4 Building modular applications

This chapter covers:

- Organizing code in modules
- Watching out for tight coupling
- Designing with loose coupling
- Testing code in discrete units
- Working with key rebinding

"To define it rudely but not ineptly, engineering is the art of doing that well with one dollar, which any bungler can do with two after a fashion."

—Arthur Wellesley

So far, we've looked at what it means for objects to be well behaved and classes to be well designed in the small. But in any real-world project, there's also a big picture to be considered. Just as there are principles that help our design of objects in the micro, there are principles for good design in the macro sense. In the coming sections, we'll talk about these design principles and see how to relate them to dependency injection.

Chief among these is testing. DI facilitates *testing* and *testable* code, the latter being an extremely important and often-overlooked condition to successful development. Closely tied to this is the concept of *coupling*, where good classes are linked to the dependencies in ways that facilitate easy replacement and therefore testing—bad ones are not. We'll see how to avoid such cases, and finally we'll look at the advanced case of modifying injector configuration in a running application.

First, let's start at in the micro level and explore the role objects play as the *construction units* of applications.

4.1 Understanding the role of an object

We're all very familiar with objects; you work with them every day and use them to model problems, effortlessly. But let's say for a moment you were asked to define what an object is—what might you say?

You might say an object is:

- A logical grouping of data and related operations
- An instance of a class of things
- A component with specific responsibilities

An object is all these things, but it is also a building block for programs. And as such, the design of objects is paramount to the design of programs themselves. Classes that have a specific, well-defined purpose and stay within clearly defined boundaries are well behaved and reliable. Classes that grow organically, with functionality bolted on when and where required, lead to a world of hurt.

A class can itself be a member of a larger unit of collective responsibilities called a *module*. A module is an independent, contractually sound unit that is focused on a broad part of business responsibility. For example, a *persistence* module may be responsible for storing and retrieving data from a database.

A module may not necessarily be meant for business functionality alone. For example, a *security* module is responsible for guarding unwarranted access to parts of an application. Modules may also be focused on infrastructure or on application logic but typically not both. In other words, a module is:

- *Whole*—A module is a *complete unit* of responsibility. With respect to an application, this means that modules can be picked up and dropped in as needed.
- *Independent*—Unlike an object, a module does not have dependencies on other modules to perform its core function. Apart from some common libraries, a module can be developed and tested independently (that is, in an isolated environment).
- *Contractually sound*—A module conforms to well-defined behavior and can be relied on to behave as expected under all circumstances.
- *Separate*—A module is not invasive of collaborators, and thus it is a discrete unit of functionality.

These qualities of a module are largely important in relation to its collaborators. Many modules interacting with one another through established, patent boundaries form a healthy application. Since modules may be contributed by several different parties (perhaps different teams, sister projects, or even external vendors), it's crucial that they follow these principles. Swapping in replacement modules, for example, replacing persistence in a database with a module that provides persistence in a data cluster or replacing a web presentation module with a desktop GUI, ought to be possible with a minimum of fuss. So long as the overall purpose of the application is maintained, a system of well-designed modules is tolerant to change. Much as different incarnations of an object graph provide variant implementations of a service, different assemblies of modules provide different application semantics. A module is thus an independent, atomic unit of reuse.

Objects and modules that collaborate typically have strong relationships with each other. Often the design of a dependency is influenced by its collaborators. However, each object has its area of responsibility, and well-designed objects stick to their areas without intruding on their collaborators. In other words, each object has its area of concern. Good design keeps those concerns separated.

4.2 Separation of concerns (my pants are too tight!)

Tight pants are very cumbersome! If, like me, you live in a sweltering subtropical environment, they can be especially discomfiting. Similarly, tightly coupled code can cause no end of development pain. It's worth taking the time at the head of a project to prevent such problems from creeping in later.

A big part of modular design is the idea that modules are deployable in different scenarios with little extra effort. The *separation* and *independence* of modules are core to this philosophy, which brings us to the tight-pants problem.

If we take a different perspective on the whole matter, it's not difficult to see that a module is itself a kind of object. The same principles that apply to module design and behavior also apply right down to objects themselves. In the next few sections we'll examine what it means to treat objects with the principle of separation. Earlier we laid down a similar principle: separating infrastructure from application logic so that logic that dealt with construction, organization, and bootstrapping was housed independently from core business logic. If you think of infrastructure as being orthogonal to application logic, then separating the two can be seen as dividing *horizontally* (see figure 4.1).

Similarly, logic dealing with different business areas can be separated *vertically* (see figure 4.2).

Checking spelling and editing text are two core parts of any email application. Both deal with application logic and are thus focused on business purpose. However, neither is especially related to the other. It

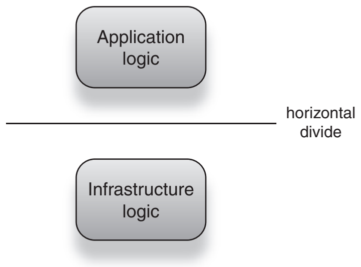

Figure 4.1 Conceptually, application logic sits *on top of* infrastructure logic.

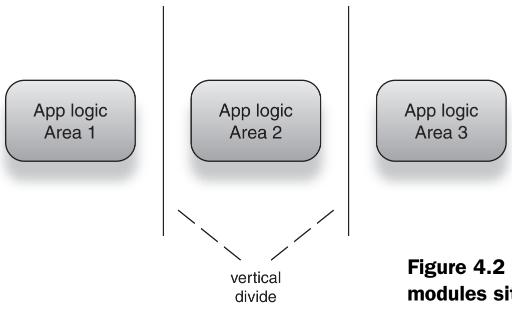

Figure 4.2 Application logic modules sit next to each other.

behooves us to separate them from each other just as we would separate any infrastructure code from the two of them. In other words, separating logic by area of concern is good practice. Figure 4.3 shows a group of modules, separated by area of application as well as infrastructure concern. This kind of modularity is indicative of healthy design.

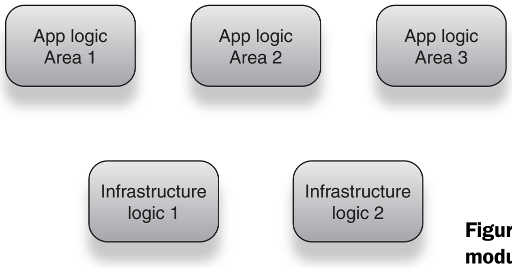

Figure 4.3 An assembly of discrete, separated modules along both infrastructure and core logic lines

When we lose this healthy separation between modules (and indeed objects), things start to get messy. Broadly speaking, this is tight coupling, and it has been a major headache for a very long time in OOP. Many language constructs were developed just to deal with this problem (interfaces and virtual methods, for instance). To know how to write healthy code, one must first be able to recognize tightly coupled code and understand *why* it is a problem.

4.2.1 Perils of tight coupling

In chapter 1, we looked at a classic example of tightly coupled code, declared that it was bad, tossed it away, and laid the foundation for DI. Let's resurrect that example and examine it closely (listing 4.1).

Listing 4.1 An email service that checks spelling in English

```
public class Emailer {
    private EnglishSpellChecker spellChecker;

    public Emailer() {
        this.spellChecker = new EnglishSpellChecker();
    }

    ...
}
```

This email service was poorly written because it encapsulated not only its dependencies but also the *creation* of its dependencies. In other words:

- It prevents any external agent (like the injector) from reaching its dependencies.
- It allows for only one *particular* structure of its object graph (created in its constructor).
- It is forced to know how to construct and assemble its dependencies.
- It is forced to know how to construct and assemble dependencies of its dependencies, and so on ad infinitum.

By preventing any external agent from reaching its dependencies, not only does it prevent an injector from creating and wiring them, it prevents a unit test from substituting *mock objects* in their place. This is an important fact because it means the class is *not testable*. Take the test case in listing 4.2 for instance:

Listing 4.2 A test case for `Emailer` using a mocked dependency

```
public class EmailerTest {

    @Test
    public final void ensureSpellingWasChecked() {
        MockSpellChecker mock = new MockSpellChecker();

        new Emailer(mock).send("Hello!");
        assert mock.verifyDidCheckSpelling()          <-- Mock tracks if
             : "failed to check spelling";                checkSpelling() was called
    }
}
```

And the mock spellchecker is as follows:

```
public class MockSpellChecker implements SpellChecker {
    private boolean didCheckSpelling = false;

    public boolean check(String text) {          <-- Mocked implementation
        didCheckSpelling = true;                     of interface method
        return true;
     }

    public boolean verifyDidCheckSpelling() {     <-- Mock-only method reports
        return didCheckSpelling;                      if spelling was checked
    }
}
```

This test case is impossible to write on `Emailer` as it exists in listing 4.1. That's because there is no constructor or setter available to pass in a mock `SpellChecker`. Mock objects are extremely useful:

- They allow you to test one class and none other. This means that any resultant errors are from that class and none other. In other words, they help you focus on discrete units.
- They allow you to replace computationally expensive dependencies (for instance, those that require hardware resources) with *fake* versions.

- They assert that the class follows the appropriate *contract* when speaking to its dependencies.
- They assert that everything happened as expected and in the expected *order*.

Even if we rewrote the original `Emailer` to take its dependency as an argument, we would still be stuck in this untenable position (see listing 4.3).

Listing 4.3 An email service that checks spelling in English, modified

```
public class Emailer {
    private EnglishSpellChecker spellChecker;

    public Emailer(EnglishSpellChecker spellChecker) {
        this.spellChecker = spellChecker;
    }

    ...
}
```

Now an external agent can set dependencies. However, the problem with testing remains, because `Emailer` is bound inextricably to an `EnglishSpellChecker`. Passing in a mock

```
MockSpellChecker mock = new MockSpellChecker();

new Emailer(mock);
```

results in a compilation error, because `MockSpellChecker` is not an `EnglishSpellChecker`. When a dependent is inextricably bound to its dependencies, code is no longer testable. This, in a nutshell, is tight coupling.

Being tightly coupled to a specific implementation, `Emailer` also prevents you from producing variant implementations. Take the following injector configuration, which tries to assemble an object graph consisting of an `Emailer` with an `EnglishSpellChecker` and a Latin character set in Spring XML (also see figure 4.4):

```
<beans ...>
    <bean id="emailer" class="Emailer">
        <constructor-arg>
            <bean class="EnglishSpellChecker">
                <constructor-arg>
                    <bean class="LatinCharset"/>
                </constructor-arg>
            </bean>
        </constructor-arg>
    </bean>
</beans>
```

Such a configuration is impossible given the tightly coupled object graph. By deciding its own dependencies, `Emailer` prevents any variant of them from being created. This makes for poor reuse and a potential explosion of similar code, since a new kind of `Emailer` would have to be written for each permutation.

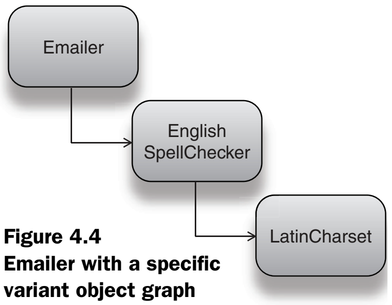

Figure 4.4 Emailer with a specific variant object graph

The example I used in chapter 1 is illustrative of the same problem—we were restricted to English spellcheckers, unable to provide variants in French or Japanese without a parallel deveMlopment effort. Tight coupling also means that we have opened the door in the iron wall separating infrastructure from application logic—something that should raise alarm bells on its own.

4.2.2 Refactoring impacts of tight coupling

We've built up quite a potent argument *against* coupling dependents to their dependencies. There's one major reason we can add that would outweigh all the others: the impact to coupled code when dependencies change. Consider the example in listing 4.4.

Listing 4.4 A utility for searching lists of strings

```
public class StringSearch {
     public String startsWith(ArrayList<String> list, String aString) {
        Iterator<String> iter = list.iterator();      <-- Needs only an iterator
        while(iter.hasNext()) {                           to do its work
            String current = iter.next();

            if (current.startsWith(aString))
                return current;
        }

        return null;
    }

    public boolean contains(ArrayList<String> list, String aString) {
        Iterator<String> iter = list.iterator();      <-- Needs only an iterator
        while(iter.hasNext()) {                           to do its work
            String current = iter.next();

            if (aString.equals(current))
                return true;
        }

        return false;
    }

    ...
}

//elsewhere
String startsWith = new StringSearch().startsWith(myList, "copern");

boolean contains = new StringSearch().contains(myList, "copernicus");
```

`StringSearch` provides a couple of utilities for searching a list of strings. The two methods I have in listing 4.5 test to see if the list contains a particular string and if it contains a partial match. You can imagine many more such utilities that comprehend a list in various ways. Now, what if I wanted to change `StringSearch` so it searched a `HashSet` instead of an `ArrayList`?

Listing 4.5 Impact of refactoring tightly coupled code

```
public class StringSearch {
     public String startsWith(HashSet<String> set, String aString) {
        Iterator<String> iter = set.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (current.startsWith(aString))
                return current;
        }

        return null;
    }

    public boolean contains(HashSet<String> set, String aString) {
        Iterator<String> iter = set.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (aString.equals(current))
                return true;
        }

        return false;
    }

    ...
}
```

Client code can now search over a HashSet instead but only after a significant number of changes to StringSearch (one per method):

```
String startsWith = new StringSearch().startsWith(mySet, "copern");

boolean contains = new StringSearch().contains(mySet, "copernicus");
```

Furthermore, you would have to make a change in every method that the implementation appeared in, and the class is no longer compatible with ArrayLists. Any clients already coded to use ArrayLists with StringSearch must be rewritten to use HashSets at a potentially an enormous refactoring cost, which may not even be appropriate in all cases. One way to solve this problem is by making the list a dependency, as shown in listing 4.6.

Listing 4.6 Refactor the list to a dependency

```
public class StringSearch {
    private final HashSet<String> set;

    public StringSearch(HashSet<String> set) {
        this.set = set;
    }

     public String startsWith(String aString) {
        Iterator<String> iter = set.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (current.startsWith(aString))
                return current;
        }
        return null;
    }

    public boolean contains(String aString) {
        Iterator<String> iter = set.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (aString.equals(current))
                return true;
        }

        return false;
    }

    ...
}
```

Consequently in client code, we can use one instance of the searching class to perform many actions on a set of strings:

```
StringSearch stringSearch = new StringSearch(mySet);

String startsWith = stringSearch.startsWith("copern");

boolean contains = stringSearch.contains("copernicus");
```

One problem is out of the way—we can now search `HashSet`s. And if you had a new requirement in the future to search, say, `TreeSet`s, then the refactoring impact would be much smaller (we'd need to change only the dependency and any clients). But this is still far from ideal. A change every time you choose a different collection data structure is excessive and untenable. We're still feeling the pangs of tight coupling. Listing 4.7 shows how *loose* coupling code cures this ailment.

Listing 4.7 Refactor to a loosely coupled dependency

```
public class StringSearch {
    private final Collection<String> collection;

    public StringSearch(Collection<String> collection) {
        this.collection = collection;
    }

     public String startsWith(String aString) {
        Iterator<String> iter = collection.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (current.startsWith(aString))
                return current;
        }

        return null;
    }

    public boolean contains(String aString) {
        Iterator<String> iter = collection.iterator();
        while(iter.hasNext()) {
            String current = iter.next();

            if (aString.equals(current))
                return true;
        }

        return false;
    }

    ...
}
```

By placing an abstraction between dependent and dependency (the interface `Collection`), the code loses any awareness of the underlying data structure and interacts with it only through an interface. All of these use cases now work without any coding changes:

```
boolean inList = new StringSearch(myList).contains("copern");

boolean inSet = new StringSearch(mySet).contains("copernicus");

boolean hasKey = new StringSearch(myMap.keySet()).contains("copernicus");

...
```

Now `StringSearch` is completely oblivious to the decisions of clients to use any kind of data structure (so long as they implement the `Collection` interface). You can even add new methods to `StringSearch` without impacting existing clients. This ability of the `Collection` interface to act as a contract for various services is extremely valuable, and you can mimic it in your own services to achieve a similar goal. Services that act according to a contract are well behaved. Going about this can be tricky, and we'll outline some best practices in the next section.

4.2.3 Programming to contract

Loose coupling via interfaces leads nicely to the idea of *programming to contract*. Recall that in chapter 1 we described a service as in part defined by a well-understood set of responsibilities called a contract. Programming to contract means that neither client nor service is aware of the other and communicates only via the contract. The only common ground is the understanding provided by this contract (interface). This means that either can evolve independently so long as both abide by the terms of the contract. A contract is also much more than a means of communication:

- It is a precise specification of behavior.
- It is a metaphor for *real-world* business purpose.
- It is a revelation of intent (of appropriate use).
- It is an inherent mode of documentation.
- It is the keystone for behavior *verification*.

The last point is extremely important because it means that a contract provides the means of testing classes. Verifying that an implementation works as it should is done by testing it against a contract. In a sense, a contract is a *functional requirements specification* for any implementation and one that is native to the programming language itself. Furthermore, a contract represents conceptual purpose and is thus akin to a *business contract.*

When ordering coffee at Starbucks, you follow a well-defined business process:

1. Stand in line.
2. Place your order.
3. Move to the edge of the counter to pick it up.

If Starbucks were to automate this process, the conceptual obligation between you as a client and the attendant would be a *programmatic* contract—probably not too different from listing 4.8.

Listing 4.8 The coffee shop contract!

```
public interface Customer {
    void waitInLine();

    boolean placeOrder(String choice);

    void pickup(Coffee prepared);
}

public interface Attendant {
    boolean takeOrder(String choice);

    Coffee prepareOrder();
}
```

Of course, there are plenty more steps in the real world (exchange of money, for one) but let's set them aside for now. What's important to note is that neither interface says anything about *how* to perform their function, merely *what* that function is. Well-behaved objects adhere strictly to their contracts, and accompanying unit tests help verify that they do. If any class were to violate the terms of its contract, the entire application would be at risk and most probably broken. If Starbucks were out of cappuccino and an order for one were placed, `Attendant.takeOrder()` would be expected to return `false`. If instead it returned `true` but failed to serve the order, this would break `Customers`.

Any class developed to the contract `Customer` or `Attendant` can be a drop-in replacement for either, while leaving the overall Starbucks system behaviorally consistent. Imagine a `BrazilianCustomer` served by a `RoboticAttendant` or perhaps a properly trained `ChimpanzeeCustomer` and `GorillaAttendant`.

Modeling conceptual business purpose goes much further. A carefully developed contract is the cornerstone for communication between business representatives and developers. Developers are clever, mathematical folk who are able to move quickly between abstractions and empiricism. But this is often difficult for representatives of the business who may be unfamiliar with software yet in turn contain all the knowledge about the problem that developers need. A contract can help clarify confusion and provide a bridge for common understanding between abstraction-favoring developers and empirical business folk. While you probably won't be pushing sheets of printed code under their noses, describing the system's interfaces and interoperation forms a reliable basis for communicating ideas around design.

A contract is also a *revelation of intent.* As such, the appropriate manner and context of a service's usage are revealed by its contract definition. This even extends to error handling and recovery. If instead of returning `false`, `Attendant.takeOrder()` threw an `OutOfCoffeeException`, the entire process would fail. Similarly, an `InputStream`'s methods throw `IOException` if there is an error because of hardware failure. `IOException` is explicitly declared on each relevant method and is thus part of the contract. Handling and recovering from such a failure are the responsibility of a client, perhaps by showing a friendly message or by trying the operation again.

Finally, a contract should be simple and readable, clear as to its purpose. Reading through a contract should tell you a lot about the classes that implement it. In this sense, it is an essential form of documentation. Look at this semantic interface from the `StringSearch` example shown earlier:

```
public interface StringSearch {
    boolean contains(String aString);

    String startsWith(String fragment);
}
```

This is a start, but there are several missing pieces. In the simple case of `contains()` we can infer that since it returns a `boolean`, it will return `true` if the string is found. However, what exactly does `startsWith()` return? At a guess, it returns a single successful match. But is this the first match? The last? And what does it return when there are no matches? Clearly, this is an inadequate specification to develop against. One further iteration is shown in listing 4.9.

Listing 4.9 `StringSearch` expressed as a contract

```
public interface StringSearch {

    /**
     * Tests if a string is contained within its list.
      *
     * @param aString Any string to look for.
     * @returns Returns true only if a matching string is found.
     */
    boolean contains(String aString);

    /**
     * Tests if a string in the list begins with a given sequence.
     *
     * @param fragment A partial string to search on.
     * @returns Returns the first match found or null if
     *     none were found.
     */
    String startsWith(String fragment);
}
```

Now StringSearch is clear about its behavior; startsWith() returns the first match or null if no matches are found. The implied benefit of this contractual programming is that code can be discretized into modules and units without their severely affecting one another when they change. DI is a natural fit in providing a framework for programming such loosely coupled code.

4.2.4 Loose coupling with dependency injection

So far, we've seen that loose coupling makes testing, reuse, and evolution of components easy. This makes for modules that are easy to build and maintain. Dependency injection helps by keeping classes relatively free of infrastructure code and by making it easy to assemble objects in various combinations. Because loosely coupled objects rely on contracts to collaborate with dependencies, it is also easy to plug in different implementations, via injection or mocks in a unit test. All it takes is a small change to the binding step. Keeping object graph structure described in one place also makes it easy to find and modify parts of it, with little impact to core application code.

Let's see this in practice. Let's say we have a book catalog, consisting of a library to search and locate books with. By following the principles of loose coupling, we've arrived at the interfaces shown in listing 4.10.

Listing 4.10 A book catalog and its library data service

```
public interface BookCatalog {
    Book search(String criteria);
}

public interface Library {
    Book findByTitle(String title);

    Book findByAuthor(String title);

    Book findByIsbn(String title);
}
```

BookCatalog refers only to the interface Library. Its search() method translates some free-form text search criteria into a search by title, author, or ISBN, subsequently calling the appropriate method on Library. In a simple case, the Library may be implemented as a file-based service, storing and retrieving Book records from disk. In bigger applications it will likely be an external database (like PostgreSQL[1] or Oracle). Altering the catalog system to use a database-backed variant of Library is as simple as changing Library's binding (see listing 4.11) to use the database-driven implementation.

Listing 4.11 A book catalog and a database-backed library service

```
public class BooksModule extends AbstractModule {

    @Override
    protected void configure() {
```

[1] PostgreSQL is an excellent, lightweight, and feature packed open source database. Find out more at http://www.postgresql.org.

```
        bind(Library.class).to(DatabaseLibrary.class);
        bind(BookCatalog.class).to(SimpleBookCatalog.class);
    }
}
```

So long as DatabaseLibrary correctly implements the Library interface, the program continues to work unhindered, and BookCatalog is none the wiser about its underlying storage mechanism. You don't have to compile it again. Listing 4.12 shows another change—this time of the catalog.

Listing 4.12 A desktop-GUI book catalog and a database-backed library service

```
public BooksModule extends AbstractModule {

    @Override
    protected void configure() {
        bind(Library.class).to(DatabaseLibrary.class);
        bind(BookCatalog.class).to(DesktopGUIBookCatalog.class);
    }
}
```

This time it's the Library that's oblivious to the goings on in our application. Loose coupling enables any of our services to evolve down their own paths and yet remain verifiable and behaviorally consistent (figure 4.5 visualizes this evolution), performing the intended service for end users.

Loose assembly of modules is extremely important to testing. It provides a tangible benefit since such code can be constructed easily in tests and given mock dependencies without any additional work. This last part is crucial to writing testable code. In the next section we will see why writing testable code is important and how it helps in developing better application designs in general.

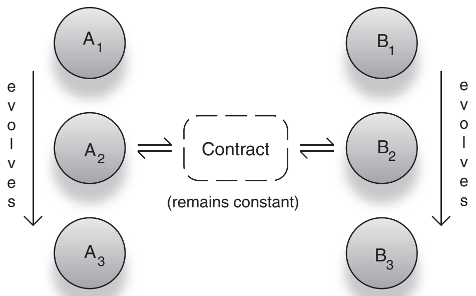

Figure 4.5 Independent evolution of A and B, with no impact to their collaboration

4.3 Testing components

One of the most important parts of software development is testing. Technology pundits argue that there is little value to code that has not been verified by some formal process. Without being able to assert expectations against an object's behavior, there's no way to tell if it fulfills its function. The fact that it compiles correctly, or runs in a deployment environment, is fairly useless in the broad sense. A functional test of an application (say by clicking through menu items in a desktop application) is a mite better than nothing at all, but not significantly so. It ignores several unseen failure points, such as bad data or concurrency bottlenecks.

While functional testing has its place in the quality assurance spectrum, the veracity of code simply can't be met without automated unit and integration tests. These tests provide the following:

- Unit-level assurance of functionality
- Simulation of disaster situations like failing hardware or resource depletion
- Detection of exceptional situations such as bugs in collaborating modules
- Detection of concurrency bottlenecks or thread-unsafe code
- Assurance of regression integrity[2]
- Benchmarking performance under extreme duress

Not only can you test a much broader spectrum of possible scenarios in code, but you can simulate disaster situations and measure performance under various loads. Automated tests are thus a vital part of any application, particularly one that is *programmed to contract*. Testing comes in different flavors, some or all of which can be automated by writing test code. Primary among these is the test accompanying an individual class or unit. This kind of test is independent of the rest of the application and intended to verify just the one unit. These tests are generally called *unit tests* but sometimes are also called *out-of-container* tests. Another form of testing asserts integration behavior between modules in a simulated environment.

4.3.1 Out-of-container (unit) testing

The idea of testing an individual unit of functionality (a class) raises questions about the surrounding environment. What about its dependencies? Its collaborators? Its clients? Testing a unit is *independent* of these surrounds. And this means independent of an injector too. As listing 4.13 shows, unit tests are not concerned with anything but the unit (single class) that they are testing.

Listing 4.13 A unit test of the book catalog, from listing 4.10

```
public class BookCatalogTest {

    @Test
    public final void freeFormSearch() {          <-- Test verifies search
        MockLibrary mock = new MockLibrary();

        new SimpleBookCatalog(mock)
             .search("dependency injection");

        assert mock.foundByKeyword();          <-- Verify mock after search
    }
}
```

We are quite happy to use construction by hand, in particular because there are very few dependencies (all of them mocked) and no dependents besides the test itself. It would be unnecessarily tedious to configure an injector just for a few mocks. If the test passes, there's some assurance that a correctly configured injector will work too, since the test uses normal language constructs. Furthermore, a *mocking framework* can make it a lot easier to create an object with mock dependencies.

[2] Regression integrity is the idea that previously verified behavior is maintained on new development iterations, so you don't undo the work that's already been done (and successfully tested).

NOTE EasyMock, Mockito, and JMock are such powerful mock objects frameworks for Java. I would highly recommend Mockito to start with and EasyMock for more sophisticated uses.

This kind of test tests services outside an injector and outside an application—hence the name out-of-container testing. A significant point about this kind of testing is that it forces you to write loosely coupled classes. Classes that rely specifically on injector behavior or on other parts of the application are difficult to test in this fashion because they rely on some dependency that only the final application environment can provide. A database connection is a good example. If it's outside an application, these dependencies are not readily available and must be simulated using mocks.

When you encounter classes whose dependencies cannot be mocked, you should probably rethink design choices. Older versions of EJB encouraged such container-dependent code. Infrastructure concerns like security, transactions, or logging often make things harder since their effects are not directly apparent. Developers tend to want to test everything as a whole and end up hacking together parts of a real application or environment. These don't do justice to testing, because they remove the focus on testing individual units of functionality. Errors raised by such tests may be misleading because of subtle differences in application environments. So when testing, try focusing your attention on a single unit or class, mocking out the rest. If you can do this individually for all units, you will have much more confidence in them and can easily assemble them into working modules.

4.3.2 I really need my dependencies!

If the class you're testing does little more than call into its dependencies, you might be tempted to use an injector—at least for the relevant module and mock the rest. It's not unusual to see tests like this:

```
public class BookCatalogTest {
    private Injector injector;

    @BeforeMethod
    public final void setup() {
        injector = Guice.createInjector(new TestBooksModule());
    }

    @Test
    public final void freeFormBookSearch() {
        new SimpleBookCatalog(injector.getInstance(Library.class))
             .search("..");

        ...
    }
}
```

This is a bad idea. Not only will you introduce unnecessary complexity in your tests (they now depend on an injector), but you'll also have to maintain ancillary code in the form of `TestBooksModule`. This is injector configuration that exists purely for the unit test and adds no real value. Furthermore, if there are errors in the injector configuration, the test will fail with a false negative. You may spend hours looking for the bug in the wrong place, wasting time and effort on maintaining code that adds very little value to the test case.

4.3.3 More on mocking dependencies

If your classes do nothing more than call into their dependencies, is it still worth writing unit tests for them? *Absolutely*. In fact, it is imperative if you want to assert your module's integrity in any meaningful way. Verifying the behavior of a service is only half the picture. A class's use of its dependencies is a critical part of its behavior. Mock objects can track and verify that these calls are according to expectation. There are many ways to do this. Let's look at one using EasyMock and writing a behavior script. Listing 4.14 shows such a script for `Library` and verifies its proper use by `BookCatalog`. Remember we're trying to verify that `BookCatalog` depends correctly on `Library`, in other words, that it *uses* `Library` properly.

Listing 4.14 A mock script and assertion of behavior

```
import static org.easymock.EasyMock.*;

public class BookCatalogTest {

    @Test
    public final void freeFormBookSearch() {
        Library mock = createStrictMock(Library.class);

        String criteria = "dependency injection";

        expect(mock.findByAuthor(criteria))       <-- First try author
            .andReturn(null);

        expect(mock.findByKeyword(criteria))      <-- Then try keyword
            .andReturn(null);

        Book di = new Book("dependency injection");
        expect(mock.findByTitle(criteria))        <-- Finally try title
            .andReturn(di);

        replay(mock);                  <--| Signal the mock
                                          | to be ready
        new SimpleBookCatalog(mock)
             .search(criteria);

        verify(mock);      <-- Verify its usage

    }
}
```

In listing 4.14, we script the `Library` mock to expect searches by author and keyword first on each and to return `null`, signaling that no such book exists. Finally, on title search, we return a hand-created instance of book that we're after. This not only asserts the correct use of `Library` by `SimpleBookCatalog` but also asserts correct *order* of use. If `SimpleBookCatalog` were written to prefer searching by title over keyword and author, this test would fail, alerting us to the mistake. To complete the test, we can add yet another assertion that tests the behavior of `SimpleBookCatalog`. Here's the relevant portion from listing 4.14, modified appropriately:

```
Book di = new Book("dependency injection");

 ...
Book result = new SimpleBookCatalog(mock).search(criteria);

assert di.equals(result) : "Unexpected result was returned";
```

Notice that we perform this assertion in addition to the assertions that the mocks provide. Now we have a complete test of `SimpleBookCatalog`. A more elaborate form of testing takes more than one unit into account. This helps detect problems in the coherency between units and determines whether the application behaves well in a broader context. This form of testing is known as integration testing.

4.3.4 Integration testing

So far we've shown how to test individual units and assert their behavior. This ought to be the first step in any development process. We've also lain down reasons for keeping tests independent of an injector or other environment-specific concerns. As your code matures, and the time arrives for a completed module to be integrated into the larger system, new testing concerns arise. These concerns have to do with the interoperation of modules, their dependence on external factors like hardware or database resources, and the overall coherence of the system. Automated tests can help in this regard too, and this flavor of tests is called integration tests.

Integration testing is about testing how various pieces fit together as a whole. Naturally, the dependency injector plays a vital part in this assembly by helping connect modules together in a transparent and cohesive fashion. Figure 4.6 illustrates this architecture.

As figure 4.6 shows, infrastructure concerns like persistence and security are encapsulated in their own modules and interfaced with logic in other modules. The dependency injector is responsible for pulling all of them together to form a coherent whole.

The idea behind integration testing is to simulate user interaction with a system or, if the application is not directly exposed to a human user, to simulate the next most relevant actor. This might be:

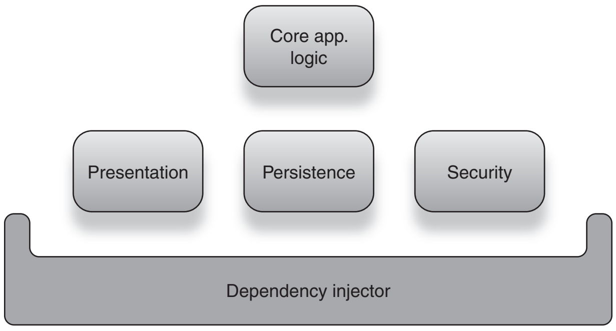

Figure 4.6 Architecture of modules brought together by dependency injection

- A batch of cleared bank checks
- Monitored cooling of a nuclear power plant
- Just about anything else you can imagine

Let's use the example of a web application that serves a page from a database. This will be an automated integration test (listing 4.15).

Listing 4.15 Integration test for a web application

```
public class HomePageIntegrationTest {
    private Injector injector;

    @BeforeTest
    public final void prepareContainer() {
        injector = Guice.createInjector(new WebModule(),
                     new TestPersistenceModule());      <-- Use a test environment for data resources
    }

    @AfterTest
    public final void cleanup() { .. }

    @Test
    public final void renderHomePage() throws Exception {
        HttpServletRequest request = createMock(..);
        HttpServletResponse response = createMock(..);

        injector.getInstance(HomeServlet.class)       <-- Get page servlet from injector and test
            .service(request, response);

        //assert something about response
    }
}
```

There are interesting things about the integration test in listing 4.15. Foremost among them, it looks like we've violated nearly every rule of unit testing. We're using an injector, not mocking anything apart from HTTP request (and response), and using a specialized configuration for tests. Of course, the simple explanation is that this isn't a unit test, so we can ignore these restrictions. But we can do better than that:

- Integration testing is about testing how modules interoperate, so we want to use actual production modules where possible.
- An accurate analog of the production environment is important, so we try to include as many of the variables as we can that go into it: infrastructure modules, the dependency injector, and any external services.
- There will probably be differences between a production and integration environment (such as which database instance to use), and these are captured in a parallel configuration, `TestPersistenceModule`.
- Integration tests are automated, so external interaction must be simulated (for example, this is done by mocking or simulating HTTP requests).

- Integration tests can be expensive, hence the `prepareContainer()` method that runs once before testing begins.
- They also require clean shutdowns of external services (such as database connections and timers), hence the presence of the `cleanup()` method, which runs after testing is complete.

When used properly, integration testing can help reveal errors in configuration and modular assembly. Typically these are environment-specific and often subtle. A healthy codebase will reveal very few coding errors in integration tests, since classes are covered by unit tests. More often than not, one finds that integration tests reveal configuration quirks in libraries or interoperability issues. These tests are an additional safeguard against design flaws in an architecture. However, they are neither the first line of defense (unit tests) nor the final word on application behavior (acceptance tests and QA). But they are a must-have tool to go in the toolbox of any developer.

Next up, we'll take a gander at integration from a number of perspectives. Modular applications can be integrated in various permutations to suit different goals, whether it be testing or deployment on weaker hardware such as a mobile device, where not all of the modules are required.

4.4 Different deployment profiles

We've shown how modules are composite globs of functionality that are like prefab units for big applications. As such, modular applications can be composed in different *profiles*, to different fits. A security module guarding HTTP access can be combined with a presentation module and a file-based persistence module to quickly create a web application. Architectures faithful to modularity principles will be able to evolve better, for instance, by replacing file-based persistence with a module that persists data in an RDBMS database.

While this sounds very simple, it's not always a plug-and-play scenario. Much of it depends on the flexibility of a module and its component classes. The more general and abstract (in other words, loosely coupled) a module's interface, the easier it will be to integrate. This generally comes from experience and natural evolution. Designing a module to be too abstract and all-purposed at the beginning is not a good idea either (and very rarely works).

Once you do have such an architecture, however, there are powerful uses to which it can be put. Chief among those is altering the deployment profile *dynamically*, for instance, by changing injector configuration at runtime. And this is what we call *rebinding*.

4.4.1 Rebinding dependencies

First of all, I must say that altering an injector configuration and the model of objects at runtime is potentially dangerous. Without careful library support, this can mean a loss of some validation and safety features that you would normally get at startup time. It can lead to unpredictable, erratic, and even undetectable bugs. However, if used with care, it can give your application very powerful dynamic features (such as *hot deployment* of changes). This is a fairly advanced use case, so convince yourself first that it is needed and that you can handle the requisite amount of testing and design.

Altering dependency bindings at runtime has several impacts. Injectors are there to give you a leg up, but they can't do everything, and putting them to unusual uses (like rebinding) is fraught with pitfalls:

- Once bound, a key provides instances of its related object until rebound.
- When you rebind a key, all objects already referencing the old binding retain the old instance(s).
- Rebinding is very closely tied to scope. A longer-lived object holding an instance of a key that has been rebound will need to be reinjected or discarded altogether.
- Rebinding is also tied to lifecycle. When significant dependencies are changed, relevant modules may need to be notified (more on this in chapter 7).
- Not all injectors support rebinding, but there are alternative design patterns in such cases.

Injectors that support rebinding are said to be mutable. This says nothing about field or object mutability (which is a different concept altogether). It refers purely to changing the association of a key to an object graph. PicoContainer is one such mutable injector. In the following section, we'll look at how to achieve mutability without a mutable injector, by using the well-known Adapter design pattern.

4.4.2 Mutability with the Adapter pattern

Here we will consider the case of an injector that does not support dynamic rebinding. The problem we're trying to solve is that there isn't enough knowledge while coding to bind all the dependencies appropriately. In other words, the structure of an object graph may change over the life of the application. One very simple solution is to maintain both object graphs and flip a switch when you need to move from one binding to another—something like this:

```
public class LongLived {
    private final DependencyA a;     |
    private final DependencyB b;     | Start with a
    private boolean useA = true;     |

    public LongLived(DependencyA a, DependencyB b) {
        this.a = a;
        this.b = b;
    }

    public void rebind() {
        useA = false;
    }

    public void go() {
        if (useA)      <— Work with a
            ...
        else
            ...     <— Or work with b
    }
}
```

Here the method `rebind()` controls which dependency `LongLived` uses. At some stage in its life, when the rebinding is called for, you need to make a call to `rebind()`.

This works—and probably quite well—but it seems verbose. What's more, it seems like there's a lot of infrastructure logic mixed in with our application. If we've learned anything so far, it's to avoid any such mixing. What's really needed is an abstraction, an intermediary to which we can move the rebinding logic and still remain totally transparent to clients. Providers and builders don't quite work because they either provide new instances of the *same* binding or provide an instance specific to some *context*. But adapters do. They are transparent to dependents (since an adapter extends its *adaptee*) and wrap any infrastructure, keeping it well hidden. Listing 4.16 demonstrates `LongLived` and its dependencies with the adapter pattern.

Listing 4.16 Dynamic rebinding with an adapter

```
public interface Dependency {
    int calculate();
}

public class DependencyAdapter implements Dependency, Rebindable {
    private final DependencyA a;
    private final DependencyB b;
    private boolean useA = true;

    @Inject
    public DependencyAdapter(DependencyA a, DependencyB b) {
        this.a = a;
        this.b = b;
    }

    public void rebind() {
        useA = false;
    }

    public int calculate() {
        if (useA)
            return a.calculate();

        return b.calculate();
    }
}
```

Now most of the infrastructure code has been moved to `DependencyAdapter`. When the rebinding occurs, flag `useA` is set to `false` and the adapter changes, now delegating calls to `DependencyB` instead. One interesting feature of this is the use of a `Rebindable` *role interface*. The control logic for dynamic rebinding is thus itself decoupled from the "rebinding" adapters. All it needs to do is maintain a list of `Rebindable`s, go through them, and signal each one to `rebind()` when appropriate. This is neat because the logic of deciding which binding to use is completely known at coding time.

Some injectors even allow *multicasting* to make this an atomic, global process. Multicasting (and lifecycle in general) is explored in depth in chapter 7. Most of all, the use of an adapter ensures our client code is lean and behaviorally focused. Here's what LongLived looks like now, after the changes from listing 4.16.

```
public class LongLived {
    private final Dependency dep;

    @Inject
    public LongLived(Dependency dep) {
        this.dep = dep;
    }

    public void go() {
        int result = dep.calculate();
        ...
    }
}
```

That's certainly more concise. Rebinding of the key associated with Dependency is now completely transparent to LongLived—and any other client code for that matter. This is especially important because it means that unit tests don't have to be rewritten to account for a change in infrastructure. This is a satisfying saving.

4.5 Summary

Objects are discrete functional units of data mixed with operations on that data. In the same sense, larger collections of objects and their responsibilities are known as *modules*. A module is typically a *separate* and *independent* compilation unit. It can be maintained, tested, and developed in isolation. So long as it fulfills the terms of its contract to collaborators, a module can be dropped into any well-designed architecture almost transparently.

Components that are invasive of other components or rely on specific implementation details are said to be *tightly coupled*. Tight coupling is detrimental to maintenance and readability. Moreover, it prevents reuse of utility-style components because they are tightly bound to concrete classes rather than abstractions such as interfaces. This reduces the overall modular integrity of a program. To avoid tight coupling, choose abstractions between a client and service. So long as each fulfills the terms of the mediary contract, either can evolve or be replaced with no impact to the other or to the system as a whole. This flows down to component granularity from the concept of modules. Contracts reveal many other things about a component and about design. These are:

- A business metaphor
- An essential form of (self-) documentation
- A revelation of intent
- A specification for building implementations
- A means of verifying behavior

Code that is modular is also easy to test. Testing specific modules (or components) is a vital part of their development, as it is the primary way to verify their correctness. Tests that rely on components to have dependencies wired are poor tests because they can easily confuse the issue when an error occurs. You may spend hours tracking down whether the component under test is responsible or if one of its dependencies caused the error. The use of mock objects is a powerful remedy to this predicament, and indeed it's an important way to narrow down and verify the behavioral correctness of a piece of code. Never allow injectors or other environment-specific frameworks to creep into unit tests, even as a crutch. Try to use mocks and test units in isolation as much as possible. Code that conforms to this focus is said to be tested out of container.

Once a module has been developed and tested, it is useful to test whether it properly fits together with the rest of the system. This too can be done by automation via integration testing. Integration tests try to be a close analog of the eventual production environment but with some obvious differences (simulated direct user interaction, lower-grade hardware, or external resources). Integration testing can be very useful in detecting bugs caused by the software or hardware environment and by configuration. These are often subtle and not caught easily in unit tests. In a healthy program, integration tests should rarely fail.

In rare cases, it is useful to alter injector configuration dynamically. This is analogous to reassembling the modular structure of the application, but done so at runtime. Not all DI libraries support this kind of functionality directly. One mitigant is to provide dependents with all possible dependencies, then force them to decide which to use as appropriate. This works, but it isn't a great solution because it pollutes application logic with infrastructure concerns. As an alternative solution, the adapter pattern works nicely. It encapsulates any infrastructure logic and can be used in place of the original dependency with no impact to client code. A change in the binding is signaled to the adapter(s) via a rebinding notification, and the adapter shifts to the new binding. While this is a robust and workable solution, it is fraught with potential pitfalls and should be weighed carefully before being embarked upon.

Modular code is wonderful for many reasons. As I've already mentioned, these include testing, reuse, and independent development in isolation. It allows many streams of development (and indeed teams of developers) to work on pieces of a very large composite application and keeps complexity to a minimum. Modularity fits very nicely into dependency injection. DI is not only able to wire and assemble modular code quickly, but it is also able to cast it in different profiles (structures) with minimal impact on collaborators. Thus, dependency injection empowers modular programming.