Guide: Writing Testable Code

To keep our code at Google in the best possible shape we provided our software engineers with these constant reminders. Now, we are happy to share them with the world.

Many thanks to these folks for inspiration and hours of hard work getting this guide done:

- Jonathan Wolter
- Russ Ruffer
- Miško Hevery

Available online at: http://misko.hevery.com/code-reviewers-guide/

Contents

Flaw: Constructor does Real Work

Work in the constructor such as: creating/initializing collaborators, communicating with other services, and logic to set up its own state *removes seams needed for testing*, forcing subclasses/mocks to inherit unwanted behavior. Too much work in the constructor prevents instantiation or altering collaborators in the test.

Warning Signs:
• new keyword in a constructor or at field declaration • Static method calls in a constructor or at field declaration • Anything more than field assignment in constructors • Object not fully initialized after the constructor finishes (watch out for initialize methods) • Control flow (conditional or looping logic) in a constructor • CL does complex object graph construction inside a constructor rather than using a factory or builder • Adding or using an initialization block

Why this is a Flaw

When your constructor has to instantiate and initialize its collaborators, the result tends to be an inflexible and prematurely coupled design. Such constructors shut off the ability to inject test collaborators when testing.

It violates the Single Responsibility Principle
When collaborator construction is mixed with initialization, it suggests that there is only one way to configure the class, which closes off reuse opportunities that might otherwise be available. Object graph creation is a full fledged responsibility — a different one from why a class exists in the first place. Doing such work in a constructor violates the Single Responsibility Principle.

Testing Directly is Difficult
Testing such constructors is difficult. To instantiate an object, the constructor must execute. And if that constructor does lots of work, you are forced to do that work when creating the object in tests. If collaborators access external resources (e.g. files, network services, or databases), subtle changes in collaborators may need to be reflected in the constructor, but may be missed due to missing test coverage from tests that weren't written because the constructor is so difficult to test. We end up in a vicious cycle.

Subclassing and Overriding to Test is Still Flawed
Other times a constructor does little work itself, but delegates to a method that is expected to be overridden in a test subclass. This may work around the problem of difficult construction, but using the "subclass to test" trick is something you only should do as a last resort. Additionally, by subclassing, you will fail to test the method that you override. And that method does lots of work (remember - that's why it was created in the first place), so it probably should be tested.

It Forces Collaborators on You

Sometimes when you test an object, you don't want to actually create all of its collaborators. For instance, you don't want a real MySqlRepository object that talks to the MySql service. However, if they are directly created using new MySqlRepositoryServiceThatTalksToOtherServers() inside your System Under Test (SUT), then you will be forced to use that heavyweight object.

It Erases a "Seam"

Seams are places you can slice your codebase to remove dependencies and instantiate small, focused objects. When you do new XYZ() in a constructor, you'll never be able to get a different (subclass) object created. (See Michael Feathers book Working Effectively with Legacy Code for more about seams).

It Still is a Flaw even if you have Multiple Constructors (Some for "Test Only")

Creating a separate "test only" constructor does not solve the problem. The constructors that do work will still be used by other classes. Even if you can test this object in isolation (creating it with the test specific constructor), you're going to run into other classes that use the hard-to-test constructor. And when testing those other classes, your hands will be tied.

Bottom Line

It all comes down to how hard or easy it is to construct the class *in isolation* or *with test-double collaborators*.

- If it's hard, you're doing too much work in the constructor!
- If it's easy, pat yourself on the back.

Always think about how hard it will be to test the object while you are writing it. Will it be easy to instantiate it via the constructor you're writing? (Keep in mind that your test-class will not be the only code where this class will be instantiated.)

So many designs are full of "*objects that instantiate other objects or retrieve objects from globally accessible locations. These programming practices, when left unchecked, lead to highly coupled designs that are difficult to test.*" [J.B. Rainsberger, JUnit Recipes, Recipe 2.11]

Recognizing the Flaw

Examine for these symptoms:

- The new keyword constructs anything you would like to replace with a test-double in a test? (Typically this is anything bigger than a simple value object).
- Any static method calls? (Remember: static calls are non-mockable, and non-injectable, so if you see Server.init() or anything of that ilk, warning sirens should go off in your head!)
- Any conditional or loop logic? (You will have to successfully navigate the logic every time you instantiate the object. This will result in excessive setup code not only when you test the class directly, but also if you happen to need this class while testing any of the related classes.)

Think about one fundamental question when writing or reviewing code:

How am I going to test this?

"If the answer is not obvious, or it looks like the test would be ugly or hard to write, then take that as a warning signal. Your design probably needs to be modified; change things around until the code is easy to test, and your design will end up being far better for the effort." [Hunt, Thomas. Pragmatic Unit Testing in Java with JUnit, p 103 (somewhat dated, but a decent and quick read)]

Note: Constructing **value objects** may be acceptable in many cases (examples: LinkedList; HashMap, User, EmailAddress, CreditCard). Value objects key attributes are: (1) Trivial to construct (2) are state focused (lots of getters/setters low on behavior) (3) do not refer to any service object.

Fixing the Flaw

Do not create collaborators in your constructor, but pass them in

Move the responsibility for object graph construction and initialization into another object. (e.g. extract a builder, factory or Provider, and pass these collaborators to your constructor).
Example: If you depend on a DatabaseService (hopefully that's an interface), then use Dependency Injection (DI) to pass in to the constructor the exact subclass of DatabaseService object you need.

To repeat: ***Do not create collaborators in your constructor***, but pass them in. (Don't look for things! Ask for things!)
If there is initialization that needs to happen with the objects that get passed in, you have three options:

1. *Best Approach using Guice:* Use a Provider<YourObject> to create and initialize YourObject's constructor arguments. Leave the responsibility of object initialization and graph construction to Guice. This will remove the need to initialize the objects on-the-go. Sometimes you will use Builders or Factories in addition to Providers, then pass in the builders and factories to the constructor.
2. *Best Approach using manual Dependency Injection:* Use a Builder, or a Factory, for YourObject's constructor arguments. Typically there is one factory for a whole graph of objects, see example below. (So you don't have to worry about having class explosion due to one factory for every class) The responsibility of the factory is to create the object graph and to do no work. (All you should see in the factory is a whole lot of new keywords and passing around of references). The responsibility of the object graph is to do work, and to do no object instantiation (There should be a serious lack of new keywords in application logic classes).
3. *Only as a Last Resort:* Have an init(…) method in your class that can be called after construction. Avoid this wherever you can, preferring the use of another object who's single responsibility is to configure the parameters for this object. (Often that is a Provider if you are using Guice)

(Also read the code examples below)

Concrete Code Examples Before and After

Fundamentally, “Work in the Constructor” amounts to doing anything that makes *instantiating your object difficult* or *introducing test-double objects difficult*.

Problem: “new” Keyword in the Constructor or at Field Declaration

Before: Hard to Test	After: Testable and Flexible Design

`// Basic new operators called directly in`
`//   the class' constructor. (Forever`
`//   preventing a seam to create different`
`//   kitchen and bedroom collaborators).`
`class House {`
`  Kitchen kitchen = new Kitchen();`
`  Bedroom bedroom;`
 	
`  House() {`
`    bedroom = new Bedroom();`
`  }`
 	
`  // ...`
`}`
	`class House {`
	`  Kitchen kitchen;`
	`  Bedroom bedroom;`
	
	`  // Have Guice create the objects`
	`  //   and pass them in`
	`  @Inject`
	`  House(Kitchen k, Bedroom b) {`
	`    kitchen = k;`
	`    bedroom = b;`
	`  }`
	`  // ...`
	`}`

`// An attempted test that becomes pretty hard`
 	
`class HouseTest extends TestCase {`
`  public void testThisIsReallyHard() {`
`    House house = new House();`
`    // Darn! I'm stuck with those Kitchen and`
`    //   Bedroom objects created in the`
`    //   constructor.`
 	
`    // ...`
`  }`
`}`
	`// New and Improved is trivially testable, with any`
	`//   test-double objects as collaborators.`
	
	`class HouseTest extends TestCase {`
	`  public void testThisIsEasyAndFlexible() {`
	`    Kitchen dummyKitchen = new DummyKitchen();`
	`    Bedroom dummyBedroom = new DummyBedroom();`
	
	`    House house =`
	`        new House(dummyKitchen, dummyBedroom);`
	
	`    // Awesome, I can use test doubles that`
	`    //   are lighter weight.`
	
	`    // ...`
	`  }`
	`}`

This example mixes object graph creation with logic. In tests we often want to create a different object graph than in production. Usually it is a smaller graph with some objects replaced with test-doubles. By leaving the new operators inline we will never be able to create a graph of objects for testing. See: “How to think about the new operator”

- Flaw: inline object instantiation where fields are declared has the same problems that work in the constructor has.
- Flaw: this may be easy to instantiate but if Kitchen represents something expensive such as file/database access it is not very testable since we could never replace the Kitchen or Bedroom with a test-double.
- Flaw: Your design is more brittle, because you can never polymorphically replace the behavior of the kitchen or bedroom in the House.

If the Kitchen is a value object such as: Linked List, Map, User, Email Address, etc., then we can create them inline as long as the value objects do not reference service objects. Service objects are the type most likely that need to be replaced with test-doubles, so you never want to lock them in with direct instantiation or instantiation via static method calls.

Problem: Constructor takes a partially initialized object and has to set it up

Before: Hard to Test

```
// SUT initializes collaborators. This prevents
//   tests and users of Garden from altering them.

class Garden {
  Garden(Gardener joe) {
    joe.setWorkday(new TwelveHourWorkday());
    joe.setBoots(
       new BootsWithMassiveStaticInitBlock());
    this.joe = joe;
  }

  // ...
}
```

After: Testable and Flexible Design

```
// Let Guice create the gardener, and have a
//   provider configure it.

class Garden {
  Gardener joe;

  @Inject
  Garden(Gardener joe) {
    this.joe = joe;
  }

  // ...
}

// In the Module configuring Guice.

@Provides
Gardener getGardenerJoe(Workday workday,
    BootsWithMassiveStaticInitBlock badBoots) {
  Gardener joe = new Gardener();
  joe.setWorkday(workday);

  // Ideally, you'll refactor the static init.
  joe.setBoots(badBoots);
  return joe;
}
```

Before: Hard to Test

```
// A test that is very slow, and forced
//   to run the static init block multiple times.

class GardenTest extends TestCase {

  public void testMustUseFullFledgedGardener() {
    Gardener gardener = new Gardener();
    Garden garden = new Garden(gardener);

    new AphidPlague(garden).infect();
    garden.notifyGardenerSickShrubbery();

    assertTrue(gardener.isWorking());

  }
}
```

After: Testable and Flexible Design

```
// The new tests run quickly and are not
//   dependent on the slow
//   BootsWithMassiveStaticInitBlock

class GardenTest extends TestCase {

  public void testUsesGardenerWithDummies() {
    Gardener gardener = new Gardener();
    gardener.setWorkday(new OneMinuteWorkday());
    // Okay to pass in null, b/c not relevant
    //   in this test.
    gardener.setBoots(null);

    Garden garden = new Garden(gardener);

    new AphidPlague(garden).infect();
    garden.notifyGardenerSickShrubbery();

    assertTrue(gardener.isWorking());
  }
}
```

Object graph creation (creating and configuring the Gardener collaborator for Garden) is a different responsibility than what the Garden should do. When configuration and instantiation is mixed together in the constructor, objects become more brittle and tied to concrete object graph structures. This makes code harder to modify, and (more or less) impossible to test.

- Flaw: The Garden needs a Gardener, but it should not be the responsibility of the Garden to configure the gardener.
- Flaw: In a unit test for Garden the workday is set specifically in the constructor, thus forcing us to have Joe work a 12 hour workday. Forced dependencies like this can cause tests to run slow. In unit tests, you'll want to pass in a shorter workday.
- Flaw: You can't change the boots. You will likely want to use a test-double for boots to avoid the problems with loading and using BootsWithMassiveStaticInitBlock. (Static initialization blocks are often dangerous and troublesome, especially if they interact with global state.)

Have two objects when you need to have collaborators initialized. Initialize them, and then pass them fully initialized into the constructor of the class of interest.

Problem: Violating the Law of Demeter in Constructor

Before: Hard to Test

```
// Violates the Law of Demeter
// Brittle because of excessive dependencies
// Mixes object lookup with assignment

class AccountView {
  User user;
  AccountView() {
   user = RPCClient.getInstance().getUser();
  }
}
```

After: Testable and Flexible Design

```
class AccountView {
  User user;

  @Inject
  AccountView(User user) {
    this.user = user;
  }
}

// The User is provided by a GUICE provider
@Provides
User getUser(RPCClient rpcClient) {
  return rpcClient.getUser();
}

// RPCClient is also provided, and it is no longer
//   a JVM Singleton.
@Provides @Singleton
RPCClient getRPCClient() {
  // we removed the JVM Singleton
  //   and have GUICE manage the scope
  return new RPCClient();
}
```

```
// Hard to test because needs real RPCClient
class ACcountViewTest extends TestCase {

  public void testUnfortunatelyWithRealRPC() {
    AccountView view = new AccountView();
    // Shucks! We just had to connect to a real
    //   RPCClient. This test is now slow.

    // ...
  }
}
```

```
// Easy to test with Dependency Injection
class AccountViewTest extends TestCase {

  public void testLightweightAndFlexible() {
    User user = new DummyUser();
    AccountView view = new AccountView(user);
    // Easy to test and fast with test-double
    //   user.

    // ...
  }
}
```

In this example we reach into the global state of an application and get a hold of the RPCClient singleton. It turns out we don't need the singleton, we only want the User. First: we are doing work (against static methods, which have zero seams). Second: this violates the "Law of Demeter".

- Flaw: We cannot easily intercept the call RPCClient.getInstance() to return a mock RPCClient for testing. (Static methods are non-interceptable, and non-mockable).
- Flaw: Why do we have to mock out RPCClient for testing if the class under test does not need RPCClient?(AccountView doesn't persist the rpc instance in a field.) We only need to persist the User.
- Flaw: Every test which needs to construct class AccountView will have to deal with the above points. Even if we solve the issues for one test, we don't want to solve them again in other tests. For example AccountServlet may need AccountView. Hence in AccountServlet we will have to successfully navigate the constructor.

In the improved code only what is directly needed is passed in: the User collaborator. For tests, all you need to create is a (real or test-double) User object. This makes for a more flexible design *and* enables better testability.

We use Guice to provide for us a User, that comes from the RPCClient. In unit tests we won't use Guice, but directly create the User and AccountView.

Problem: Creating Unneeded Third Party Objects in Constructor.

Before: Hard to Test

```
// Creating unneeded third party objects,
//   Mixing object construction with logic, &
//   "new" keyword removes a seam for other
//   EngineFactory's to be used in tests.
//   Also ties you to the (slow) file system.

class Car {
  Engine engine;
  Car(File file) {
    String model = readEngineModel(file);
    engine = new EngineFactory().create(model);
  }

  // ...
}
```

After: Testable and Flexible Design

```
// Asks for precisely what it needs

class Car {
  Engine engine;

  @Inject
  Car(Engine engine) {
    this.engine = engine;
  }

  // ...
}

// Have a provider in the Module
//   to give you the Engine
@Provides
Engine getEngine(
    EngineFactory engineFactory,
    @EngineModel String model) {
  //
  return engineFactory
      .create(model);
}

// Elsewhere there is a provider to
//   get the factory and model
```

```
// The test exposes the brittleness of the Car
class CarTest extends TestCase {

  public void testNoSeamForFakeEngine() {
    // Aggh! I hate using files in unit tests
    File file = new File("engine.config");
    Car car = new Car(file);

    // I want to test with a fake engine
    //   but I can't since the EngineFactory
    //   only knows how to make real engines.
  }
}
```

```
// Now we can see a flexible, injectible design
class CarTest extends TestCase {

  public void testShowsWeHaveCleanDesign() {
    Engine fakeEngine = new FakeEngine();
    Car car = new Car(fakeEngine);

    // Now testing is easy, with the car taking
    //   exactly what it needs.
  }
}
```

Linguistically, it does not make sense to require a Car to get an EngineFactory in order to create its own engine. Cars should be given readymade engines, not figure out how to create them. The car you ride in to work shouldn't have a reference back to its factory. In the same way, some constructors reach out to third party objects that aren't directly needed — only something the third party object can create is needed.

- Flaw: Passing in a file, when all that is ultimately needed is an Engine.
- Flaw: Creating a third party object (EngineFactory) and paying any assorted costs in this non-injectable and non-overridable creation. This makes your code more brittle because you can't change the factory, you can't decide to start caching them, and you can't prevent it from running when a new Car is created.
- Flaw: It's silly for the car to know how to build an EngineFactory, which then knows how to build an engine. (Somehow when these objects are more abstract we tend to not realize we're making this mistake).
- Flaw: Every test which needs to construct class Car will have to deal with the above points. Even if we solve the issues for one test, we don't want to solve them again in other tests. For example another test for a Garage may need a Car. Hence in Garage test I will have to successfully navigate the Car constructor. And I will be forced to create a new EngineFactory.

- Flaw: Every test will need a access a file when the Car constructor is called. This is slow, and prevents test from being true unit tests.

Remove these third party objects, and replace the work in the constructor with simple variable assignment. Assign pre-configured variables into fields in the constructor. Have another object (a factory, builder, or Guice providers) do the actual construction of the constructor's parameters. Split off of your primary objects the responsibility of object graph construction and you will have a more flexible and maintainable design.

Problem: Directly Reading Flag Values in Constructor

Before: Hard to Test

```
// Reading flag values to create collaborators

class PingServer {
  Socket socket;
  PingServer() {
    socket = new Socket(FLAG_PORT.get());
  }

  // ...
}
```

```
// The test is brittle and tied directly to a
//   Flag's static method (global state).

class PingServerTest extends TestCase {
  public void testWithDefaultPort() {
    PingServer server = new PingServer();
    // This looks innocent enough, but really
    //   it forces you to mutate global state
    //   (the flag) to run on another port.
  }
}
```

After: Testable and Flexible Design

```
// Best solution (although you also could pass
//   in an int of the Socket's port to use)

class PingServer {
  Socket socket;

  @Inject
  PingServer(Socket socket) {
    this.socket = socket;
  }
}

// This uses the FlagBinder to bind Flags to
// the @Named annotation values. Somewhere in
// a Module's configure method:
  new FlagBinder(
      binder().bind(FlagsClassX.class));

// And the method provider for the Socket
  @Provides
  Socket getSocket(@Named("port") int port) {
    // The responsibility of this provider is
    //   to give a fully configured Socket
    //   which may involve more than just "new"
    return new Socket(port);
  }
```

```
// The revised code is flexible, and easily
//   tested (without any global state).

class PingServerTest extends TestCase {
  public void testWithNewPort() {
    int customPort = 1234;
    Socket socket = new Socket(customPort);
    PingServer server = new PingServer(socket);

    // ...
  }
}
```

What looks like a simple no argument constructor actually has a lot of dependencies. Once again the API is lying to you, pretending it is easy to create, but actually PingServer is brittle and tied to global state.

- Flaw: In your test you will have to rely on global variable FLAG_PORT in order to instantiate the class. This will make your tests flaky as the order of tests matters.
- Flaw: Depending on a statically accessed flag value prevents you from running tests in parallel. Because parallel running test could change the flag value at the same time, causing failures.
- Flaw: If the socket needed additional configuration (i.e. calling setSoTimeout()), that can't happen because the object construction happens in the wrong place. Socket is created inside the PingServer, which is backwards. It needs to happen externally, in something whose sole responsibility is object graph construction — i.e. a Guice provider.

PingServer ultimately needs a socket not a port number. By passing in the port number we will have to tests with real sockets/threads. By passing in a socket we can create a mock socket in tests and test the class without any real sockets / threads. Explicitly passing in the port number removes the dependency on global state and greatly simplifies testing. Even better is passing in the socket that is ultimately needed.

Problem: Directly Reading Flags and Creating Objects in Constructor

Before: Hard to Test

```
// Branching on flag values to determine state.

class CurlingTeamMember {
  Jersey jersey;

  CurlingTeamMember() {
    if (FLAG_isSuedeJersey.get()) {
      jersey = new SuedeJersey();
    } else {
      jersey = new NylonJersey();
    }
  }
}
```

```
// Testing the CurlingTeamMember is difficult.
//   In fact you can't use any Jersey other
//   than the SuedeJersey or NylonJersey.

class CurlingTeamMemberTest extends TestCase {
  public void testImpossibleToChangeJersey() {
    //  You are forced to use global state.
    // ... Set the flag how you want it
    CurlingTeamMember russ =
        new CurlingTeamMember();

    // Tests are locked in to using one
    //   of the two jerseys above.
  }
}
```

After: Testable and Flexible Design

```
// We moved the responsibility of the selection
//   of Jerseys into a provider.

class CurlingTeamMember {
  Jersey jersey;

  // Recommended, because responsibilities of
  // Construction/Initialization and whatever
  // this object does outside it's constructor
  // have been separated.
  @Inject
  CurlingTeamMember(Jersey jersey) {
    this.jersey = jersey;
  }
}

// Then use the FlagBinder to bind flags to
//   injectable values. (Inside your Module's
//   configure method)
  new FlagBinder(
      binder().bind(FlagsClassX.class));

// By asking for Provider<SuedeJersey>
//   instead of calling new SuedeJersey
//   you leave the SuedeJersey to be free
//   to ask for its dependencies.
@Provides
Jersey getJersey(
     Provider<SuedeJersey> suedeJerseyProvider,
     Provider<NylonJersey> nylonJerseyProvider,
     @Named('isSuedeJersey') Boolean suede) {
  if (suede) {
    return suedeJerseyProvider.get();
  } else {
    return nylonJerseyProvider.get();
  }
}
```

```
// The code now uses a flexible alternataive:
//   dependency injection.

class CurlingTeamMemberTest extends TestCase {

  public void testWithAnyJersey() {
    // No need to touch the flag
    Jersey jersey = new LightweightJersey();
    CurlingTeamMember russ =
        new CurlingTeamMember(jersey);

    // Tests are free to use any jersey.
  }
}
```

Guice has something called the FlagBinder which lets you–at a very low cost–remove flag references and replace them with injected values. Flags are pervasively used to change runtime parameters, yet we don't have to directly read the flags for their global state.

- Flaw: Directly reading flags is reaching out into global state to get a value. This is undesirable because global state is not isolated: previous tests could set it to a different value, or other threads could mutate it unexpectedly.

- Flaw: Directly constructing the differing types of Jersey, depending on a flag's value. Your tests that instantiate a CurlingTeamMember have no seam to inject a different Jersey collaborator for testing.
- Flaw: The responsibility of the CurlingTeamMember is broad: both whatever the core purpose of the class, and now also Jersey configuration. Passing in a preconfigured Jersey object instead is preferred. Another object can have the responsibility of configuring the Jersey.

Use the FlagBinder (is a class you write which knows how to bind command line flags to injectable parameters) to attach all the flags from a class into Guice's scope of what is injectable.

Problem: Moving the Constructor's "work" into an Initialize Method

Before: Hard to Test

```
// With statics, singletons, and a tricky
//   initialize method this class is brittle.

class VisualVoicemail {
  User user;
  List<Call> calls;

  @Inject
  VisualVoicemail(User user) {
    // Look at me, aren't you proud? I've got
    // an easy constructor, and I use Guice
    this.user = user;
  }

  initialize() {
     Server.readConfigFromFile();
     Server server = Server.getSingleton();
     calls = server.getCallsFor(user);
  }

  // This was tricky, but I think I figured
  // out how to make this testable!
  @VisibleForTesting
  void setCalls(List<Call> calls) {
    this.calls = calls;
  }

  // ...
}
```

```
// Brittle code exposed through the test

class VisualVoicemailTest extends TestCase {

  public void testExposesBrittleDesign() {
    User dummyUser = new DummyUser();
    VisualVoicemail voicemail =
        new VisualVoicemail(dummyUser);
    voicemail.setCalls(buildListOfTestCalls());

    // Technically this can be tested, as long
    //   as you don't need the Server to have
    //   read the config file. But testing
    //   without testing the initialize()
    //   excludes important behavior.

    // Also, the code is brittle and hard to
    //   later on add new functionalities.
  }
}
```

After: Testable and Flexible Design

```
// Using DI and Guice, this is a
//   superior design.

class VisualVoicemail {
  List<Call> calls;

  VisualVoicemail(List<Call> calls) {
    this.calls = calls;
  }
}

// You'll need a provider to get the calls
@Provides
List<Call> getCalls(Server server,
    @RequestScoped User user) {
  return server.getCallsFor(user);
}

// And a provider for the Server. Guice will
//  let you get rid of the JVM Singleton too.
@Provides @Singleton
Server getServer(ServerConfig config) {
  return new Server(config);
}

@Provides @Singleton
ServerConfig getServerConfig(
    @Named("serverConfigPath") path) {
  return new ServerConfig(new File(path));
}

// Somewhere, in your Module's configure()
//   use the FlagBinder.
  new FlagBinder(binder().bind(
      FlagClassX.class))
```

```
// Dependency Injection exposes your
//   dependencies and allows for seams to
//   inject different collaborators.

class VisualVoicemailTest extends TestCase {

  VisualVoicemail voicemail =
      new VisualVoicemail(
          buildListOfTestCalls());

  // ... now you can test this however you want.
}
```

Moving the "work" into an initialize method is not the solution. You need to decouple your objects into single responsibilities. (Where one single responsibility is to provide a fully-configured object graph).

- Flaw: At first glance it may look like Guice is effectively used. For testing the VisualVoicemail object is very easy to construct. However the code is still brittle and tied to several Static initialization calls.
- Flaw: The initialize method is a glaring sign that this object has too many responsibilities: whatever a VisualVoicemail needs to do, and initializing its dependencies. Dependency initialization should happen in another class, passing *all* of the ready-to-be-used objects into the constructor.
- Flaw: The Server.readConfigFromFile() method is non interceptable when in a test, if you want to call the initialize method.
- Flaw: The Server is non-initializable in a test. If you want to use it, you're forced to get it from the global singleton state. If two tests run in parallel, or a previous test initializes the Server differently, global state will bite you.
- Flaw: Usually, @VisibleForTesting annotation is a smell that the class was not written to be easily tested. And even though it will let you set the list of calls, it is only a *hack* to get around the root problem in the initialize() method.

Solving this flaw, like so many others, involves removing JVM enforced global state and using Dependency Injection.

Problem: Having Multiple Constructors, where one is Just for Testing

Before: Hard to Test

```
// Half way easy to construct. The other half
//   expensive to construct. And for collaborators
//   that use the expensive constructor - they
//   become expensive as well.

class VideoPlaylistIndex {
  VideoRepository repo;

  @VisibleForTesting
  VideoPlaylistIndex(
      VideoRepository repo) {
    // Look at me, aren't you proud?
    // An easy constructor for testing!
    this.repo = repo;
  }

  VideoPlaylistIndex() {
    this.repo = new FullLibraryIndex();
  }

  // ...

}

// And a collaborator, that is expensive to build
//   because the hard coded index construction.

class PlaylistGenerator {

  VideoPlaylistIndex index =
      new VideoPlaylistIndex();

  Playlist buildPlaylist(Query q) {
    return index.search(q);
  }
}
```

```
// Testing the VideoPlaylistIndex is easy,
//  but testing the PlaylistGenerator is not!

class PlaylistGeneratorTest extends TestCase {

  public void testBadDesignHasNoSeams() {
    PlaylistGenerator generator =
        new PlaylistGenerator();
    // Doh! Now we're tied to the
    //   VideoPlaylistIndex with the bulky
    //   FullLibraryIndex that will make slow
    //   tests.
  }
}
```

After: Testable and Flexible Design

```
// Easy to construct, and no other objects are
//   harmed by using an expensive constructor.

class VideoPlaylistIndex {
  VideoRepository repo;

  VideoPlaylistIndex(
      VideoRepository repo) {
    // One constructor to rule them all
    this.repo = repo;
  }
}

// And a collaborator, that is now easy to
//   build.

class PlaylistGenerator {
  VideoPlaylistIndex index;

  // pass in with manual DI
  PlaylistGenerator(
      VideoPlaylistIndex index) {
    this.index = index;
  }

  Playlist buildPlaylist(Query q) {
    return index.search(q);
  }
}
```

```
// Easy to test when Dependency Injection
//   is used everywhere.

class PlaylistGeneratorTest extends TestCase {

  public void testFlexibleDesignWithDI() {
    VideoPlaylistIndex fakeIndex =
        new InMemoryVideoPlaylistIndex()
    PlaylistGenerator generator =
        new PlaylistGenerator(fakeIndex);

    // Success! The generator does not care
    //   about the index used during testing
    //   so a fakeIndex is passed in.
  }
}
```

Multiple constructors, with some only used for testing, is a hint that parts of your code will still be hard to test. On the left, the VideoPlaylistIndex is easy to test (you can pass in a test-double VideoRepository). However, whichever dependant objects which use the no-arg constructor will be hard to test.

- Flaw: PlaylistGenerator is hard to test, because it takes advantage of the no-arg constructor for VideoPlaylistIndex, which is hard coded to using the FullLibraryIndex.You wouldn't really want to test the FullLibraryIndex in a test of the PlaylistGenerator, but you are forced to.
- Flaw: Usually, the @VisibleForTesting annotation is a smell that the class was not written to be easily tested. And even though it will let you set the list of calls, it is only a *hack* to get around the root problem.

Ideally the PlaylistGenerator asks for the VideoPlaylistIndex in its constructor instead of creating its dependency directly. Once PlaylistGenerator asks for its dependencies no one calls the no argument VideoPlaylistIndex constructor and we are free to delete it. We don't usually need multiple constructors.

Frequently Asked Questions

Q1: Okay, so I think I get it. I'll *only* do assignment in the constructor, and then I'll have an init() method or init(…) to do all the work that used to be in the constructor. Does that sound okay?
A: We discourage this, see the code example above.

Q2: What about multiple constructors? Can I have one simple to construct, and the other with lots of work? I'll only use the easy one in my tests. I'll even mark it @VisibleForTesting. Okay?
A: We discourage this, see the code example above.

Q3: Can I create named constructors as additional constructors which may do work? I'll have a simple assignment to fields only constructor to use for my tests.
A: Someone actually ask this and we'll elaborate.

Q4: I keep hearing you guys talk about creating "Factories" and these objects whose responsibility is exclusively construction of object graphs. But seriously, that's too many objects, and too much for such a simple thing.
A: Typically there is one factory for a whole graph of objects, see example. So you don't have to worry about having class explosion due to one factory per class. The responsibility of the factory is to create the object graph and to do no work (All you should see is a whole lot of new keywords and passing around of references). The responsibility of the object graph is to do work, and to do no object instantiation.

Flaw: Digging into Collaborators

Avoid “holder”, “context”, and “kitchen sink” objects (these take all sorts of other objects and are a grab bag of collaborators). Pass in the specific object you need as a parameter, instead of a holder of that object. Avoid reaching into one object, to get another, etc. Do not look for the object of interest by walking the object graph (which needlessly makes us intimate with all of the intermediary objects. The chain does not need to be deep. If you count more than one period in the chain, you’re looking right at an example of this flaw. *Inject directly the object(s) that you need, rather than walking around and finding them.*

Warning Signs
• Objects are passed in but never used directly (only used to get access to other objects) • Law of Demeter violation: method call chain walks an object graph with more than one dot (.) • Suspicious names: context, environment, principal, container, or manager

Why this is a Flaw

Deceitful API
If your API says that all you need is a credit card number as a String, but then the code secretly goes off and dig around to find a CardProcessor, or a PaymentGateway, the real dependencies are not clear. Declare your dependencies in your API (the method signature, or object’s constructor) the collaborators which you really need.

Makes for Brittle Code
Imagine when something needs to change, and you have all these “Middle Men” that are used to dig around in and get the objects you need. Many of them will need to be modified to accommodate new interactions. Instead, try to get the most specific object that you need, where you need it. (In the process you may discover a class needs to be split up, because it has more than one responsibility. Don’t be afraid; strive for the Single Responsibility Principle). Also, when digging around to find the object you need, you are intimate with the intermediaries, and they are now needed as dependencies for compilation.

Bloats your Code and Complicates what’s Really Happening
With all these intermediary steps to dig around and find what you want; your code is more confusing. And it’s longer than it needs to be.

Hard for Testing
If you have to test a method that takes a context object, when you exercise that method it’s hard to guess what is pulled out of the context, and what isn’t cared about. Typically this means we pass in an

empty context, get some null pointers, then set some state in the context and try again. We repeat until all the null pointers go away. See also Breaking the Law of Demeter is Like Looking for a Needle in the Haystack.

Recognizing the Flaw

This is also known as a "Train Wreck" or a "Law of Demeter" violation (See Wikipedia on the Law of Demeter).

- Symptom: having to create mocks that return mocks in tests
- Symptom: reading an object named "context"
- Symptom: seeing more than one period "." in a method chaining where the methods are getters
- Symptom: difficult to write tests, due to complex fixture setup

Example violations:
getUserManager().getUser(123).getProfile().isAdmin() // this is egregiously bad (all you need to know if the user is an admin)

context.getCommonDataStore().find(1234) // this is bad

Fixing the Flaw

Instead of looking for things, simply ask for the objects you need in the constructor or method parameters. By asking for your dependencies in constructor you move the responsibility of object finding to the factory which created the class, typically a factory or GUICE.

- Only talk to your immediate friends.
- Inject (pass in) the more specific object that you really need.
- Leave the object location and configuration responsibility to the caller ie the factory or GUICE.

Concrete Code Examples Before and After

Fundamentally, "Digging into Collaborators" is whenever you don't have the actual object you need, and you need to dig around to get it. Whenever code starts going around asking for other objects, that is a clue that it is going to be hard to test. Instead of asking other objects to get your collaborators, declare collaborators as dependencies in the parameters to the method or constructor. (Don't look for things; Ask for things!)

Problem: Service Object Digging Around in Value Object

Before: Hard to Test	After: Testable and Flexible Design

`// This is a service object that works with a value`
`// object (the User and amount).`
 	
`class SalesTaxCalculator {`
`  TaxTable taxTable;`
 	
`  SalesTaxCalculator(TaxTable taxTable) {`
`    this.taxTable = taxTable;`
`  }`
 	
`  float computeSalesTax(User user, Invoice invoice) {`
`    // note that "user" is never used directly`
 	
`    Address address = user.getAddress();`
`    float amount = invoice.getSubTotal();`
`    return amount * taxTable.getTaxRate(address);`
`  }`
`}`
	`// Reworked, it only asks for the specific`
	`// objects that it needs to collaborate with.`
	
	`class SalesTaxCalculator {`
	`  TaxTable taxTable;`
	
	`  SalesTaxCalculator(TaxTable taxTable) {`
	`    this.taxTable = taxTable;`
	`  }`
	
	`  // Note that we no longer use User, nor do we`
	`  // dig inside the address. (Note: We would`
	`  // use a Money, BigDecimal, etc. in reality).`
	`  float computeSalesTax(Address address,`
	`        float amount) {`
	`    return amount * taxTable.getTaxRate(address);`
	`  }`
	`}`

`// Testing exposes the problem by the amount`
`// of work necessary to build the object graph, and // test the small behavior you are interested in.`
 	
`class SalesTaxCalculatorTest extends TestCase {`
 	
`  SalesTaxCalculator calc =`
`      new SalesTaxCalculator(new TaxTable());`
`  // So much work wiring together all the`
`  // objects needed`
`  Address address =`
`      new Address("1600 Amphitheatre Parkway...");`
`  User user = new User(address);`
`  Invoice invoice =`
`      new Invoice(1, new ProductX(95.00));`
`  // ...`
`  assertEquals(`
`      0.09, calc.computeSalesTax(user, invoice), 0.05);`
`}`
	`// The new API is clearer in what collaborators`
	`// it needs.`
	
	`class SalesTaxCalculatorTest extends TestCase {`
	
	`  SalesTaxCalculator calc =`
	`      new SalesTaxCalculator(new TaxTable());`
	`    // Only wire together the objects that`
	`    // are needed`
	
	`    Address address =`
	`        new Address("1600 Amphitheatre Parkway...");`
	`    // ...`
	`    assertEquals(`
	`        0.09,`
	`        calc.computeSalesTax(address, 95.00),`
	`        0.05);`
	`  }`
	
	`}`

This example mixes object lookup with calculation. The core responsibility is to multiply an amount by a tax rate.

- Flaw: To test this class you need to instantiate a User and an Invoice and populate them with a Zip and an amount. This is an extra burden to testing.
- Flaw: For users of the method, it is unclear that all that is needed is an Address and an Invoice. (The API lies to you).
- Flaw: From code reuse point of view, if you wanted to use this class on another project you would also have to supply source code to unrelated classes such as Invoice, and User. (Which in turn may pull in more dependencies)

The solution is to declare the specific objects needed for the interaction through the method signature, and nothing more.

Problem: Service Object Directly Violating Law of Demeter

Before: Hard to Test

```
// This is a service object which violates the
//   Law of Demeter.

class LoginPage {
  RPCClient client;
  HttpRequest request;

  LoginPage(RPCClient client,
      HttpServletRequest request) {
    this.client = client;
    this.request = request;
  }

  boolean login() {
    String cookie = request.getCookie();
    return client.getAuthenticator()
        .authenticate(cookie);
  }
}
```

```
// The extensive and complicated easy mock
// usage is a clue that the design is brittle.

class LoginPageTest extends TestCase {

  public void testTooComplicatedThanItNeedsToBe() {
    Authenticator authenticator =
        new FakeAuthenticator();
    IMocksControl control =
        EasyMock.createControl();
    RPCClient client =
        control.createMock(RPCClient.class);
    EasyMock.expect(client.getAuthenticator())
        .andReturn(authenticator);
    HttpServletRequest request =
        control.createMock(HttpServletRequest.class);
    Cookie[] cookies =
        new Cookie[]{new Cookie("g", "xyz123")};
    EasyMock.expect(request.getCookies())
        .andReturn(cookies);
    control.replay();
    LoginPage page = new LoginPage(client, request);
    // ...
    assertTrue(page.login());
    control.verify();
}
```

After: Testable and Flexible Design

```


// The specific object we need is passed in
//   directly.

class LoginPage {

  LoginPage(@Cookie String cookie,
            Authenticator authenticator) {
    this.cookie = cookie;
    this.authenticator = authenticator;
  }

  boolean login() {
    return authenticator.authenticate(cookie);
  }
}
```

```


// Things now have a looser coupling, and are more
//   maintainable, flexible, and testable.

class LoginPageTest extends TestCase {

  public void testMuchEasier() {
    Cookie cookie = new Cookie("g", "xyz123");
    Authenticator authenticator =
        new FakeAuthenticator();
    LoginPage page =
        new LoginPage(cookie, authenticator);
    // ...
    assertTrue(page.login());
}
```

The most common Law of Demeter violations have many chained calls, however this example shows that you can violate it with a single chain. Getting the Authenticator from the RPCClient is a violation, because the RPCClient is not used elsewhere, and is only used to get the Authenticator.

- Flaw: Nobody actually cares about the RPCCllient in this class. Why are we passing it in?
- Flaw: Nobody actually cares about the HttpRequest in this class. Why are we passing it in?
- Flaw: The cookie is what we need, but we must dig into the request to get it. For testing, instantiating an HttpRequest is not a trivial matter.
- Flaw: The Authenticator is the real object of interest, but we have to dig into the RPCClient to get the Authenticator.

For testing the original bad code we had to mock out the RPCClient and HttpRequest. Also the test is very intimate with the implementation since we have to mock out the object graph traversal. In the fixed code we didn't have to mock any graph traversal. This is easier, and helps our code be less brittle. (Even if we chose to mock the Authenticator in the "after" version, it is easier, and produces a more loosely coupled design).

Problem: Law of Demeter Violated to Inappropriately make a Service Locator

Before: Hard to Test

```
// Database has an single responsibility identity
//   crisis.

class UpdateBug {

  Database db;

  UpdateBug(Database db) {
     this.db = db;
  }

  void execute(Bug bug) {
    // Digging around violating Law of Demeter
    db.getLock().acquire();
    try {
      db.save(bug);
    } finally {
      db.getLock().release();
    }
  }
}
```

```
// Testing even the happy path is complicated
// with all
//   the mock objects that are needed. Especially
//   mocks that take mocks (very bad).

class UpdateBugTest extends TestCase {

  public void testThisIsRidiculousHappyPath() {
    Bug bug = new Bug("description");

    // This both violates Law of Demeter and abuses
    //   mocks, where mocks aren't entirely needed.
    IMocksControl control =
        EasyMock.createControl();
    Database db =
        control.createMock(Database.class);
    Lock lock = control.createMock(Lock.class);

    // Yikes, this mock (db) returns another mock.
    EasyMock.expect(db.getLock()).andReturn(lock);
    lock.acquire();
    db.save(bug);
    EasyMock.expect(db.getLock()).andReturn(lock);
    lock.release();
    control.replay();
    // Now we're done setting up mocks, finally!

    UpdateBug updateBug = new UpdateBug(db);
    updateBug.execute(bug);
    // Verify it happened as expected
    control.verify();
    // Note: another test with multiple execute
    //   attempts would need to assert the specific
    //   locking behavior is as we expect.
  }
}
```

After: Testable and Flexible Design

```
// The revised Database has a
// Single Responsibility.

class UpdateBug {

  Database db;
  Lock lock;

  UpdateBug(Database db, Lock lock) {
     this.db = db;
  }

  void execute(Bug bug) {
    // the db no longer has a getLock method
    lock.acquire();
    try {
      db.save(bug);
    } finally {
      lock.release();
    }
  }
}
// Note: In Database, the getLock() method
// was removed
```

```
// Two improved solutions: State Based Testing
//   and Behavior Based (Mockist) Testing.

// First Sol'n, as State Based Testing.
class UpdateBugStateBasedTest extends TestCase {
  public void testThisIsMoreElegantStateBased() {
    Bug bug = new Bug("description");

    // Use our in memory version instead of a mock
    InMemoryDatabase db = new InMemoryDatabase();
    Lock lock = new Lock();
    UpdateBug updateBug = new UpdateBug(db, lock);

    // Utilize State testing on the in memory db.
    assertEquals(bug, db.getLastSaved());
  }
}

// Second Sol'n, as Behavior Based Testing.
//   (using mocks).
class UpdateBugMockistTest extends TestCase {

  public void testBehaviorBasedTestingMockStyle() {
    Bug bug = new Bug("description");

    IMocksControl control =
        EasyMock.createControl();
    Database db =
        control.createMock(Database.class);
    Lock lock = control.createMock(Lock.class);
    lock.acquire();
    db.save(bug);
    lock.release();
    control.replay();
    // Two lines less for setting up mocks.

    UpdateBug updateBug = new UpdateBug(db, lock);
    updateBug.execute(bug);
    // Verify it happened as expected
    control.verify();
  }
}
```

We need our objects to have one responsibility, and that is not to act as a Service Locator for other objects. When using Guice, you will be able to remove any existing Service Locators. Law of Demeter violations occur when one method acts as a locator in addition to its primary responsibility.

- Flaw: db.getLock() is outside the single responsibility of the Database. It also violates the law of demeter by requiring us to call db.getLock().acquire() and db.getLock().release() to use the lock.
- Flaw: When testing the UpdateBug class, you will have to mock out the Database's getLock method.
- Flaw: The Database is acting as a database, as well as a service locator (helping others to find a lock). It has an identity crisis. Combining Law of Demeter violations with acting like a Service Locator is worse than either problem individually. The point of the Database is not to distribute references to other services, but to save entities into a persistent store.

The Database's getLock() method should be eliminated. Even if Database needs to have a reference to a lock, it is a better if Database does not share it with others. *You should never have to mock out a setter or getter.*

Two solutions are shown: one using State Based Testing, the other with Behavior Based Testing. The first style asserts against the state of objects after work is performed on them. It is not coupled to the implementation, just that the result is in the state as expected. The second style uses mock objects to assert about the internal behavior of the System Under Test (SUT). Both styles are valid, although different people have strong opinions about one or the other.

Problem: Object Called "Context" is a Great Big Hint to look for a Violation

Before: Hard to Test	After: Testable and Flexible Design

`// Context objects can be a java.util.Map or some`
`//   custom grab bag of stuff.`
 	
`class MembershipPlan {`
 	
`  void processOrder(UserContext userContext) {`
`    User user = userContext.getUser();`
`    PlanLevel level = userContext.getLevel();`
`    Order order = userContext.getOrder();`
 	
`    // ... process`
`  }`
`}`
	`// Replace context with the specific parameters`
	`// that are needed within the method.`
	
	`class MembershipPlan {`
	
	`  void processOrder(User user, PlanLevel level,`
	`      Order order) {`
	
	`    // ... process`
	`  }`
	`}`

`// An example test method working against a`
`//   wretched context object.`
 	
`public void testWithContextMakesMeVomit() {`
`  MembershipPlan plan = new MembershipPlan();`
`  UserContext userContext = new UserContext();`
`  userContext.setUser(new User("Kim"));`
`  PlanLevel level = new PlanLevel(143, "yearly");`
`  userContext.setLevel(level);`
`  Order order =`
`      new Order("SuperDeluxe", 100, true);`
`  userContext.setOrder(order);`
 	
`  plan.processOrder(userContext);`
 	
`  // Then make assertions against the user, etc ...`
`}`
	`// The new design is simpler and will`
	`// easily evolve.`
	
	`public void testWithHonestApiDeclaringWhatItNeeds() {`
	`  MembershipPlan plan = new MembershipPlan();`
	`  User user = new User("Kim");`
	`  PlanLevel level = new PlanLevel(143, "yearly");`
	`  Order order =`
	`      new Order("SuperDeluxe", 100, true);`
	
	`  plan.processOrder(user, level, order);`
	
	`  // Then make assertions against the user, etc ...`
	`  }`
	`}`

Context objects may sound good in theory (no need to change signatures to change dependencies) but they are very hard to test.

- Flaw: Your API says all you need to test this method is a userContext map. But as a writer of the test, you have no idea what that actually is! Generally this means you write a test passing in null, or an empty map, and watch it fail, then progressively stuff things into the map until it will pass.
- Flaw: Some may claim the API is "flexible" (in that you can add any parameters without changing the signatures), but *really it is brittle* because you cannot use refactoring tools; users don't know what parameters are really needed. It is not possible to determine what its collaborators are just by examining the API. This makes it hard for new people on the project to understand the behavior and purpose of the class. We say that API lies about its dependencies.

The way to fix code using context objects is to replace them with the specific objects that are needed. This will expose true dependencies, and may help you discover how to decompose objects further to make an even better design.

When This is not a Flaw:

Caveat (Not a Problem): Domain Specific Languages can violate the Law of Demeter for Ease of Configuration

Breaking the Law of Demeter may be acceptable if working in some Domain Specific Languages. They violate the Law of Demeter for ease of configuration. This is not a problem because it is building up a value object, in a fluent, easily understandable way. An example is in Guice modules, building up the bindings.

```
// A DSL may be an acceptable violation.
//   i.e. in a GUICE Module's configure method
  bind(Some.class)
      .annotatedWith(Annotation.class)
      .to(SomeImplementaion.class)
      .in(SomeScope.class);
```

Flaw: Brittle Global State & Singletons

Accessing global state statically doesn't clarify those shared dependencies to readers of the constructors and methods that use the Global State. Global State and Singletons make APIs lie about their true dependencies. To really understand the dependencies, developers must read every line of code. It causes Spooky Action at a Distance: when running test suites, global state mutated in one test can cause a subsequent or parallel test to fail unexpectedly. Break the static dependency using manual or Guice dependency injection.

Warning Signs:
• Adding or using singletons • Adding or using static fields or static methods • Adding or using static initialization blocks • Adding or using registries • Adding or using service locators

NOTE: *When we say "Singleton" or "JVM Singleton" in this document, we mean the classic Gang of Four singleton. (We say that this singleton enforces its own "singletonness" though a static instance field). An "application singleton" on the other hand is an object which has a single instance in our application, but which does* not *enforce its own "singletonness."*

Why this is a Flaw

Global State Dirties your Design

The root problem with global state is that it is globally accessible. In an ideal world, an object should be able to interact only with other objects which were directly passed into it (through a constructor, or method call).

In other words, if I instantiate two objects A and B, and I never pass a reference from A to B, then neither A nor B can get hold of the other or modify the other's state. This is a very desirable property of code. *If I don't tell you about something, then it is not yours to interact with!* Notice how this not the case with global variables or with Singletons. Object A could — unknown to developers — get hold of singleton C and modify it. If, when object B gets instantiated, it too grabs singleton C, then A and B can affect each other through C. (Or interactions can overlap through C).

"The problem with using a Singleton is that it introduces a certain amount of coupling into a system — coupling that is almost always unnecessary. You are saying that your class can only collaborate with one particular implementation of a set of methods — the implementation that the Singleton provides.

You will allow no substitutes. This makes it difficult to test your class in isolation from the Singleton. The very nature of test isolation assumes the ability to substitute alternative implementations… for an object's collaborators. … [U]nless you change your design, you are forced to rely on the correct behavior of the Singleton in order to test any of its clients." [J.B. Rainsberger, Junit Recipes, Recipe 14.4]

Global State enables Spooky Action at a Distance

Spooky Action at a Distance is when we run one thing that we believe is isolated (since we did not pass any references in) but unexpected interactions and state changes happen in distant locations of the system which we did not tell the object about. This can only happen via global state. For instance, let's pretend we call printPayrollReport(Employee employee). This method lives in a class that has two fields: Printer and PayrollDatabase.

Very sinister Spooky Action at a Distance would become possible if this also initiated direct deposit at the bank, via a global method: BankAccount.payEmployee(Employee employee, Money amount). Your code authors would never do that … or would they?

How about this for a less sinister example. What if printPayrollReport also sent a notification out to PrinterSupplyVendor.notifyTonerUsed(int pages, double perPageCoverage)? This may seem like a convenience — when you print the payroll report, also notify the supplies vendor how much toner you used, so that you'll automatically get a refill before you run out. It is not a convenience. It is bad design and, like all statics, it makes your code base harder to understand, evolve, test, and refactor.

You may not have thought of it this way before, but whenever you use static state, you're creating secret communication channels and not making them clear in the API. Spooky Action at a Distance forces developers to read every line of code to understand the potential interactions, lowers developer productivity, and confuses new team members.

Do not approve code that uses statics and allows for a Spooky code base. Favor dependency injection of the specific collaborators needed (a PrinterSupplyVendor instance in the example above … perhaps scoped as a Guice singleton if there's only one in the application).

Global State & Singletons makes for Brittle Applications (and Tests)

- Static access prevents collaborating with a subclass or wrapped version of another class. By hard-coding the dependency, we lose the power and flexibility of polymorphism.
- Every test using global state needs it to start in an expected state, or the test will fail. But another object might have mutated that global state in a previous test.
- Global state often prevents tests from being able to run in parallel, which forces test suites to run slower.
- If you add a new test (which doesn't clean up global state) and it runs in the middle of the suite, another test may fail that runs after it.
- Singletons enforcing their own "Singletonness" end up cheating.
    - You'll often see mutator methods such as reset() or setForTest(…) on so-called singletons, because you'll need to change the instance during tests. If you forget to reset

the Singleton after a test, a later use will use the stale underlying instance and may fail in a way that's difficult to debug.

Global State & Singletons turn APIs into Liars

Let us look at a test we want to write:

```
testActionAtADistance() {
  CreditCard = new CreditCard("4444444444444441", "01/11");
  assertTrue(card.charge(100));
  // but this test fails at runtime!
}
```

Charging a credit card takes more then just modifying the internal state of the credit card object. It requires that we talk to external systems. We need to know the URL, we need to authenticate, we need to store a record in the database. But none of this is made clear when we look at how CreditCard is used. We say that the CreditCard API is lying. Let's try again:

```
testActionAtADistanceWithInitializtion() {
  // Global state needs to get set up first
  Database.init("dbURL", "user", "password");
  CreditCardProcessor.init("http://processorurl", "security key", "vendor");

  CreditCard = new CreditCard("4444444444444441", "01/11");
  assertTrue(card.charge(100));

  // but this test still fails!
}
```

By looking at the API of CreditCard, there is no way to know the global state you have to initialize. Even looking at the source code of CreditCard will not tell you which initialization method to call. At best, you can find the global variable being accessed and from there try to guess how to initialize it.

Here is how you fix the global state. Notice how it is much clearer initializing the dependencies of CreditCard.

```
testUsingDependencyInjectedObjects() {
  Database db = new Database("dbURL", "user", "password");
  CreditCardProcessor processor = new CreditCardProcessor(db,
      "http://processorurl", "security key", "vendor");
  CreditCard = new CreditCard(processor, "4444444444444441", "01/11");
  assertTrue(card.charge(100));
}
```

Each object that you needed to create declared its dependencies in the API of its constructor. It is no longer ambiguous how to build the objects you need in order to test CreditCard.

By declaring the dependency explicitly, it is clear which objects need to be instantiated (in this case that the CreditCard needs a CreditCardProcessor which in turn needs Database).

Globality and "Global Load" is Transitive

We can define *"global load"* as how many variables are exposed for (direct or indirect) mutation through global state. The higher the number, the bigger the problem. Below, we have a global load of one.

```
class UniqueID {
    // Global Load of 1 because only nextID is exposed as global state
    private static int nextID = 0;
    int static get() {
         return nextID++;
    }
}
```

What would the global load be in the example below?

```
class AppSettings {
    static AppSettings instance = new AppSettings();
    int numberOfThreads = 10;
    int maxLatency = 20;
    int timeout = 30;
    private AppSettings(){} // To prevent people from instantiating

}
```

Here the problem is a bit more complicated. The instance field is declared as static final. By traversing the global instance we expose three variables: numberOfThreads, maxLatency, and timeout. Once we access a global variable, all variables accessed through it become global as well. The global load is 3.

From a behavior point of view, there is no difference between a global variable declared directly through a static and a variable made global transitively. They are equally damaging. An application can have only very few static variables and yet transitively accumulate a high global load.

A Singleton is Global State in Sheep's Clothing

Most software engineers will agree that Global State is undesirable. However, a Singleton creates Global State, yet so many people still use that in new code. Fight the trend and encourage people to use other mechanisms instead of a classical JVM Singleton. Often we don't really need singletons (object creation is pretty cheap these days). If you need to guarantee one shared instance per application, use Guice's Singleton Scope.

"But My Application Only has One Singleton" is Meaningless

Here is a typical singleton implementation of Cache.

```
class Cache {
    static final instance Cache = new Cache();

    Map<String, User> userCache = new HashMap<String, User>();
    EvictionStrategy eviction = new LruEvictionStrategy();

    private Cache(){} // private constructor //..
}
```

This singleton has an unboundedly high level of Global Load. (We can place an unlimited number of users in userCache). The Cache singleton exposes the Map<String, User> into global state, and thus exposes every user as globally visible. In addition, the internal state of each User, and the EvictionStrategy is also exposed as globally mutable. As we can see it is very easy for an innocent object to become entangled with global state. **Statements like "But my application only has one singleton" are meaningless, because *total exposed global state* is the transitive closure of the objects accessible from explicit global state.**

Global State in *Application Runtime* May Deceptively "Feel Okay"

At production we instantiate one instance of an application, hence global state is not a problem from a state collision point of view. (There is still a problem of dirtying your design — see above). It is uncommon to instantiate two copies of the same application in a single JVM, so it may "feel okay" to have global state in production. (One exception is a web server. It is common to have multiple instances of a web server running on different ports. Global state could be problematic there.) As we will soon see this "ok feeling" is misleading, and it actually is an Insidious Beast.

Global State in *Test Runtime* is an Insidious Beast

At test time, each test is an isolated partial instantiation of an application. No external state enters the test (there is no external object passed into the tests constructor or test method). And no state leaves the tests (the test methods are void, returning nothing). When an ideal test completes, all state related to that test disappears. This makes tests isolated and all of the objects it created are subject to garbage collection. In addition the state created is confined to the current thread. This means we can run tests in parallel or in any order. However, when global state/singletons are present all of these nice assumptions break down. State can enter and leave the test and it is not garbage collected. This makes the order of tests matter. You cannot run the tests in parallel and your tests can become flaky due to thread interactions.

True singletons are most likely impossible to test. As a result most developers who try to test applications with singletons often relax the singleton property of their singletons into two ways. (1) They remove the final keyword from the static final declaration of the instance. This allows them to substitute different singletons for different tests. (2) they provide a second initalizeForTest() method which allows them to modify the singleton state. However, these solutions at best are a *hack* which produce hard to maintain and understand code. Every test (or tearDown) affecting any global state must undo those changes, or leak them to subsequent tests. And test isolation is nearly impossible if running tests in parallel.

Global state is the single biggest headache of unit testing!

Recognizing the Flaw

- Symptom: Presence of static fields
- Symptom: Code in the CL makes static method calls
- Symptom: A Singleton has initializeForTest(…), uninitialize(…), and other resetting methods (i.e. to tell it to use some light weight service instead of one that talks to other servers).

- Symptom: Tests fail when run in a suite, but pass individually or vice versa
- Symptom: Tests fail if you change the order of execution
- Symptom: Flag values are read or written to, or Flags.disableStateCheckingForTest() and Flags.enableStateCheckingForTest() is called.
- Symptom: Code in the CL has or uses Singletons, Mingletons, Hingletons, and Fingletons (see Google Singleton Detector.)

There is a distinction between global as in "JVM Global State" and global as in "Application Shared State."

- JVM Global State occurs when the static keyword is used to make accessible a field or a method that returns a shared object. The use of static in order to facilitate shared state is the problem. Because static is enforced as One Per JVM, parallelizing and isolating tests becomes a huge problem. From a maintenance point of view, static fields create coupling, hidden colaborators and APIs which lie about their true dependencies. *Static access is the root of the problem.*
- Application Shared State simply means the same instance is shared in multiple places throughout the code. There may or may not be multiple instances of a class instantiated at any one time, depending on whether application logic enforces uniqueness. The shared state is not accessed globally through the use of static. It is passed to collaborators who need the state, or Guice manages the consistent injection of needed shared state (i.e. During a request the same User needs to be injected into multiple objects within the thread that is servicing that user's request. Guice would scope that as @RequestScoped.) It is not shared state in and of itself that is a problem. There are places in an application that need access to shared state. It's sharing that state through *statics* that causes brittle code and difficulty for testing.

Test for JVM Global State:
Answer the following question: "Can I, in theory, create a second instance of your application in the same JVM, and not have any collisions?" If you can't, then you're using JVM Global State (using the static keyword, you're having the JVM enforce a singleton's singletoness). Use Dependency Injection with Guice instead for shared state.

Fixing the Flaw

Dependency Injection is your Friend.
Dependency Injection is your Friend.
Dependency Injection is your Friend.

- If you need a collaborator, use Dependency Injection (pass in the collaborator to the constructor). Dependency injection will make your collaborators clear, and give you seams for injecting test-doubles.
- If you need shared state, use Guice which can manage Application Scope singletons in a way that is still entirely testable.
- If a static is used widely in the codebase, and you cannot replace it with a Guice Singleton Scoped object in one CL, try one of these workaround:
    - Create an Adapter class. It will probably have just a default constructor, and methods of the Adapter will each be named the same as (and call through to) a static method you're

trying to decouple from. This doesn't fully fix the problems–the static access still exists, but at least the Adapter can be faked/mocked in testing. Once all consumers of a static method or utility filled with static methods have been adapted, the statics can be eliminated by pushing the shared behavior/state into the Adapters (turning them from adapters into full fledged collaborators). Use application logic or Guice scopes to enforce necessary sharing.
    - Rather than wrapping an adapter around the static methods, you can sometimes move the shared behavior/state into an instantiable class early. Have Guice manage the instantiable object (perhaps in @Singleton scope) and place a Guice-managed instance behind the static method, until all callers can be refactored to inject the instance instead of using it through the static method. Again, this is a half-solution that still retains statics, but it's a step toward removing the statics that may be useful when dealing with pervasive static methods.
    - When eliminating a Singleton in small steps, try binding a Guice Provider to the class you want to share in Scopes.Singleton (or use a provider method annotated @Singleton). The Provider returns an instance it retrieves from the GoF Singleton. Use Guice to inject the shared instance where possible, and once all sites can be injected, eliminate the GoF Singleton.
- If you're stuck with a library class' static methods, wrap it in an object that implements an interface. Pass in the object where it is needed. You can stub the interface for testing, and cut out the static dependency. See the example below.
- If using Guice, you can use GUICE to bind flag values to injectable objects. Then wherever you need the flag's value, inject it. For tests, you can pass in any value with dependency injection, bypassing the flag entirely and enabling easy parallelization.

Concrete Code Examples Before and After

Problem: You have a Singleton which your App can only have One Instance of at a Time

This often causes us to add a special method to change the singleton's instance during tests. An example with setForTest(…) methods is shown below. The solution is to use Guice to manage the singleton scope.

Problem: Need to Call setForTest(…) and/or resetForTest() Methods

Before: Hard to Test

```
// JVM Singleton needs to be swapped out in tests.

class LoginService {
  private static LoginService instance;

  private LoginService() {};

  static LoginService getInstance() {
    if (instance == null) {
      instance = new RealLoginService();
    }
    return instance;
  }

  // Call this at the start of your tests
  @VisibleForTesting
  static setForTest(LoginService testDouble) {
    instance = testDouble;
  }

  // Call this at the end of your tests, or you
  //   risk leaving the testDouble in as the
  //   singleton for subsequent tests.
  @VisibleForTesting
  static resetForTest() {
    instance = null;
  }

  // ...
}

// Elsewhere...
// A method uses the singleton

class AdminDashboard {

  //...

  boolean isAuthenticatedAdminUser(User user) {
    LoginService loginService =
        LoginService.getInstance();
    return loginService.isAuthenticatedAdmin(user);
  }
}
```

After: Testable and Flexible Design

```
// Guice managed Application Singleton is
//   Dependency Injected into where it is needed,
//   making tests very easy to create and run.

class LoginService {
  // removed the static instance
  // removed the private constructor
  // removed the static getInstance()

  // ... keep the rest
}

// In the Guice Module, tell Guice how to create
//   the LoginService as a RealLoginService, and
//   keep it in singleton scope.

  bind(LoginService.class)
      .to(RealLoginService.class)
      .in(Scopes.SINGLETON);

// Elsewhere...
// Where the single instance is needed

class AdminDashboard {

  LoginService loginService;

  // This is all we need to do, and the right
  //   LoginService is injected.
  @Inject
  AdminDashboard(LoginService loginService) {
    this.loginService = loginService;
  }

  boolean isAuthenticatedAdminUser(User user) {
    return loginService.isAuthenticatedAdmin(user);
  }
}
```

```
// Trying to write a test is painful!

class AdminDashboardTest extends TestCase {

  public void testForcedToUseRealLoginService() {
     // ...
     assertTrue(adminDashboard
         .isAuthenticatedAdminUser(user));
     // Arghh! Because of the Singleton, this is
     //   forced to use the RealLoginService()
}
```

```
// With DI, the test is now easy to write.

class AdminDashboardTest extends TestCase {

  public void testUsingMockLoginService() {
    // Testing is now easy, we just pass in a test-
    //   double LoginService in the constructor.
    AdminDashboard dashboard =
        new AdminDashboard(new MockLoginService());
    // ... now all tests will be small and fast
  }
}
```

For various reasons, it may be necessary to have only one of something in your application. Typically this is implemented as a Singleton [GoF], in which the class can give out one instance of an object, and it is impossible to instantiate two instances at the same time. There is a price to pay for such a JVM Singleton, and that price is flexibility and testability.People may work around these problems (by breaking encapsulation) with setForTest(…) and resetForTest() methods to alter the underlying singleton's instance.

- Flaw: As in all uses of static methods, there are no seams to polymorphically change the implementation. Your code becomes more fragile and brittle.
- Flaw: Tests cannot run in parallel, as each thread's mutations to shared global state will collide.

- Flaw: @VisibleForTesting is a hint that the class should be re-worked so that it does not need to break encapsulation in order to be tested. Notice how that is removed in the solution.

If you need a guarantee of "just one instance" in your application, tell Guice that object is in Singleton scope. Guice managed singletons are not a design problem, because in your tests you can create multiple instances (to run in parallel, preventing interactions, and under different configurations). During production runtime, Guice will ensure that the same instance is injected.

Problem: Tests with Static Flags have to Clean Up after Themselves

Before: Hard to Test

```
// Awkward and brittle tests, obfuscated by Flags'
//   boilerplate setup and cleanup.

class NetworkLoadCalculatorTest extends TestCase {

  public void testMaximumAlgorithmReturnsHighestLoad() {
    Flags.disableStateCheckingForTest();

ConfigFlags.FLAG_loadAlgorithm.setForTest("maximum");

    NetworkLoadCalculator calc =
        new NetworkLoadCalculator();
    calc.setLoadSources(10, 5, 0);
    assertEquals(10, calc.calculateTotalLoad());

    // Don't forget to clean up after
    // yourself following every test
    // (this could go in tearDown).
    ConfigFlags.FLAG_loadAlgorithm.resetForTest();
    Flags.enableStateCheckingForTest();
  }
}

// Elsewhere... the NetworkLoadCalculator's methods
class NetworkLoadCalculator {

  // ...

  int calculateTotalLoad() {
    // ... somewhere read the flags' global state
    String algorithm =
        ConfigFlags.FLAG_loadAlgorithm.get();
    // ...
  }
}
```

After: Testable and Flexible Design

```
// The new test is easier to understand and less
//   likely to break other tests.

class NetworkLoadCalculatorTest {

  public void testMaximumAlgorithmReturnsHighestLoad()
{
    NetworkLoadCalculator calc =
        new NetworkLoadCalculator("maximum");
    calc.setLoadSources(10, 5, 0);
    assertEquals(10, calc.calculateTotalLoad());
  }
}

  // Replace the global dependency on the Flags with
  // the Guice FlagBinder that gives named annotations
  // to flags automatically. String Flag_xxx is bound
  // to String.class annotated with @Named("xxx").
  // (All flag types are bound, not just String.)
  // In your Module:

    new FlagBinder(binder()).bind(ConfigFlags.class);

// Replace all the old calls where you read Flags with
//   injected values.
class NetworkLoadCalculator {

  String loadAlgorithm;

  // Pass in flag value into the constructor
  NetworkLoadCalculator(
      @Named("loadAlgorithm") String loadAlgorithm) {
    //... use the String value however you want,
    //and for tests, construct different
    //NetworkLoadCalculator objects with other values.
    this.loadAlgorithm = loadAlgorithm;
  }

  // ...
}
```

Also Known As: *That Gnarley Smell You Get When Calling* Flags.disableStateCheckingForTest() *and* Flags.enableStateCheckingForTest().

Flag classes with static fields are recognized as a way to share settings determined at application start time (as well as share global state). Like all global state, though, they come with a heavy cost. Flags have a serious flaw. Because they share global state, they need to be very carefully adjusted before and after tests. (Otherwise subsequent tests might fail).

- Flaw: One test can set a flag value and then forget to reset it, causing subsequent tests to fail.
- Flaw: If two tests need different values of a certain flag to run, you cannot parallelize them. If you tried to, there would be a race condition on which thread sets the flags value, and the other thread's tests would fail.

- Flaw: The code that needs flags is brittle, and consumers of it don't know by looking at the API if flags are used or not. The API is lying to you.

To remedy these problems, turn to our friend Dependency Injection. You can use Guice to discover and make injectable all flags in any given classes. Then you can automatically inject the flag *values* that are needed, without ever referencing the static flag variables. Because you're working with regular java objects (not Flags) there is no longer a need to call Flags.disableStateCheckingForTest() or Flags.enableStateCheckingForTest().

Problem: Static Initialization Blocks Can Lock You Out of Desired Behavior

Before: Hard to Test

```
// Static init block makes a forced dependency
//   and concrete Backends are instantiated.

class RpcClient {

  static Backend backend;

  // static init block gets run ONCE, and whatever
  //   flag is read will be stuck forever.
  static {
    if (FLAG_useRealBackend.get()) {
      backend = new RealBackend();
    } else {
      backend = new DummyBackend();
    }
  }

  static RpcClient client = new RpcClient();

  public RpcClient getInstance() {
    return client;
  }

  // ...

}

class RpcCache {

  RpcCache(RpcClient client) {
    // ...
  }

  // ...
}
```

```
// Two tests which fail in the current ordering.
//   However they pass if run in reverse order.

@LargeTest
class RpcClientTest extends TestCase {

  public void testXyzWithRealBackend() {
    FLAG_useRealBackend.set(true);
    RpcClient client = RpcClient.getInstance();

    // ... make assertions that need a real backend
    // and then clean up

    FLAG_useRealBackend.resetForTest();
  }
}

@SmallTest
class RpcCacheTest extends TestCase {

  public void testCacheWithDummyBackend() {
    FLAG_useRealBackend.set(false);
    RpcCache cache =
        new RpcCache(RpcClient.getInstance());

    // ... make assertions
    // and then clean up
```

After: Testable and Flexible Design

```
// Guice managed dependencies, no static init block.

class RpcClient {

  Backend backend;

  @Inject
  RpcClient(Backend backend) {
    this.backend = backend;
  }

  // ...

}

class RpcCache {

  @Inject
  RpcCache(RpcClient client) {
    // ...
  }

  // ...
}
```

```
// These tests pass in any order, and if run in
//   parallel.

@LargeTest
class RpcClientTest extends TestCase {

  public void testXyzWithRealBackend() {

    RpcClient client =
        new RpcClient(new RealBackend());

    // ... make assertions that need a real backend
  }
}


@SmallTest
class RpcCacheTest extends TestCase {

  public void testCacheWithDummyBackend() {
    RpcCache cache = new RpcCache(
        new RpcClient(new DummyBackend()));

    // ... make assertions
 }
}
```

```
    FLAG_useRealBackend.resetForTest();
  }
}
```

```
// Configure Guice to manage RpcClient as a singleton
//   in the Module:
  bind(RpcClient.class).in(Scopes.SINGLETON);

// and use FlagBinder to bind the flags
  new FlagBinder(binder()).bind(FlagsClass.class);
```

Tests for classes exhibiting this problem may pass individually, but fail when run in a suite. They also might fail if the test ordering changes in the suite.

Depending on the order of execution for tests, whichever first causes RpcClient to load will cause FLAG_useRealBackend to be read, and the value permanently set. Future tests that may want to use a different backend can't, because statics enable global state to persist between setup and teardowns of tests.

If you work around this by exposing a setter for the RpcClient's Backend, you'll have the same problem as *"Problem: Need to call setForTest(…) and/or resetForTest() Methods,"* above. The underlying problem with statics won't be solved.

- Flaw: Static Initialization Blocks are run once, and are non-overridable by tests
- Flaw: The Backend is set once, and never can be altered for future tests. This may cause some tests to fail, depending on the ordering of the tests.

To remedy these problems, first remove the static state. Then inject into the RpcClient the Backend that it needs. Dependency Injection to the rescue. Again. Use Guice to manage the single instance of the RpcClient in the application's scope. Getting away from a JVM Singleton makes testing all around easier.

Problem: Static Method call in a Depended on Library

Before: Hard to Test

```
// Hard to test, since findNextTrain() will always
//   call the third party library's static method.

class TrainSchedules {
  // ...
  Schedule findNextTrain() {
    // ... do some work and get a track

    if (TrackStatusChecker.isClosed(track)) {
      // ...
    }

    // ... return a Schedule
  }
}
```

After: Testable and Flexible Design

```
// Wrap the library in an injectable object of your own.

class TrackStatusCheckerWrapper
    implements StatusChecker {

  // ... wrap and delegate to each of the
  //   3rd party library's methods

  boolean isClosed(Track track) {
    return TrackStatusChecker.isClosed(track);
  }
}

// Then in your class, take the new LibraryWrapper as
//   a dependency.

class TrainSchedules {
  StatusChecker wrappedLibrary;

  // Inject in the wrapped dependency, so you can
  //   test with a different test-double implementation.
  public TrainSchedules(StatusChecker wrappedLibrary) {
    this.wrappedLibrary = wrappedLibrary;
  }



  // ...
```

```
  Schedule findNextTrain() {
    // ...

    // Now delegate to the injected library.
    if (wrappedLibrary.isClosed(track)) {
      // ...
    }

    // ... return a Schedule
  }
}
```

```
// Testing something like this is difficult,
//   becuase the design is flawed.

class TrainSchedulesTest extends TestCase {
  public void testFindNextTrainNoClosings() {
    // ...
    assertNotNull(schedules.findNextTrain());
    // Phooey! This forces the slow
    //   TrackStatusChecker to get called,
    //   which I don't want in unit tests.
  }

}
```

```
// Testing this is trivial, because of DI.

class TrainSchedulesTest extends TestCase {
  public void testFindNextTrainNoClosings() {
    StatusCheckerWrapper localWrapper =
        new StubStatusCheckerWrapper();
    TrainSchedules schedules =
        new TrainSchedules(localWrapper);

    assertNotNull(schedules.findNextTrain());
    // Perfect! This works just as we wanted,
    //   allowing us to test the TrainSchedules in
    //   isolation.
  }
}
```

Sometimes you will be stuck with a static method in a library that you need to prevent from running in a test. But you need the library so you can't remove or replace it with a non-static implementation. Because it is a library, you don't have the control to remove the static modifier and make it an instance method.

- Flaw: You are forced to execute the TrackStatusChecker's method even when you don't want to, because it is locked in there with a static call.
- Flaw: Tests may be slower, and risk mutating global state through the static in the library.
- Flaw: Static methods are non-overridable and non-injectable.
- Flaw: Static methods remove a seam from your test code.

If you control the code (it is not a third party library), you want to fix the root problem and remove the static method.

If you can't change the external library, wrap it in a class that implements the same interface (or create your own interface). You can then inject a test-double for the wrapped library that has different test-specific behavior. This is a better design, as well as more easily testable. Often we find that testable code is higher quality, more easily maintained, and more productive-to-work-in code. Consider also that if you create your own interface, you may not need to support every method in a library class–just adapt the functionality you actually use.

Caveat: When is Global State OK?

There are two cases when global state is tolerable.

(1) When the *reference is a constant* and the object it points to is either *primitive or immutable*. So static final String URL = "http://google.com"; is OK since there is no way to mutate the value. Remember, the transitive closure of all objects you are pointing to must be immutable as well since globality is transitive. The String is safe, but replace it with a MyObject, and it gets be risky due to the

transitive closure of all state MyObject exposes. You are on thin ice if someone in the future decides to add mutable state to your immutable object and then your innocent code changes into a headache.

(2) When the *information only travels one way*. For example a Logger is one big singleton. However our application only writes to logger and never reads from it. More importantly our application does not behave differently based on what is or is not enabled in our logger. While it is not a problem from test point of view, it is a problem if you want to assert that your application does indeed log important messages. This is because there is no way for the test to replace the Logger with a test-double (I know we can set our own handler for log level, but those are just more of the problems shown above). If you want to test the logger then change the class to dependency inject in the Logger so that you can insert a MockLogger and assert that the correct things were written to the Logger. (As a convenience, Guice automatically knows how to Constructor Inject a configured logger for any class, just add it to the constructor's params and the right one will be passed in.)

Flaw: Class Does Too Much

The class has too many responsibilities. Interactions between responsibilities are buried within the class. You design yourself into a corner where it is difficult to alter one responsibility at a time. Tests do not have a clear seam between these interactions. *Construction of depended upon components (such as constructing object graph) is a responsibility that should be isolated from the class's real responsibility. Use dependency injection to pass in pre-configured objects. Extract classes with single responsibilities.*

Warning Signs:
• Summing up what the class does includes the word "and" • Class would be challenging for new team members to read and quickly "get it" • Class has fields that are only used in some methods • Class has static methods that only operate on parameters

Why this is a Flaw

When classes have a large span of responsibilities and activities, you end up with code that is:

- Hard to debug
- Hard to test
- Non-extensible system
- Difficult for Nooglers and hard to hand off
- Not subject to altering behavior via standard mechanisms: decorator, strategy, subclassing: you end up adding another conditional test
- Hard to give a name to describe what the class does. Whenever struggling with a name, the code is telling you that the responsibilities are muddled. When objects have a clear name, it is easy to keep them focused and shear off any excess baggage.

We do not want code like this in our codebase. And it is your responsibility as a CL Reviewer to request changes to be made in classes with too many responsibilities.

This Flaw is Also Described with These Phrases
None of the following terms are particularly flattering, yet often they come to accurately describe code that is taking on too many responsibilities. If not immediately, in time these classes grow into such descriptions:

- Kitchen Sink
- Dumping Ground
- Class who's Behavior has too many "AND's"
- First thing's KIll All The Managers (*See Shakespeare)

- God Class
- “You can look at anything except for this one class”

Recognizing the Flaw

Try to sum up what a class does in a single phrase. If this phrase includes “AND” it’s probably doing too much.

A class’ name should express what it does. The name should clearly communicate its role. If the class needs an umbrella name like Manager, Utility, or Context it is probably doing too much. When you can name something accurately and completely, you are better able to isolate responsibilities and create a focused class. When you can’t name it, it probably needs to be multiple objects.

- Example Troubling Description: “KillerAppServer contains main() and is responsible for parsing flags AND initializing filters chains and servlets AND mapping servlets for Google Servlet Engine AND controlling the server loop… “
- Example Troubling Description: “SyndicationManager caches syndications AND implements complex expiration logic AND performs RPCs to repopulate missing or expired entries AND keeps statistics about syndications per user.” (In reality, initializing collaborators may be a separate responsibility, independent from the work that actually happens once they are wired together.)

Warning signs your class Does Too Much: Constructor does Real Work

- Excessive scrolling
- Many fields
- Many methods
- Clumping of unrelated/related methods
- Many collaborators (passed in, constructed inside the class, accessed through statics, or yanked out of other collaborators)
- The class just “feels too large.” It’s hard to hold in your head at once and understand it as a single chunk of behavior. It would be hard for a Noogler to read this class and “just get it.”
- Class works with “dumb collaborators” that don’t have any behavior
- Hidden interactions behind the public methods

This class is working with “dumb collaborators” that don’t have their own behavior.
If the class you’re working with is doing all the work on behalf of the collaborators that interact with it, that responsibility belongs on the collaborators. People may call this a “god class” that assumes all the behavior that should be in other objects that interact with it. This is often caused by Primitive Obsession [*Refactoring*, Fowler].

Hidden interactions behind one public API, which could be addressed better through composition.

- From a design perspective, this hides a point of flexibility that composition would give us.
- Encapsulating the behavior for a single responsibility is usually good information hiding.

- Encapsulating the interaction between responsibilities (by having one class with many responsibilities) is usually bad. This makes the collaborations brittle.
- From a testing perspective, you don't get the seams that you want to test individual pieces in isolation.

Fixing the Flaw

If this CL tried to introduce a class that does too much, require the author to split it up

1. Identify the individual responsibilities.
2. Name each one crisply.
3. Extract functionality into a separate class for each responsibility.
4. One class may perform the hidden responsibility of mediating between the others.
5. Celebrate that now you can test each class in isolation much easier than before.

If working with a legacy class that did too much before this CL, and you can't fix the whole legacy problem today, you can at least:

1. Sprout a new class with the sole responsibility of the new functionality.
2. Extract a class where you are altering existing behavior. As you work on existing functionality, (i.e. adding another conditional) extract a class pulling along that responsibility. This will start to take chunks out of the legacy class, and you will be able to test each chunk in isolation (using Dependency Injection).

As you introduce these other collaborators, you may find the composition to be awkward or unnatural. Despite this awkwardness, you have to take steps today to prevent the large class from growing. If you don't it will only grow, gaining more and more extraneous responsibilities, and get worse.

If this CL has a class with fields that are only used in a few methods:
If you have a few methods that are the only clients of a certain field - there is your new class. Encapsulate the work these methods do into a new class. *If this CL has a static method that operates on parameters:*

Static methods are often a sign of a homeless method. Look at the parameters passed into the static method. You probably have a method that belongs on one of the parameters or a wrapper around one of the parameters. Move the method onto the parameter it belongs on.

Caveat: Living with the Flaw

Some legacy classes are "beyond the scope of this one CL." It may be unreasonable for a reviewer to demand a large, pre-existing problem be fixed in order to make a small change. But it is reasonable to draw a line in the sand and request that the author take steps in the right direction instead of making a bad situation worse. For example, sprout a new class instead of adding another responsibility to an existing, hard to test, object.