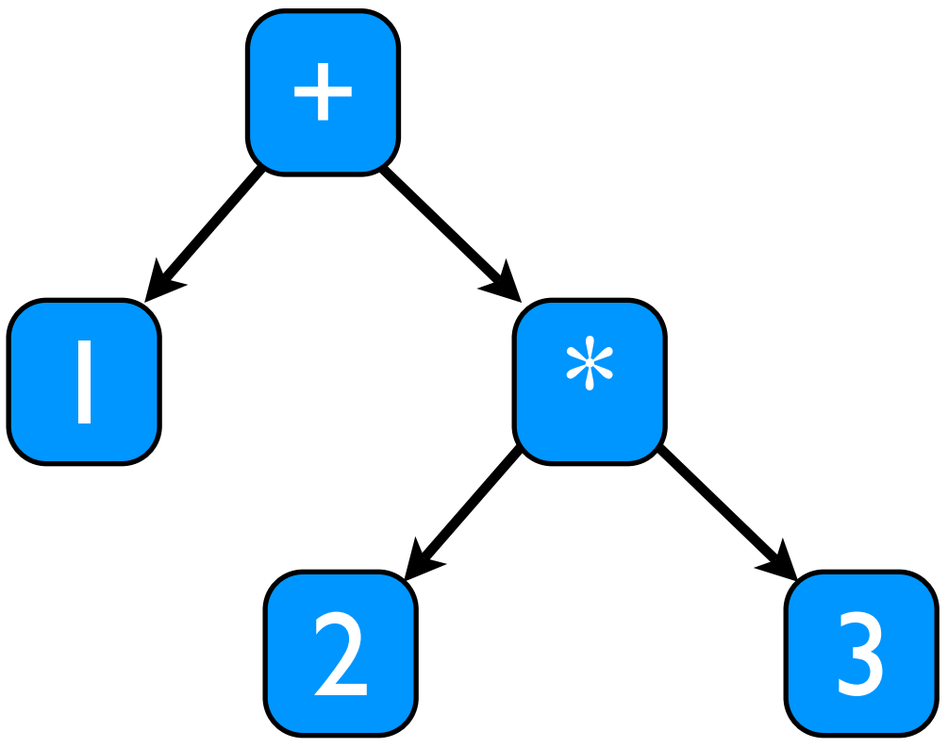
Google

# When the object behaves differently based on state

Google

# Example

1 + 2 * 3



# Assignment

- define `evaluate()` method which computes the result of an expression.

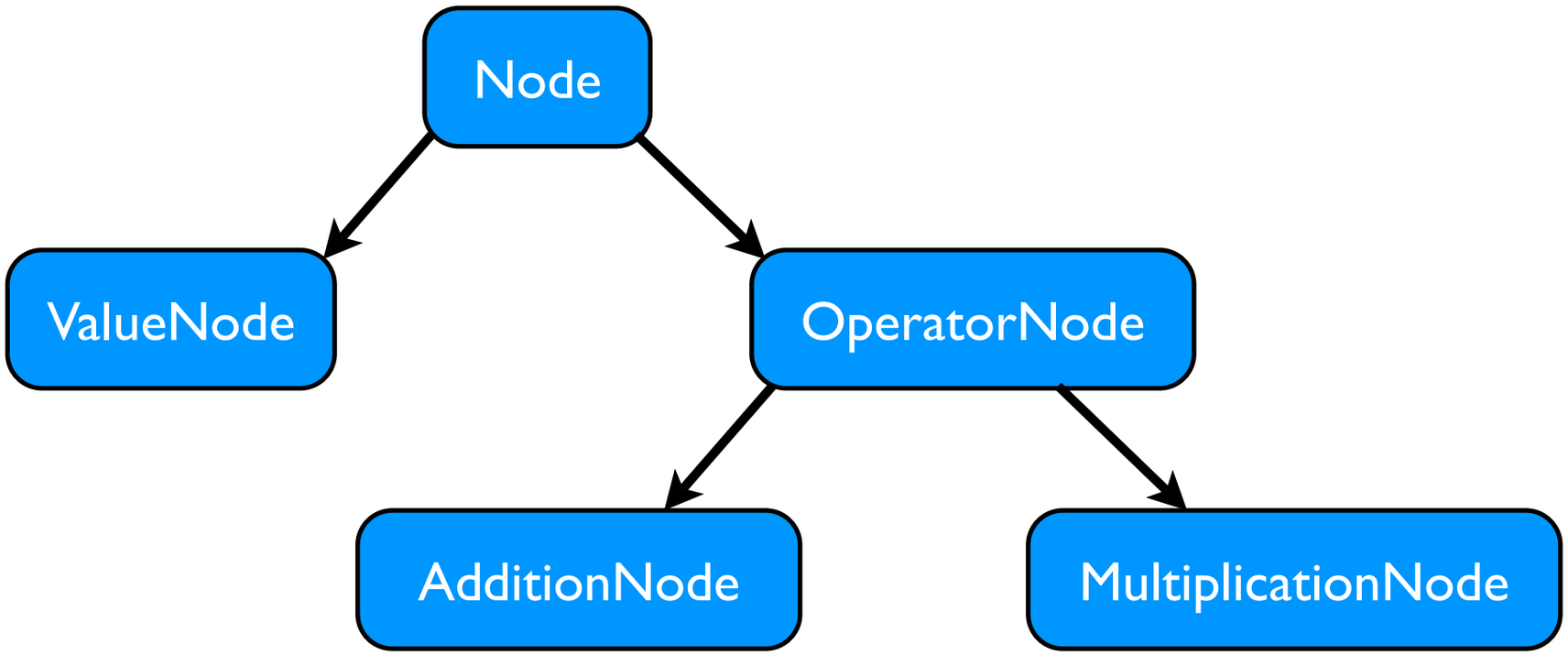
Google

# Example

```
class Node {
  char operator;
  double value;
  Node left;
  Node right;

  double evaluate(){
    switch (operator) {
      case ‘#’: return value;
      case ‘+’: return left.evaluate() + right.evaluate();
      case ‘*’: return left.evaluate() * right.evaluate();
      case ...
    }
  }
}
```

| | # | + | * |
|---|---|---|---|
| operator (behavior) | ✓ | ✓ | ✓ |
| value | ✓ | | |
| left | | ✓ | ✓ |
| right | | ✓ | ✓ |

# Class Hierarchy



# Example

```
interface Node {
  double evaluate();
}

class ValueNode implements Node {
  double value;
  double evaluate(){
    return value;
  }
}

class AdditionNode implements Node {
  Node left;
  Node right;
  double evaluate(){
    return left.evaluate() + right.evaluate();
  }
}
```

Google

# Assignment

- define `toString()` prints the infix expression placing parenthesis only when necessary.

# Assignment

- Add new math operators:
  - power: ^
  - factorial: !
  - logarithm
  - trigonometry

# When to use if and when to Subclass

- Polymorphic solution is better because
  - New behavior can be added without having the original source code
  - Each operation / concern is separated in a separate file
  - Makes it easy to test / understand

- ***Prefer* polymorphism over conditionals**
  - Switch screams polymorphism
  - `IF`s are more subtle

Google™

# If you check same condition in multiple places

# Getting rid of the IF

```
Class Update {
  execute() {
    if (FLAG_i18n_ENABLED) {
      // DO A;
    } else {
      // DO B;
    }
  }
  render() {
    if (FLAG_i18n_ENABLED) {
      // render A;
    } else {
      // render B;
    }
  }
}
```

```
public void testExecuteDoA {
    FLAG_i18n_ENABLED = true;
    Update u = new Update()
    u.execute();
    assertXXX();
}
```

```
public void testExecuteDoB {
    FLAG_i18n_ENABLED = false;
    Update u = new Update()
    u.execute();
    assertXXX();
}
```

# Getting rid of the IF

```
Class Update {
}

Class I18NUpdate
     extends Update {
  execute() {
    // DO A;
  }
  render() {
    // render A;
  }
}

Class MyNonI18NUpdate
     extends Update {
 execute() {
    // DO B;
  }
  render() {
    // render B;
  }
}
```

```
public void testExecuteDoA {
   Update u = new MyI18NUpdate()
   u.execute();
   assertXXX();
}

public void testExecuteDoB {
   Update u = new MyNonI18NUpdate()
   u.execute();
   assertXXX();
}
```

Google

# Getting rid of the IF

```
class Consumer {
  Consumer(Update u){...}
}

class Factory {
  Consumer build() {
    Update u = FLAG_i18n_ENABLED
                  ? new I18NUpdate()
                  : new NonI18NUpdate();
    return new Consumer(u);
  }
}

class UpdateProvider implements Provider<Update> {
  @Inject Provider<I18NUpdate> i18n;
  @Inject Provider<NonI18NUpdate> ni18n;
  @Inject boolean isI18N;
  Update get() {
    return isI18N ? i18n.get() : ni18n.get();
  }
}
```

Google

# When to use if and when to Subclass

- Polymorphic solution is better because
- The i18n code is separate from non i18n code.
  - It is easy to tell what the differences are between them

# Inheritance Abuse

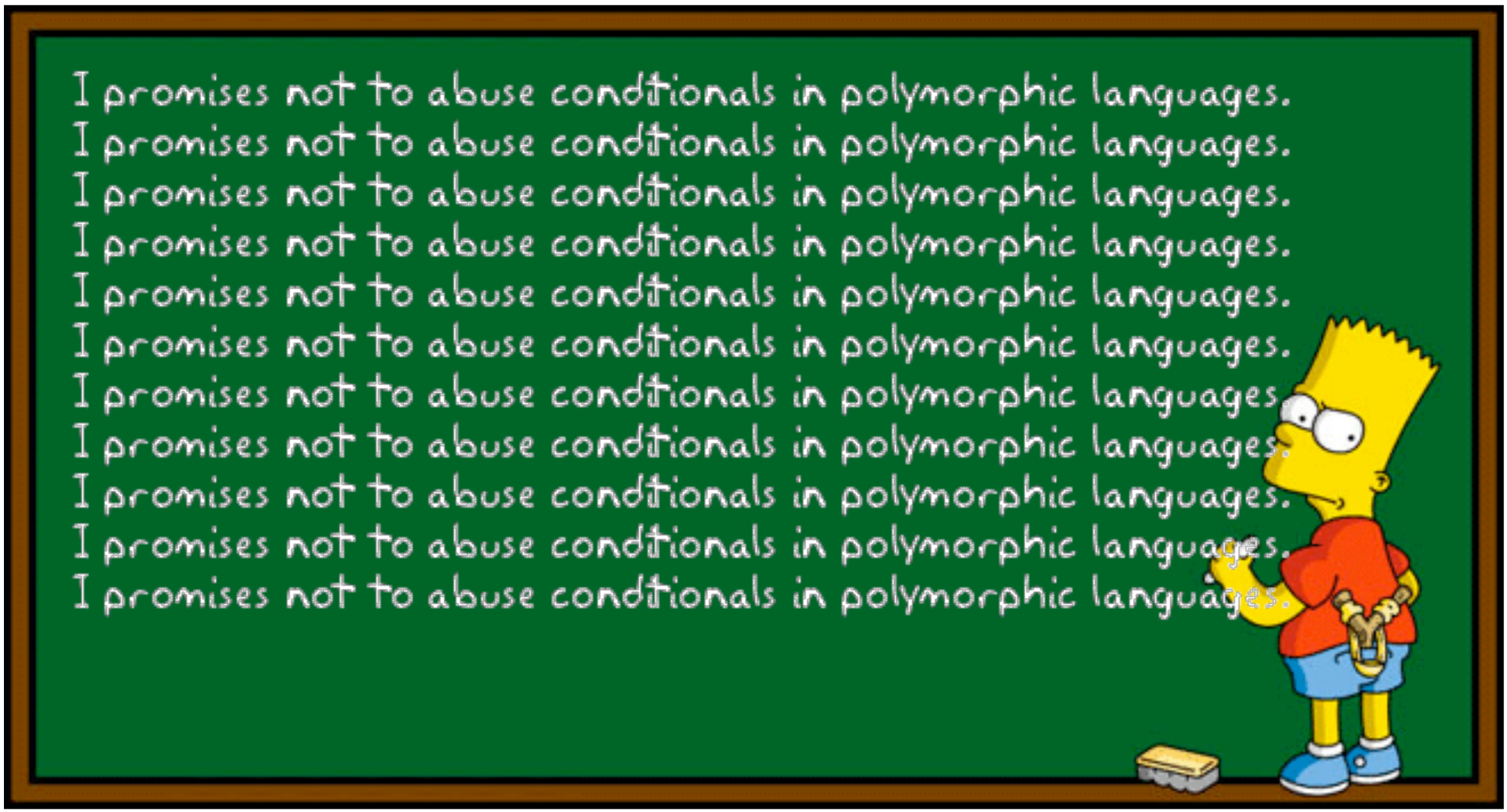
- Polymorphism is powerful and can replace if and make your code easier to understand

- Polymorphism can also make your code hard to understand

- In general we prefer object composition over inheritance, use polymorphism only if you have an actual problem to solve

- As with any new tool try not to use it everywhere, make sure you have a need before you put something like this in place

# Bart says...

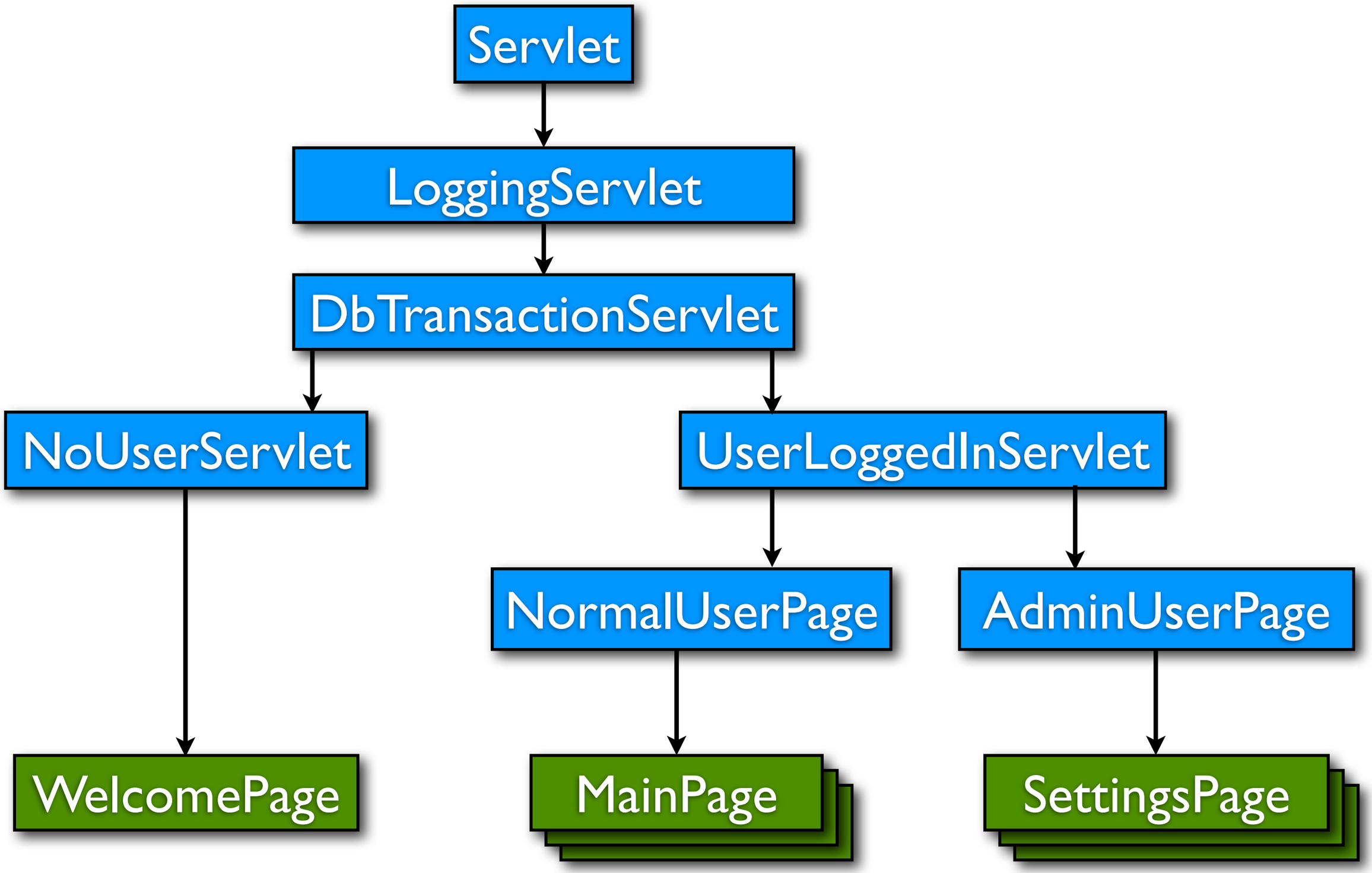



Google

# Composition vs Inheritance

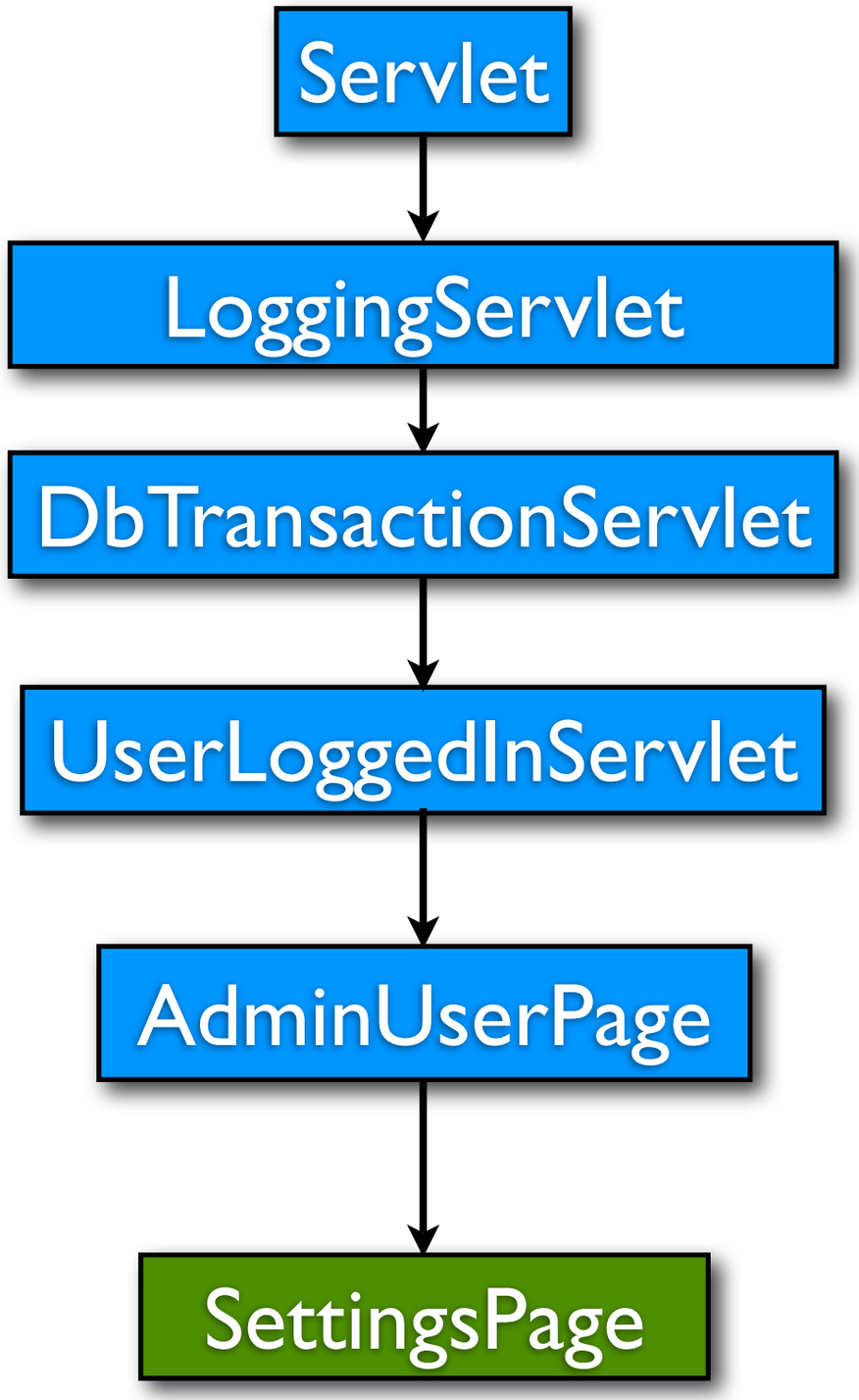
- The purpose of Inheritance is polymorphic behavior
- If you don't take advantage of polymorphism you should reuse code through delegation / composition

# 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!
  }
}
```



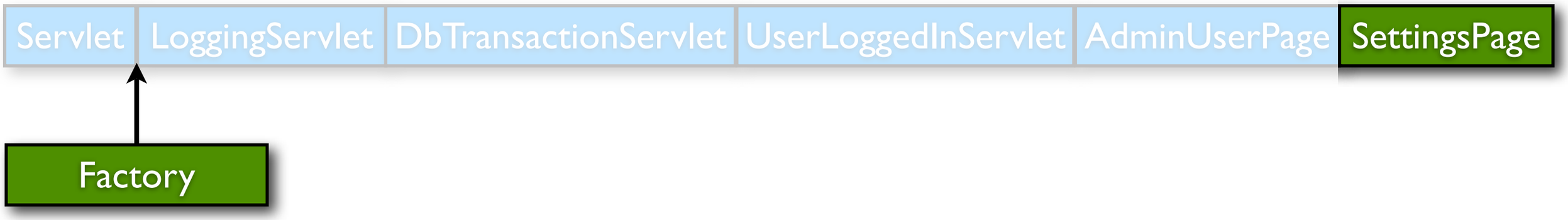

Google

# 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




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

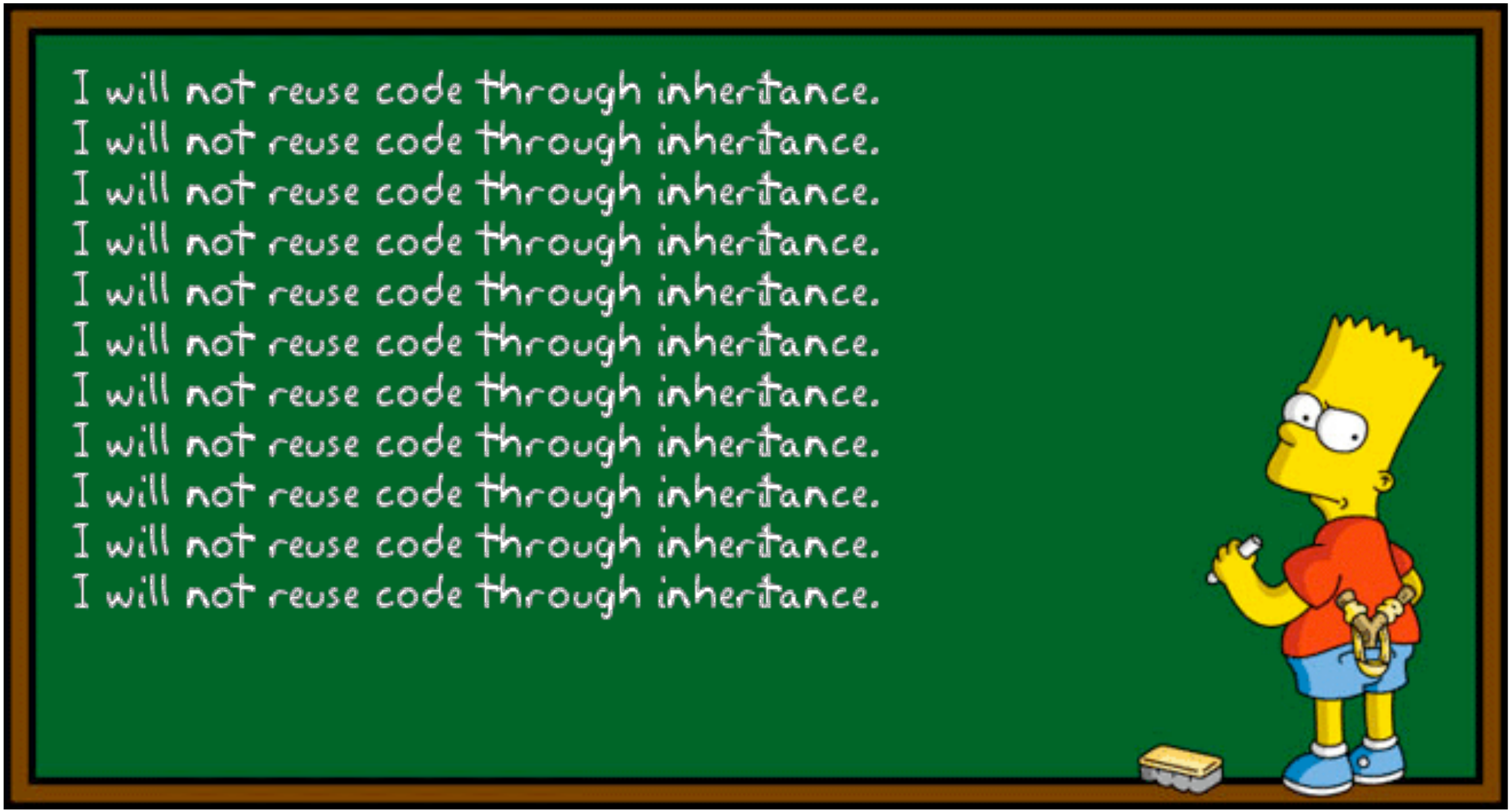
```
public void testAddUser() {
  UserRepository users = new InMemoryUserRepository();
  SettingsPage page = new SettingsPage(users);
  page.addUser(“jon”);
  assertNotNull(users.getUser(“jon”))
}
```

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

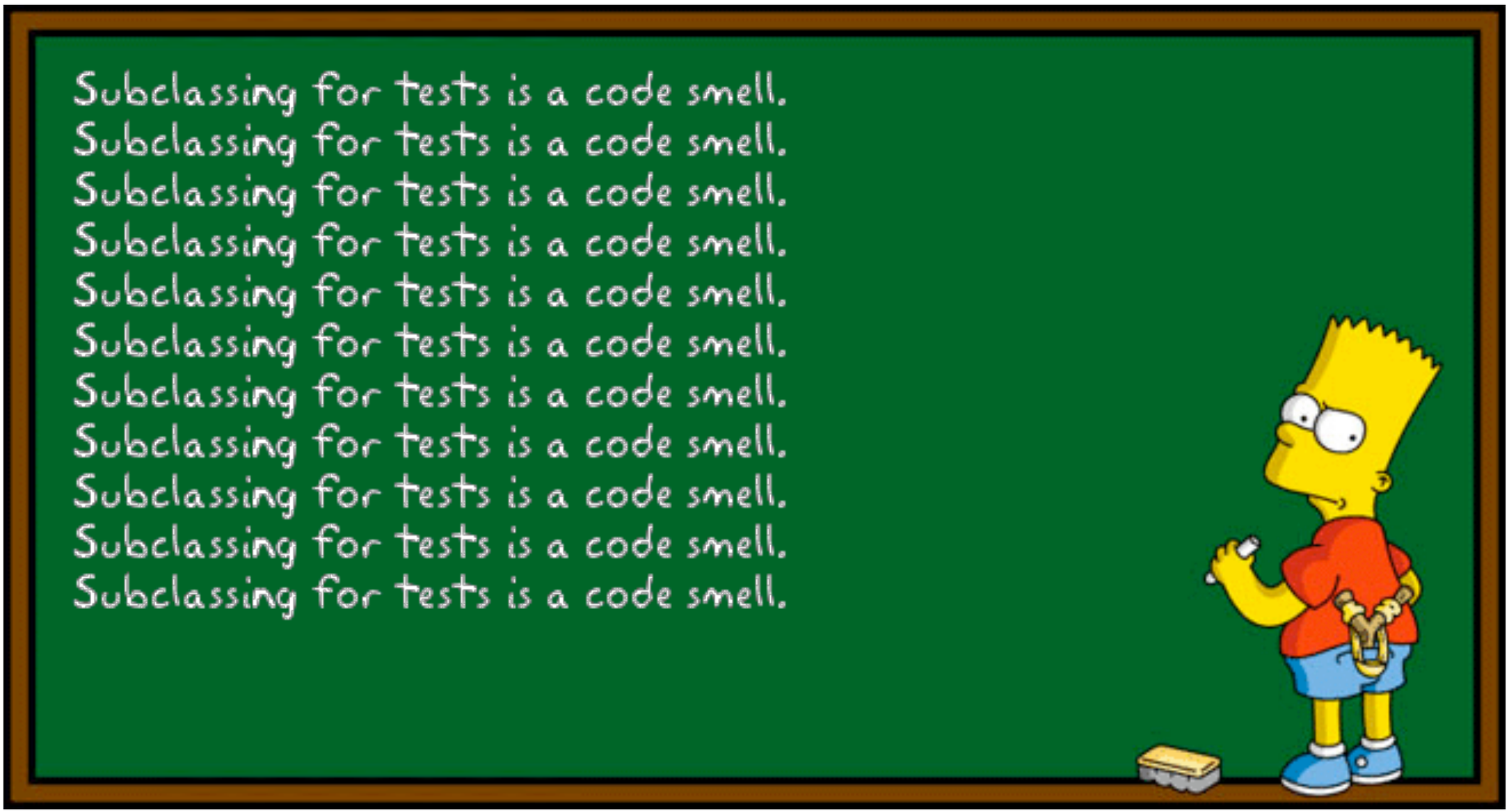
# Sub classing in tests

```
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”);
}
```

# 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




Google

# Summary

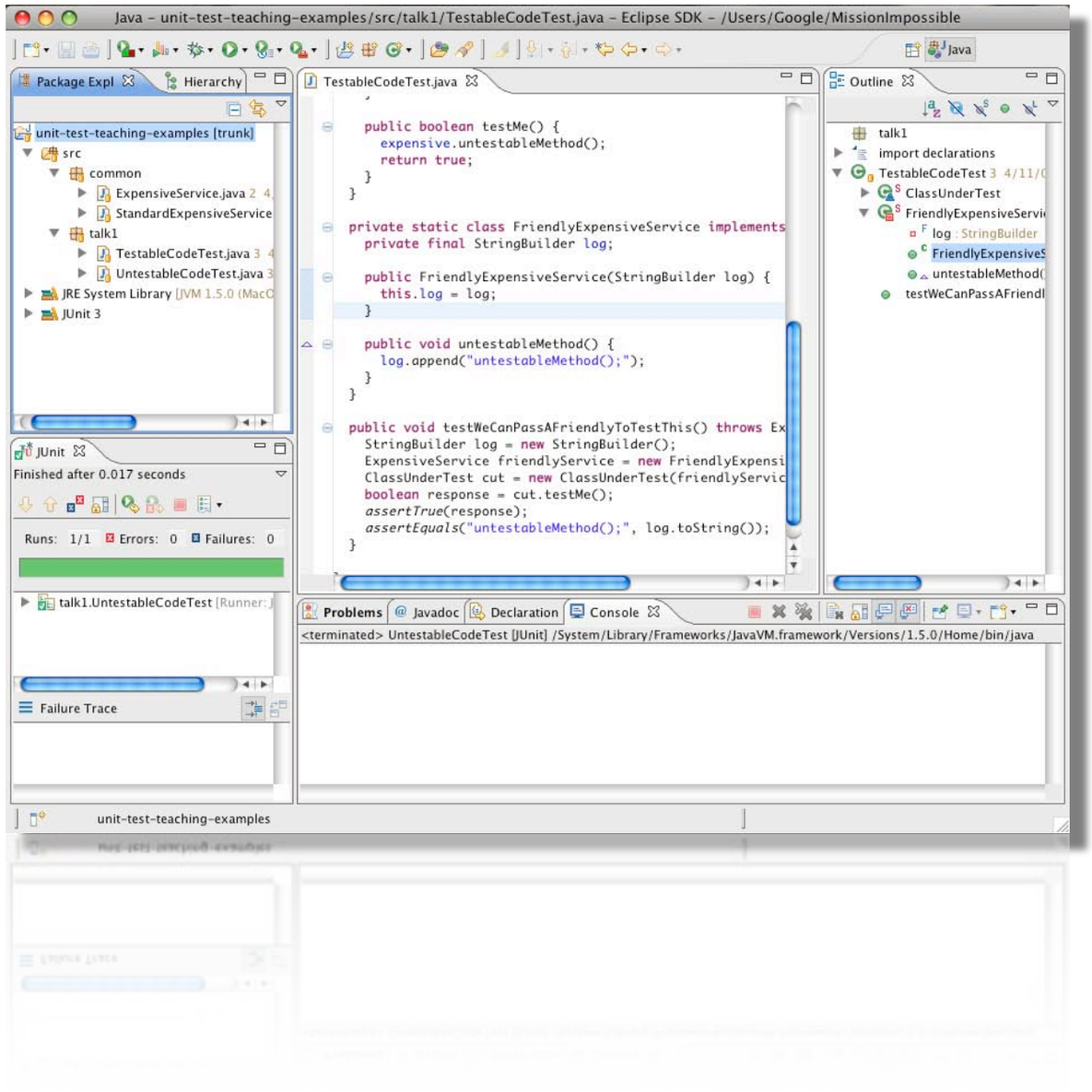
- Prefer polymorphism over conditionals
- Prefer composition over inheritance
- Subclassing in tests is code-smell

Google

# Test Driven Development

ping pong
programing by intention
Why TDD creates cleaner OO

# Code Example




Google

# Testing Services with Value Objects

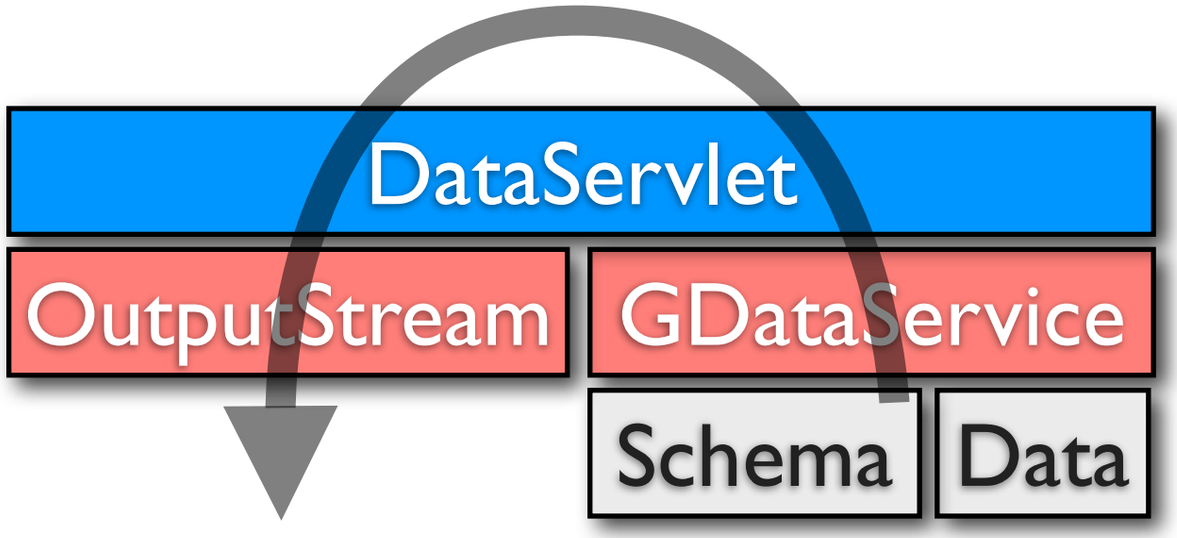
Service has interfaces
Value Objects are easy to Make
Breadth vs Depth
Making a Mockery

# Putting it all together

- Make objects easy to construct
- Always ask for things, don't construct / look for things
- Avoid global state
- Favor polymorphism over conditionals
- Favor composition over inheritance

- And your code can still be hard to test

# Building a Layer Cake



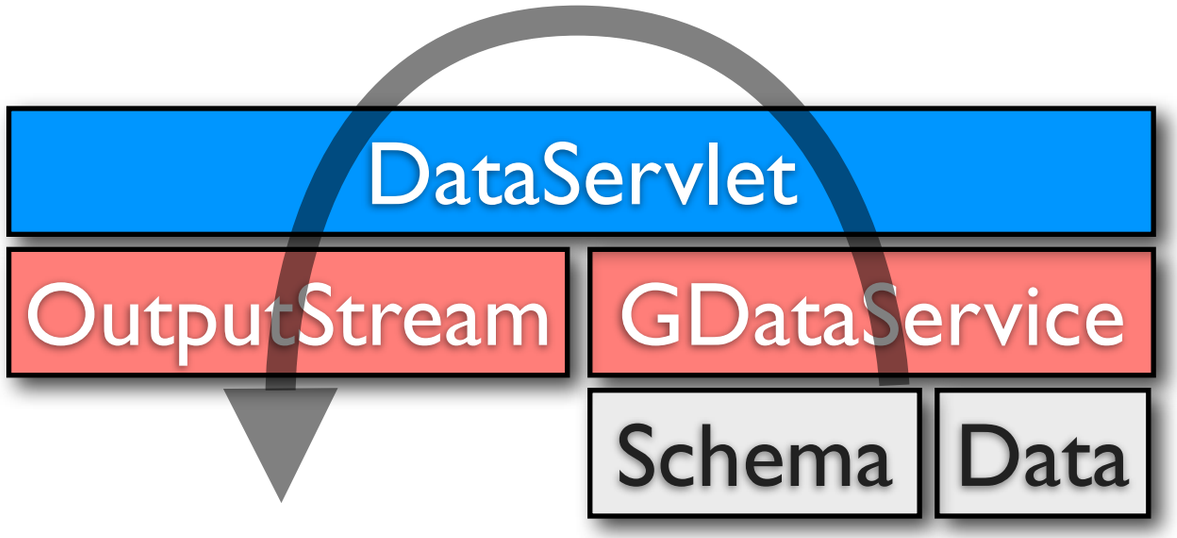

- Servlet reads data from GDataService and reformats it as XML output
- Reading particular data requires knowledge of where that column is (column locations are stored in schema)

# Building a Layer Cake

```
class DataServlet extends HttpServlet {
  public doGet(HttpRequest request, HttpResponse response) {
     data = new GDataReader(“http://...”);
     meta = new GDataReader(“http://...”);
     for each row in data
        for each row in meta
          find the row of interest in meta
        translate data into time series
        convert time series into XML
     write XML into response.getOutput()
  }

}
```

Google

# How would you test this?




- Servlets need no-arg constructors? How would you pass data in?
- How would you mock out GDataService?
  - *Even the simplest example requires many calls to GDataService*
- What would you assert on OutputStream?
  - *Even the simplest example produces large XML*

Google

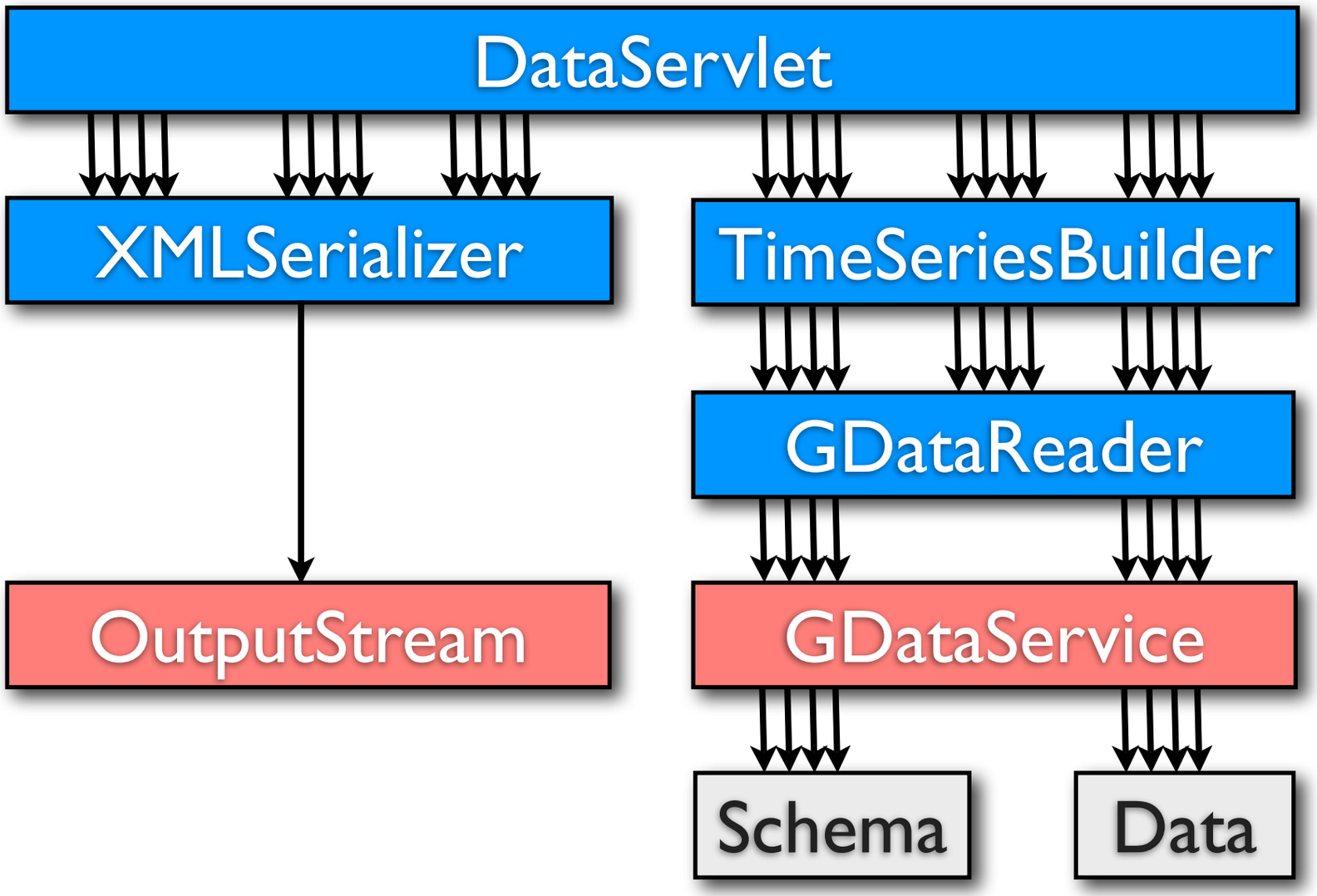
# Building a Layer Cake

```
class DataServletTest extends TestCase {
  testServlet() {
    Mockery mockery = new Mockery();
    GDataService gData = mockery.mock(GDataService.class);
    mockery.checking(new Expectations(){{
      one(gData).get(0, 1); will(returnValue(new Cell(....));
      one(gData).get(0, 2); will(returnValue(new Cell(....));
      one(gData).get(2, 1); will(returnValue(new Cell(....));
      .....
      one(gData).get(20, 19); will(returnValue(new Cell(....));
    }});
    DataServlet ds = new DataServlet(?gData?);
    ds.doGet(?request?, ?response?);
    mockery.assertIsSatisfied();
    ?assert output against gold file?
  }
}
```

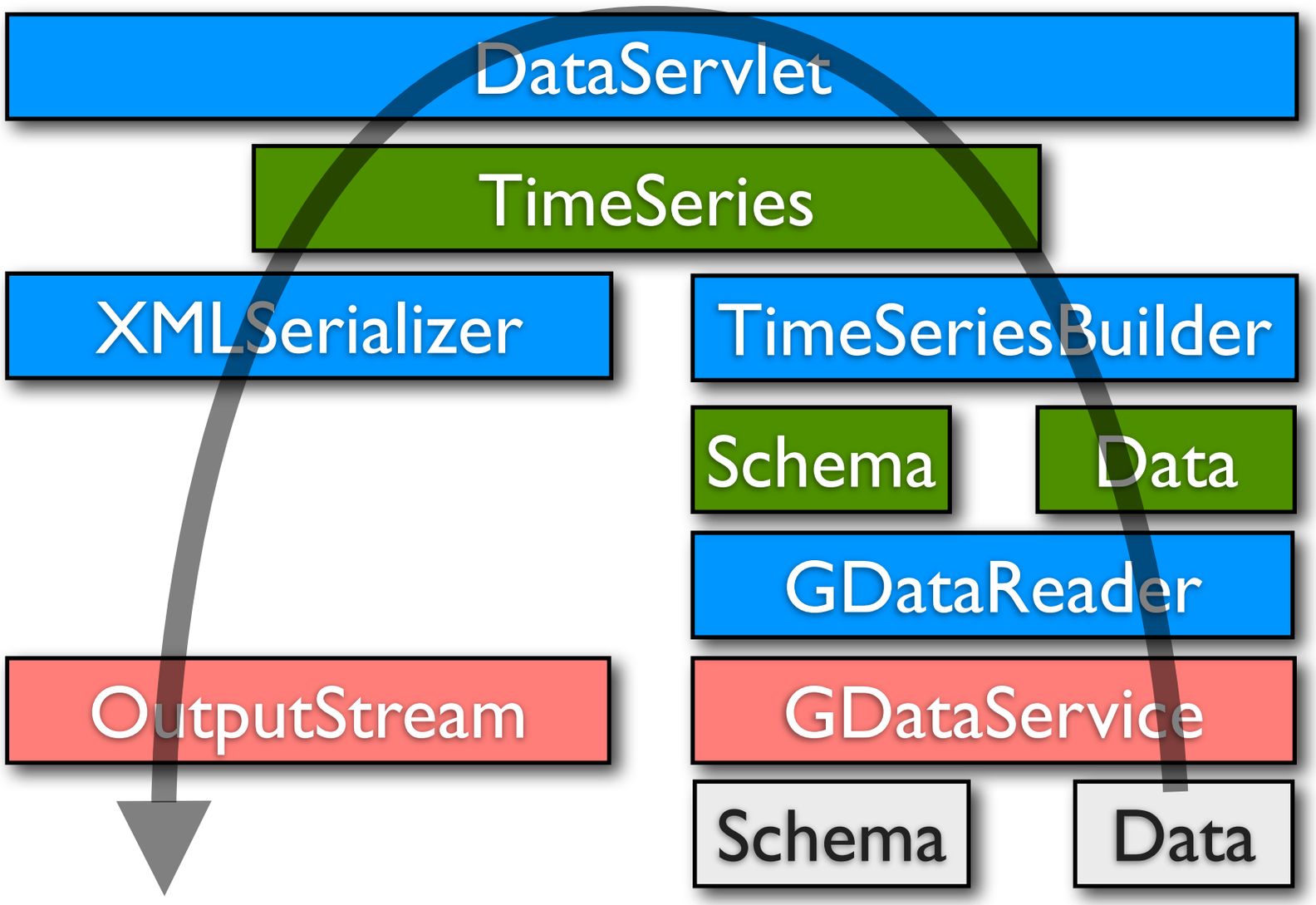
Google

- What will happen to your tests if the format changes?

# How would you test this?



# How would you test this?



# Building a Layer Cake

```
class DataServlet extends HttpServlet {
  public class DataServlet(
     XMLSerializer serializer,
     TimeSeriesBuilder timeSeriesBuilder,
     GDataReader gDataReader){...}

  public class DataServlet() {
    this(new XMLSerializer(...),
         new TimeSeriesBuilder(),
         new GDataReader(new GDataService("http...")
    )
  }

  public doGet(HttpRequest request, HttpResponse response) {
    Sheet data = gDataReader.read(request.get(...));
    Sheet meta = gDataReader.read(request.get(...));
    TimeSeries series = timeSeriesBuilder.build(meta, data);
    serializer.write(response.getOutputStream(), series);
  }
}
```

Google

# Value Object vs Service

**Value Object**

- Easy to Construct
- State Only
- Never Mock it
- Probably no Interface

**Service Object**

- DI heavy Constructs
- All about collaboration
- Mock candidate
- Probably has Interface
- Standard / InMemory

| | |
|---|---|
| `User, Address, MailMessage, PhoneNumber, CreditCard, ID, Location,` | `DBConnection, CreditCardProcessor, EventLogger, Repository, MailServer, LDAP, Authenticator` |

# Value Object vs Service

- Value Object should not ***refer*** to any services as constructor args
- Prefer immutable Value Objects
- Services often produce Value Objects
- Services are always DI, never created
- Value Objects may be DI or "`new`"

# JAVA:
# Introduction to GUICE

Modules
Providers
Scopes
Object Lifetime Management

# JAVA:
# JMock & EasyMock

# Credits

- All Images: (C) Creative Commons

Google