Use your singletons wisely

Know when to use singletons, and when to leave them behind

J.B. Rainsberger

July 01, 2001

Are singletons overused? Veteran programmer J. B. Rainsberger suggests they might be, explains why, and offers tips for knowing when to use singletons and when to seek more flexible alternatives.

The programming community discourages using global data and objects. Still, there are times when an application needs a single instance of a given class and a global point of access to that class. The general solution is the design pattern known as *singletons*. However, singletons are unnecessarily difficult to test and may make strong assumptions about the applications that will use them. In this article I discuss strategies for avoiding the singleton pattern for that majority of cases where it is not appropriate. I also describe the properties of some classes that are truly singletons.

Automated unit testing is most effective when:

- Coupling between classes is only as strong as it needs to be
- It is simple to use mock implementations of collaborating classes in place of production implementations

Certainly, when classes are loosely coupled, it is possible to concentrate on testing a single class independently. When classes are tightly coupled, it is possible only to test a group of classes together, making bugs more difficult to isolate. Generally speaking, testing is easiest when classes collaborate without assuming anything beyond their respective contracts.

Unit tests are meant to ensure that each class behaves as it claims: independently of the rest of the system. One common technique for making unit tests more effective, and making them run more quickly, is using mock objects in place of production implementations of collaborating objects. For example, in order to test how class `A` responds when class `B` throws an exception, it suffices to write something like the code in Listing 1.

Listing 1. Example code using mock objects

```
public class MyTestCase extends TestCase {
     ...
     public void testBThrowsException() {
         MockB b = new MockB();
         b.throwExceptionFromMethodC(NoSuchElementException.class);

         A a = new A(b);  // Pass in the mock version
         try {
             a.doSomethingThatCallsMethodC();
         }
         catch (NoSuchElementException success) {
             // Check exception parameters?
         }
     }
     ...
}
```

It is much simpler to simulate behavior than it is to recreate that behavior. Here, we simulate method `c()` throwing a `NoSuchElementException`, rather than recreate the scenario under which the production implementation of class B would do that. The simulation requires less specialized knowledge of class B than recreating the scenario would.

Singletons know too much

There is one implementation anti-pattern that flourishes in an application with too many singletons: the *I know where you live* anti-pattern. This occurs when, among collaborating classes, one class knows where to get instances of the other.

Where's the harm? Coupling among classes is vastly increased when classes know where to get instances of their collaborators. First, any change in how the supplier class is instantiated ripples into the client class. This violates the *Liskov Substitution Principle*, which states that you should allow any application the freedom to tell the client class to collaborate with any subclass of the supplier. This violation is felt by unit tests, but more importantly, it makes it difficult to enhance the supplier in a backward-compatible way. First, the unit tests cannot pass the client class a mock supplier instance for the purposes of simulating the supplier's behavior, as described above. Next, no one can enhance the supplier without changing the client code, which requires access to the supplier's source. Even when you have access to the supplier's source, do you really want to change all 178 clients of the supplier? Weren't you planning on having a nice, relaxing weekend?

Collaborating classes should be built to allow the application to decide how to wire them together. This increases the flexibility of the application and makes unit testing simpler, faster, and generally more effective. Remember that the easier it is to test a class, the more likely a developer will test it.

Moving away from singletons

Since singletons are not as desirable as you might first believe, I'll discuss how to code clients effectively so they don't know that their supplier is a singleton.

I'll elaborate on the problem. Whenever a client retrieves a singleton's instance, that client becomes unnecessarily coupled with the fact that its supplier is a singleton. As an example, consider a `Deployer` and a `Deployment`. The application only needs one `Deployer`, so we make it a singleton. Now we can code this method, as shown Listing 2.

Listing 2. Coding Deployer

```
public class Deployment {
     ...
     public void deploy(File targetFile) {
         Deployer.getInstance().deploy(this, targetFile);
     }
     ...
}
```

This looks like a good shortcut because the client can simply ask the `Deployment` to deploy itself -- the client is not responsible for knowing about `Deployer`s. Although this is true, the advantage is soon outweighed by the consequences when we now have to (or want to) use a different kind of `Deployer`. `Deployment` knows about the concrete class `Deployer`, so we cannot substitute a subclass of `Deployer` without changing the source for `Deployment`.

Rather than the `Deployment` knowing that `Deployer` is a singleton, clients should pass a `Deployer` instance to the `Deployment`'s constructor, as in Listing 3.

Listing 3. Passing Deployer to the Deployment's constructor

```
public class Deployment {
     private Deployer deployer;

     public Deployment(Deployer aDeployer) {
         deployer = aDeployer;
     }

     public void deploy(File targetFile) {
         deployer.deploy(this, targetFile);
     }
     ...
}
```

The two classes are less tightly coupled: There is now a simple association from `Deployment` to `Deployer`, rather than a reliance on the way that `Deployer`s are created. You let the application make that decision.

With the old code, the client must change; there can be no static methods on an interface. With the new code, `Deployment` need not change. Instead, `Deployment`'s client, the application, changes in a way that makes sense for the application. Moreover, if the application is well designed, one change will cause the behavior of all `Deployment` instances in the application to change along with it. So there's no problem next week -- or month, or year -- when you find out that `Deployer` needs to be an interface.

The unit tests also benefit. When it comes time to run multiple test cases, each test case may need a slightly different `Deployer`: Usually the `Deployer` needs to be in a certain state, in order to simulate a particular bit of its behavior. As we have already seen, the easiest way to achieve this is to create mock `Deployer` implementations. This test case code in Listing 4 illustrates this technique well.

Listing 4. Mock Deployer implementation

```
public class DeploymentTestCase extends TestCase {
     ...
     public void testTargetFileDoesNotExist() {
         MockDeployer deployer = new MockDeployer();
         deployer.doNotFindAnyFiles();

         try {
             Deployment deployment = new Deployment(deployer);
             deployment.deploy(new File("validLocation"));
         }
         catch (FileNotFoundException success) {
         }
     }
     ...
}
```

Here, we tell our mock `Deployer`, "Do not find any files, no matter what file object I pass you." We use this technique to simulate the case where the client attempts to deploy to a file in a nonexistent folder, for example. You may wonder why you shouldn't just specify a silly file name to actually create the exception condition, rather than simulate it. We're interested only in simulating the exception condition, and not re-creating it -- doing the latter more easily leads to confusion. Suppose a novice developer on your team chooses `d:/doesNotExist` as the test case's invalid file location. She passes the tests on her machine and integrates her changes. Now you run the tests on your machine, which just happens to have `d:/doesNotExist` on the filesystem: Maybe it's left behind from a test case of yours that didn't tear itself down correctly. The test unexpectedly fails, not because the code is wrong, but because the test depends too much on the environment around it.

This wastes time. You spend 30 minutes isolating the cause of the problem, 15 minutes explaining to the novice developer why `d:/doesNotExist` was a risky choice, and 20 minutes crafting a note to the rest of the team, warning them against such coding practices. Of course, when someone new joins the team, it will probably happen again. Writing a single mock deployer with a method called `doNotFindAnyFiles` allows you to avoid such annoyances.

Aggregating singletons: the Toolbox

Singleton abuse can be avoided by looking at the problem from a different angle. Suppose an application needs only one instance of a class and the application configures that class at startup: Why should the class itself be responsible for being a singleton? It seems quite logical for the application to take on this responsibility, since the application requires this kind of behavior. The application, not the component, should be the singleton. The application then makes an instance of the component available for any application-specific code to use. When an application uses several such components, it can aggregate them into what we have called a toolbox.

Put simply, the application's toolbox is a singleton that is responsible either for configuring itself or for allowing the application's startup mechanism to configure it. The general pattern of the `Toolbox` singleton is as shown here:

Listing 5. General pattern of the Toolbox singleton

```
public class MyApplicationToolbox {
     private static MyApplicationToolbox instance;

     public static MyApplicationToolbox getInstance() {
         if (instance == null) {
             instance = new MyApplicationToolbox();
         }
         return instance;
     }

     protected MyApplicationToolbox() {
         initialize();
     }

     protected void initialize() {
         // Your code here
     }

     private AnyComponent anyComponent;

     public AnyComponent getAnyComponent() {
         return anyComponent();
     }
     ...

     // Optional: standard extension allowing
     // runtime registration of global objects.
     private Map components;

     public Object getComponent(String componentName) {
         return components.get(componentName);
     }

     public void registerComponent(String componentName, Object component)
{
         components.put(componentName, component);
     }

     public void deregisterComponent(String componentName) {
         components.remove(componentName);
     }

}
```

The `Toolbox` is itself a singleton, and it manages the lifetime of the various component instances. Either the application configures it, or it asks the application for configuration information in method `initialize`. Now the application can decide how many instances of which classes it requires. Changes in those decisions may affect application-specific code, but not reusable, infrastructure-level code. Moreover, testing infrastructure code is much easier, as those classes do not rely on the way in which any application may choose to use them.

When it really is a singleton

To decide whether a class is truly a singleton, you must ask yourself some questions.

- Will every application use this class exactly the same way? (*exactly* is the key word)
- Will every application ever need only one instance of this class? (*ever* and *one* are the key words)
- Should the clients of this class be unaware of the application they are part of?

If you answer yes to all three questions, then you've found a singleton. The key points here are that a class is only a singleton if all applications treat it exactly the same and if its clients can use the class without an application context.

A classic example of a true singleton is a logging service. Suppose we have an event-based logging service: Client objects request that text be logged by sending a message to the logging service. Other objects actually log the text somewhere (console, file, whatever) by listening to the logging service for these logging requests and handling them. First, notice that the logging service passes the classic test for being a singleton:

- The requesters need a well-known object to which to send requests to log. This means a global point of access.
- Since the logging service is a single event source to which multiple listeners can register, there only needs to be one instance.

The classic singleton design pattern requirements are met, but there's more:

- Although different applications may log to different output devices, the way they register their listeners is always the same. All customization is done through the listeners. Clients can request logging without knowing how or where the text will be logged. Every application would therefore use the logging service exactly the same way.
- Any application should be able to get away with only one instance of the logging service.
- Any object can be a logging requester, including reusable components, so that they should not be coupled to any particular application.

In addition to the classic requirements, our above requirements are also met by the logging service. We can safely implement the logging service as a singleton without concern that we may later regret our choice.

Despite these rules, you should consider letting the code tell you when a class should be a singleton. Once you identify an object that you can't quite figure out how to get your hands on, you'll ask yourself:

- Where am I going to get an instance of this class?
- Does this object belong to the application or the component I'm writing?
- Can I write this class so that customization can be pushed back to its clients?

If you've gotten this far, then perhaps it really is a singleton, but once the code tells you that it wants to use a subclass here, but not there, you need to reconsider your decision.

Don't worry: the code will always tell you what to do. Just listen.

Related topics

- A discussion on the Singleton Design Pattern. As always, don't be lured by hard-and-fast rules that use words like "always" and "never." On the same site, you'll find a discussion of the OnceAndOnlyOnce design principle, which takes as its starting point Kent Beck's suggestion that "[c]ode wants to be simple."
- Yahoo's Extreme Programming Group is full of discussions on how unit testing techniques affect design, usually for the better. This was my primary motivation for moving away from singletons where possible.
- The test cases in this article were implemented using the JUnit test framework. JUnit is lightweight, simple, and powerful, making it the ideal test framework for both small and big products.