Dependency Injection & Testable Objects

By Jeremy Weiskotten, April 11, 2006

Post a Comment

Source Code Accompanies This Article. Download It Now.

- depend.txt
- depend.zip

Dependency injection is a pattern for designing loosely coupled and testable objects.

Jeremy is a senior software engineer for Kronos. He can be contacted at jeremy@weiskotten.com.

You can easily create reusable objects through encapsulation. However, even the most cohesive, well-encapsulated objects can be tightly coupled with other well-encapsulated objects. Tight coupling results in a tangled web of dependencies composed of objects that are fragile, difficult to reuse, and difficult to unit test. Applications with loosely coupled components, on the other hand, are modular, flexible, and easily tested with unit tests. Dependency injection, the topic I examine in this article, is a simple pattern to loosely couple objects and their dependencies.

It's easy to understand objects that have no dependencies on other objects. You can look at the code, see how its public methods work, fix bugs, and enhance its behavior. But what if another object uses the one you've just changed? Did you know about it and did your change break it? Are you prepared to make changes to that consumer, then make changes to its consumers? It doesn't take long before the problem snowballs. Tightly coupled objects cannot be changed easily without consequences. When there is no explicit contract that defines the relationship between objects, it's often difficult to predict what effect a change in one object will have on its consumers.

Coupling is transitive. If *A* is coupled to *B* and *B* is coupled to *C*, then *A* is effectively coupled to *C*. Objects with tightly coupled dependencies cannot be reused without lugging around the baggage of those dependencies. Dependencies can be expensive, often performing time-consuming procedures such as getting data from a database, and they may have expensive dependencies of their own. Objects that have heavy, expensive dependencies can't easily be reused in lightweight environments like unit tests.

Unit tests are "white box" tests that exercise objects (given intimate knowledge of their design). Unit testing exercises an object in isolation and is not designed to measure how an object interacts with its dependencies. You are much more likely to run unit tests continuously if they run quickly. Unit tests should run in seconds, not minutes, which allows for short feedback cycles when writing new code or refactoring to improve the design.

A unit test is a user of the object it tests. An object must exist for a reason, so there must be other objects that also use it. Even when only one application object uses it, the fact that the unit test is also a user means it has at least two users, and therefore the object must be reusable by design. Test-driven development (TDD), in which unit tests are written before the objects they test, is gaining as a development best practice. One beneficial consequence of TDD is that objects are designed with testability in mind, and therefore, an object and its dependencies must be loosely coupled and reusable. Dependency injection is a popular pattern to design loosely

coupled—and therefore robust, reusable, and testable—objects.

Interfaces as Contracts

Modern object-oriented languages such as Java and C# provide a way to define interfaces. An interface defines the explicit contract between an object that implements the interface (a service) and an object that depends on it (a consumer). In this article, I use Java and JUnit, along with common Java terms (like "extend" and "implement"). The C++ equivalent of a Java interface is a class that contains only public, pure virtual methods, and a C++ class that implements an interface actually inherits from such a class.

An interface defines the methods that must be exposed by a service for use by its consumers. There may be any number of different implementations of an interface, each of which provides the same service (through public methods), but may be designed and implemented differently—for example, a "stub" implementation holds the fort until real implementations are designed, developed, or provided, while a "mock" implementation simulates the behavior of a production implementation for lightweight unit-testing purposes.

Both stubs and mocks simulate the behavior of production implementations to test components as early as possible without their dependencies, and the mechanics are often identical. The difference between them is subtle, but important. Think of it this way: Stubs are temporary placeholders and don't add significant value to your project (other than letting you defer design decisions), while mocks add permanent value when applied to unit tests.

Coding to well-defined interfaces, particularly when using the dependency injection pattern, is the key to achieving loose coupling. By coupling an object to an interface instead of a specific implementation, you have the ability to use any implementation with minimal change and risk.

An Example

The *Carpenter* interface in Example 1 defines the contract for a carpenter. The interface contains one method, *getShelter*, which returns a *Shelter*. The interface doesn't expose or describe how *Carpenter* works, only that *Carpenter* agrees to provide a particular service. It's up to the implementation to determine whether to use hand tools or power tools, whether to do the job alone or hire a crew of 50, whether to create a new shelter from scratch or to purchase a fixer-upper, or whether to build a palace or a shack.

```
1  public interface Carpenter {
2      Shelter getShelter();
3  }
```

Example 1: Carpenter interface.

Example 2 is an implementation of the *Carpenter* interface, *TentCarpenter*, which creates and returns a new *Tent* instance from *getShelter* (assuming that *Tent* is a concrete class that implements the *Shelter* interface).

```
1  public class TentCarpenter implements Carpenter {
2     public Shelter getShelter() {
3         return new Tent(); // class Tent implements Shelter
4      }
5  }
```

Example 2: TentCarpenter implementation.

A person relocating to a new home is encapsulated by the class *Relocater* (Example 3). A*Relocater* depends on a *Carpenter* to provide a shelter to move into. In this example, the*Relocater* and *Carpenter* are already coupled fairly loosely because the object is accessed only through methods defined by the interface. The only time the actual implementation is specified is when the *Relocater* is constructed and a *TentCarpenter* is created.

```
1  public class Relocater {
2      private final Carpenter carpenter;
3      public Relocater() {
4          carpenter = new TentCarpenter();
5      }
6      public Shelter moveIntoNewDigs() {
7          Shelter digs = carpenter.getShelter();
8          return digs;
9      }
10 }
```

Example 3: Loosely coupled by coding to an interface.

The *Relocater* class is immutable—once constructed, its state never changes because the field carpenter is declared final and initialized in the constructor. It's a good practice to use immutable objects whenever possible, because they're inherently thread safe.

Refactoring for Testability

Every instance of *Relocater* uses the *TentCarpenter* implementation, so there is no convenient way to unit test *Relocater* in isolation with the current design. *Relocater*'s *getShelter* method always gets a *Tent* from a *TentCarpenter*, but you want to test *Relocater*, not *TentCarpenter* or*Tent*. (Of course, you've probably already written unit tests that exercise *TentCarpenter* and*Tent*, and you could want to write an integration test that exercises the relationship between*Relocater* and *TentCarpenter*, but there's not much point in doing that until you know for sure that *Relocater* works in isolation.) If *Relocater* and its *Carpenter* dependency were more loosely coupled, you could use a lightweight mock implementation of *Carpenter* for unit testing.

There are a few ways to refactor this class to make it more flexible and amenable to testing:

- Extend and Construct Mock. A unit test extends *Relocater* and provides a new constructor that creates a mock *Carpenter*. However, the field "*carpenter*" is declared private, so you need to add a protected method named *setCarpenter* to *Relocater* to let you set the*carpenter* from a subclass. The field *carpenter* was originally declared final to make it immutable. This new strategy breaks the immutability, so the final keyword must be removed. This is pretty major surgery (with some nasty side effects like losing immutability) just to make the object unit testable!
- Extend and Override Getter. Add a method named *getCarpenter* to *Relocater* that returns the field *carpenter*, and change any code that directly accesses the field to call the new method. A unit test extends *Relocater* and overrides *getCarpenter*, making it return a mock*Carpenter*. The *carpenter* field can be lazily initialized in *getCarpenter*. This strategy isn't too horrible—it preserves the object's immutability, but you still need to subclass *Relocater* for unit testing.
- Dependency Injection. Add a constructor to *Relocater* with a *Carpenter* parameter. The constructor sets the *carpenter* field to the instance passed into the constructor. This constructor provides a way for the unit test to provide (inject) its mock implementation.

Example 4 shows the constructor that has been refactored for dependency injection.

```
1  public Relocater(Carpenter carpenter) {
2          this.carpenter = carpenter;
3  }
```

Example 4: Refactored to dependency injection.

The "Extend" options get the job done, but at a cost. At a minimum, the cost is complexity that obscures the original design, making the code harder to understand and maintain in the future. In addition, the unit tests are more complex and therefore harder to write and understand.

Using dependency injection, the object is much more flexible without adding a significant amount of complexity or sacrificing any design principles (such as immutability). It's also easy to test, with no need to write a subclass (see Example 5).

```
1  public class RelocaterTest extends TestCase {
2      public void testMoveIntoNewDigs() {
3          Relocater relocater =
4              new Relocater(new MockCarpenter());
5          Shelter shelter = relocater.moveIntoNewDigs();
6          assertNotNull(shelter);
7      }
8  }
```

Example 5: Unit testing with a mock.

An object's consumer—whoever has a dependency on the object—should rarely be required to explicitly inject dependencies. Not only would this result in more (and more complicated) code on the consumer's end, it violates the principle of encapsulation (by exposing an implementation detail) and can result in tightly coupled code. In most cases, a component should be ignorant of its dependencies' dependencies. Therefore, you should consider a few additional design options when refactoring to dependency injection:

- Default Constructor Delegation. The default constructor might delegate to the dependency-injecting constructor, passing in an instance of the default *Carpenter* implementation. Any other construction details would only need to be added to one constructor.
- Protected Constructor. Use default constructor delegation, but make the dependency-injecting constructor protected, so that it can still be called from a unit test that lives in the same package (but in a different directory) as the class. This prevents outsiders from invoking the constructor with an unknown *Carpenter* implementation. You might do this if you are only using dependency injection for testability, not modularity.
- Factory Method. Remove (or make private or protected) the default constructor, make the dependency-injecting constructor protected (so that it can still be called from a unit test), and add a static factory method that injects an instance of the default *Carpenter*implementation. Any consumers that currently instantiate the class using the new operator must be changed to call the factory method.

This strategy also makes it easy to use a lightweight dependency injection container in the future. If you decide to use a container like Spring, just gut the factory method and add the code to create the object using the Spring API.

Example 6 shows a factory method that uses dependency injection to assemble a *Relocater*and its *Carpenter*.

```
public static Relocater getInstance() {
    Carpenter carpenter = new TentCarpenter();
    return new Relocater(carpenter);
}
```

Example 6: Injecting the dependency in a factory.

Even after you refactor to use a factory method, the JUnit test from Example 5 should still pass.

Dependency Injection Containers

An object and its dependencies can be constructed and assembled using a lightweight dependency-injection container such as Spring (http://www.springframework.org/), PicoContainer (http://www.picocontainer.org/), and HiveMind (http://jakarta.apache.org/hivemind/). These open-source containers take care of the construction, injection, and lifecycle-management details for your objects and their dependencies.

Spring and HiveMind also let you define the relationships between objects in external XML configuration files, which results in modular systems. The containers construct and inject the appropriate dependencies at runtime. If you want or need to swap one implementation of a dependency for another, you simply modify the configuration and restart the application—no recompiling or redeploying!

Example 7 is a Spring configuration file, beans.xml, that associates *Relocater* to an implementation of its *Carpenter* dependency. In Example 8, the factory method has been changed to use the Spring BeanFactory API to create the bean.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC
    "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="relocater" class=
            "com.ddj.dependencyinjection.Relocater">
        <constructor-arg><ref bean=
            "carpenter"/></constructor-arg>
    </bean>
    <bean id="carpenter" class="com.ddj.dependencyinjection.TentCarpenter"/>
</beans>
```

Example 7: Spring beans.xml configuration file.

```
public static Relocater create() {
    ClassPathResource res =
        new ClassPathResource("beans.xml");
    XmlBeanFactory factory = new XmlBeanFactory(res);
    return (Relocater)factory.getBean("relocater");
}
```

Example 8: Factory method that uses the Spring API to create the object.

Setter Injection

The examples so far have demonstrated constructor injection, where dependencies are injected through the object's constructor. Another common dependency injection pattern is setter injection.

You might prefer, at least in certain circumstances, setter injection, where dependencies are injected through setter methods, or mutators, into an object that has already been constructed. Example 9 demonstrates setter injection from a factory method.

```
1  public static Relocater getInstance() {
2      Relocater relocater = new Relocater();
3      relocater.setCarpenter(new TentCarpenter());
4      return relocater;
5  }
```

Example 9: Factory method that uses setter injection.

Setter injection has some benefits over constructor injection:

- An object's dependencies can be changed at runtime.
- Constructors with lots of parameters, especially when some parameters are of the same type, can be difficult to understand and use properly. Setter methods are a more self-descriptive option.
- Constructors are not inheritable in Java but setters are. A component extending a class that uses constructor injection must explicitly declare each constructor and forward to the superclass's implementation. Setter methods, on the other hand, are inherited.

The primary detriment is that an object with setters cannot be immutable, and immutability has many benefits. (You might design semimutable objects that have mutable attributes but immutable dependencies.) Because I prefer to write immutable objects whenever possible, I prefer constructor injection, but it's a design decision that may differ from one object to the next. Some developers prefer constructor injection in new objects but may refactor to setter injection if the constructor becomes unwieldy.

Setter and constructor injection are equally supported by several open-source dependency injection containers.

DDJ