Spring Recipes: A Problem-Solution Approach

ISBN-13 (pbk): 978-1-59059-979-2

ISBN-10 (pbk): 1-59059-979-9

ISBN-13 (electronic): 978-1-4302-0624-8

ISBN-10 (electronic): 1-4302-0624-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Steve Anglin, Tom Welsh
Technical Reviewers: Sam Brannen, Kris Lander
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editor: Damon Larson
Associate Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor: Dina Quan
Proofreader: April Eddy
Indexer: Ron Strauss
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 1

Inversion of Control and Containers

In this chapter, you will learn the *Inversion of Control (IoC)* design principle, used by many modern containers to decouple dependencies between components. The Spring framework provides a powerful and extensible IoC container for you to manage your components. This container lies at the heart of the Spring framework and integrates closely with Spring's other modules. This chapter aims to give you the prerequisite knowledge you need to start using Spring quickly.

When talking about components in the Java EE platform, most developers will think of EJB (Enterprise JavaBeans). The EJB specification clearly defines the contract between EJB components and EJB containers. By running in an EJB container, EJB components can get the benefits of life cycle management, transaction management, and security services. However, in EJB versions prior to 3.0, a single EJB component requires a remote/local interface, a home interface, and a bean implementation class. These EJBs are called heavyweight components due to their complexity.

Moreover, in those EJB versions, an EJB component can only run within an EJB container and must look up other EJBs with JNDI (Java Naming and Directory Interface). So EJB components are technology dependent because they cannot be reused and tested outside the scope of an EJB container.

Many lightweight containers are designed to overcome the shortcomings of EJB. They are lightweight in that they can support simple Java objects as components. A challenge posed by lightweight containers is how to decouple dependencies between components. IoC has proved to be an effective solution to this problem.

While IoC is a general design principle, *Dependency Injection (DI)* is a concrete design pattern that embodies this principle. As DI is the most typical (if not the only) realization of IoC, the terms IoC and DI are often used interchangeably.

Upon finishing this chapter, you will be able to write a simple IoC container that is conceptually similar to the Spring IoC container. If you are already familiar with IoC, feel free to go directly to Chapter 2, which introduces the overall architecture and setup of the Spring framework.

1-1. Using a Container to Manage Your Components

Problem

The basic idea of object-oriented design is to break your system into a group of reusable objects. Without a central module to manage the objects, they have to create and manage their own dependencies. As a result, the objects are tightly coupled.

Solution

You need a *container* to manage the objects that make up your system. A container centralizes the creation of objects and acts as a registry to provide lookup services. A container is also responsible for managing the life cycle of objects and providing a platform for the objects to run on.

Objects running inside a container are called *components*. They must conform to the specification defined by the container.

How It Works

Separating Interface from Implementation

Suppose you are going to develop a system, one of whose functions is to generate different types of reports in either HTML or PDF format. According to the "separating interface from implementation" principle in object-oriented design, you should create a common interface for generating reports. Let's assume that the contents of the report are a table of records, appearing in the form of a two-dimensional string array.

```
package com.apress.springrecipes.report;

public interface ReportGenerator {

    public void generate(String[][] table);
}
```

Then you create two classes, `HtmlReportGenerator` and `PdfReportGenerator`, to implement this interface for HTML reports and PDF reports. For the purpose of this example, skeleton methods will suffice.

```
package com.apress.springrecipes.report;

public class HtmlReportGenerator implements ReportGenerator {

    public void generate(String[][] table) {
        System.out.println("Generating HTML report ...");
    }
}

package com.apress.springrecipes.report;

public class PdfReportGenerator implements ReportGenerator {
```

```
    public void generate(String[][] table) {
        System.out.println("Generating PDF report ...");
    }
}
```

The `println` statements in the method bodies will let you know when each method has been executed.

With the report generator classes ready, you can start creating the service class `ReportService`, which acts as a service provider for generating different types of reports. It provides methods such as `generateAnnualReport()`, `generateMonthlyReport()`, and `generateDailyReport()` for generating reports based on the statistics of different periods.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator = new PdfReportGenerator();

    public void generateAnnualReport(int year) {
        String[][] statistics = null;
        //
        // Gather statistics for the year ...
        //
        reportGenerator.generate(statistics);
    }

    public void generateMonthlyReport(int year, int month) {
        String[][] statistics = null;
        //
        // Gather statistics for the month ...
        //
        reportGenerator.generate(statistics);
    }

    public void generateDailyReport(int year, int month, int day) {
        String[][] statistics = null;
        //
        // Gather statistics for the day ...
        //
        reportGenerator.generate(statistics);
    }
}
```

As the report generation logic has already been implemented in the report generator classes, you can create an instance of either class as a private field and make a call to it whenever you need to generate a report. The output format of the reports depends on which report generator class is instantiated.

Figure 1-1 shows the UML class diagram for the current dependencies between `ReportService` and different `ReportGenerator` implementations.

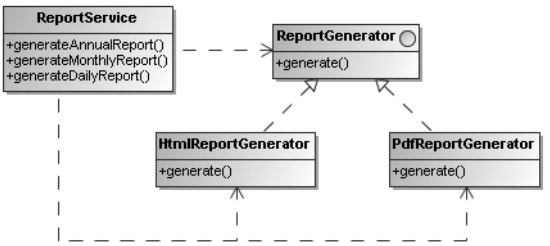

Figure 1-1. *Dependencies between ReportService and different ReportGenerator implementations*

For now, `ReportService` is creating the instance of `ReportGenerator` internally, so it has to be aware of which concrete class of `ReportGenerator` to use. This will cause a direct dependency from `ReportService` to either of the `ReportGenerator` implementations. Later you will be able to eliminate the dependency lines to the `ReportGenerator` implementations completely.

Employing a Container

Suppose that your report-generating system is designed for more than one organization to use. Some of the organizations may prefer HTML reports while the others may prefer PDF. You have to maintain two different versions of `ReportService` for different report formats. One creates an instance of `HtmlReportGenerator` while another creates an instance of `PdfReportGenerator`.

The cause of this inflexible design is that you have created the instance of `ReportGenerator` inside `ReportService` directly, so that it needs to know which `ReportGenerator` implementation to use. Do you remember the dependency lines from `ReportService` to `HtmlReportGenerator` and `PdfReportGenerator` in the class diagram (see Figure 1-1)? As a result, any switch of report generator implementation involves modification of `ReportService`.

To solve this problem, you need a container to manage the components that make up your system. A full-featured container would be extremely complex, but let's begin by having you create a very simple one:

```
package com.apress.springrecipes.report;
...
public class Container {

    // The global instance of this Container class for the components to locate.
    public static Container instance;

    // A map for storing the components with their IDs as the keys.
    private Map<String, Object> components;

    public Container() {
        components = new HashMap<String, Object>();
        instance = this;
```

```
        ReportGenerator reportGenerator = new PdfReportGenerator();
        components.put("reportGenerator", reportGenerator);

        ReportService reportService = new ReportService();
        components.put("reportService", reportService);
    }

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

In the preceding container example, a map is used to store the components with their IDs as the keys. The container initializes the components and puts them into the map in its constructor. At this moment, there are only two components, `ReportGenerator` and `ReportService`, working in your system. The `getComponent()` method is used to retrieve a component by its ID. Also note that the public static `instance` variable holds the global instance of this `Container` class. This is for the components to locate this container and look up other components.

With a container to manage your components, you can replace the `ReportGenerator` instance creation in `ReportService` with a component lookup statement.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator =
        (ReportGenerator) Container.instance.getComponent("reportGenerator");

    public void generateAnnualReport(int year) {
        ...
    }

    public void generateMonthlyReport(int year, int month) {
        ...
    }

    public void generateDailyReport(int year, int month, int day) {
        ...
    }
}
```

This modification means that `ReportService` doesn't have to worry about which `ReportGenerator` implementation to use, so you don't have to modify `ReportService` any more when you want to switch report generator implementation.

Now by looking up a report generator through the container, your `ReportService` is more reusable than before, because it has no direct dependency on either `ReportGenerator` implementation. You can configure and deploy different containers for different organizations without modifying the `ReportService` itself.

Figure 1-2 shows the UML class diagram after employing a container to manage your components.

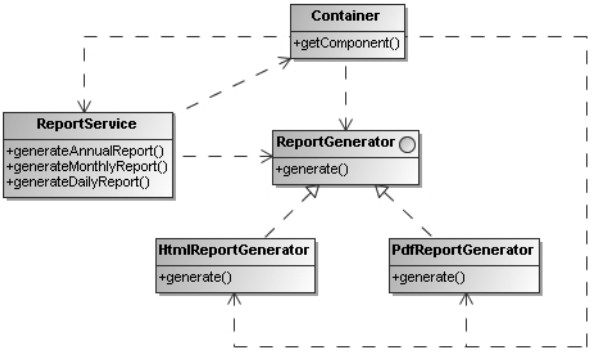

Figure 1-2. *Employing a container to manage your components*

The central class `Container` has dependencies on all the components under its management. Also note that the dependencies from `ReportService` to the two `ReportGenerator` implementations have been eliminated. Instead, a dependency line from `ReportService` to `Container` has been added, as it has to look up a report generator from `Container`.

Now you can write a `Main` class to test your container and components:

```
package com.apress.springrecipes.report;

public class Main {

    public static void main(String[] args) {
        Container container = new Container();
        ReportService reportService =
            (ReportService) container.getComponent("reportService");
        reportService.generateAnnualReport(2007);
    }
}
```

In the `main()` method, you first create a container instance and retrieve the `ReportService` component from it. Then when you call the `generateAnnualReport()` method on `ReportService`, `PdfReportGenerator` will handle the report generation request, as it has been specified by the container.

In conclusion, employing a container can help reduce coupling between different components within a system, and hence increase the independence and reusability of each component. In this way, you are actually separating configuration (e.g., which type of report generator to use) from programming logic (e.g., how to generate a report in PDF format) in order to promote overall system reusability. You can continue to enhance your container by reading a configuration file for component definition, which will be discussed later in this chapter.

1-2. Using a Service Locator to Reduce Lookup Complexity

Problem

Under a container's management, components depend on each other through their interfaces, not their implementations. However, they can only look up the container by using complex proprietary code.

Solution

To reduce the lookup complexity of your components, you can apply one of Sun's core Java EE design patterns, *Service Locator*. The idea behind this pattern is as simple as using a service locator to encapsulate the complex lookup logic, while exposing simple methods for lookup. Then, any component can delegate lookup requests to this service locator.

How It Works

Suppose you have to reuse the `ReportGenerator` and `ReportService` components in other containers with different lookup mechanisms, such as JNDI. For `ReportGenerator` there's no problem. But it would be trickier for `ReportService`, because you have embedded the lookup logic in the component itself. You will have to change the lookup logic before it can be reused.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator =
        (ReportGenerator) Container.instance.getComponent("reportGenerator");
    ...
}
```

A service locator can be a simple class that encapsulates the lookup logic and exposes simple methods for component lookup.

```
package com.apress.springrecipes.report;

public class ServiceLocator {

    private static Container container = Container.instance;

    public static ReportGenerator getReportGenerator() {
        return (ReportGenerator) container.getComponent("reportGenerator");
    }
}
```

Then in `ReportService`, you make a call to `ServiceLocator` to look up a report generator, instead of performing the lookup directly.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator =
        ServiceLocator.getReportGenerator();

    public void generateAnnualReport(int year) {
        ...
    }

    public void generateMonthlyReport(int year, int month) {
        ...
    }

    public void generateDailyReport(int year, int month, int day) {
        ...
    }
}
```

Figure 1-3 shows the UML class diagram after applying the Service Locator pattern. Note that the original dependency line from `ReportService` to `Container` now goes through `ServiceLocator`.

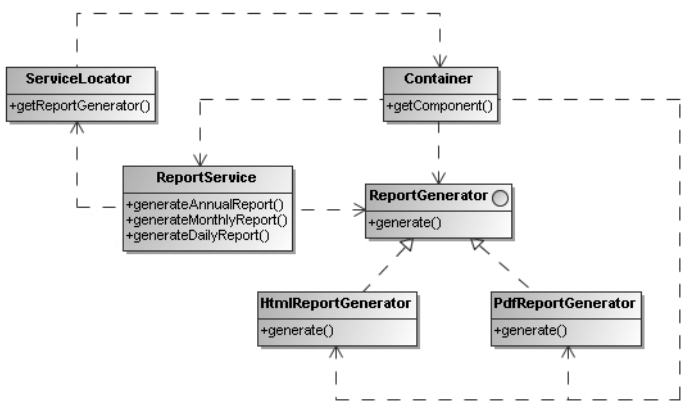

Figure 1-3. *Applying the Service Locator pattern to reduce lookup complexity*

Applying the Service Locator pattern can help to separate the lookup logic from your components, and hence reduce the lookup complexity of your components. This pattern can also increase the reusability of your components in different environments with different lookup mechanisms. Remember that it is a common design pattern used in resource (not only component) lookup.

1-3. Applying Inversion of Control and Dependency Injection

Problem

When your component needs an external resource, such as a data source or a reference to another component, the most direct and sensible approach is to perform a lookup. Let's consider this motion an *active* lookup. The shortcoming of this kind of lookup is that your component needs to know about resource retrieval, even though you have used a service locator to encapsulate the lookup logic.

Solution

A better solution to resource retrieval is to apply IoC. The idea of this principle is to invert the direction of resource retrieval. In a traditional lookup, components seek resources by making requests to a container, and the container duly returns the resources in question. With IoC, the container itself actively delivers resources to its managed components. A component has only to choose a way of accepting the resources. This could be described as a *passive* form of lookup.

IoC is a general principle, while DI is a concrete design pattern that embodies this principle. In the DI pattern, a container is responsible for injecting appropriate resources into each component in some predetermined way, such as via a setter method.

How It Works

To apply the DI pattern, your `ReportService` may expose a setter method to accept a property of the `ReportGenerator` type.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator; // No need to look up actively

    public void setReportGenerator(ReportGenerator reportGenerator) {
        this.reportGenerator = reportGenerator;
    }

    public void generateAnnualReport(int year) {
        ...
    }

    public void generateMonthlyReport(int year, int month) {
        ...
    }

    public void generateDailyReport(int year, int month, int day) {
        ...
    }
}
```

The container is responsible for injecting the necessary resources into each component. As there's no active lookup any more, you can erase the static `instance` variable in `Container` and delete the `ServiceLocator` class as well.

```
package com.apress.springrecipes.report;
...
public class Container {

    // No need to expose itself for the components to locate
    // public static Container instance;

    private Map<String, Object> components;

    public Container() {
        components = new HashMap<String, Object>();

        // No need to expose the current instance of container
        // instance = this;

        ReportGenerator reportGenerator = new PdfReportGenerator();
        components.put("reportGenerator", reportGenerator);

        ReportService reportService = new ReportService();
        reportService.setReportGenerator(reportGenerator);
        components.put("reportService", reportService);
    }

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

Figure 1-4 shows the UML class diagram after applying IoC. Note that the dependency line from `ReportService` to `Container` (see Figure 1-2) can be eliminated even without the help of `ServiceLocator`.

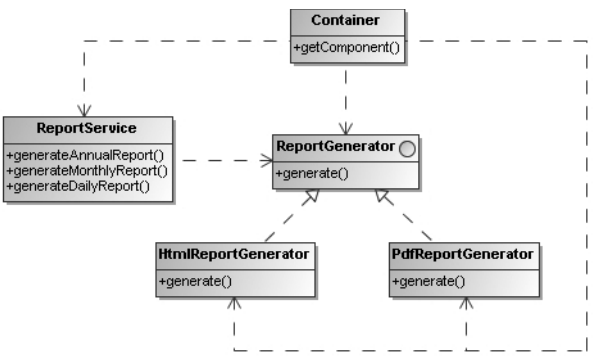

Figure 1-4. *Applying the IoC principle for resource retrieval*

The IoC principle resembles Hollywood's infamous catchphrase, "Don't call us, we'll call you," so it is sometimes called the "Hollywood principle." Moreover, as DI is the most typical implementation of IoC, the terms IoC and DI are often used interchangeably.

1-4. Understanding Different Types of Dependency Injection

Problem

Injecting a dependency via a setter method is not the only way of implementing DI. You will need different types of DI in different scenarios.

Solution

There are three main types of DI:

- Interface injection (Type 1 IoC)
- Setter injection (Type 2 IoC)
- Constructor injection (Type 3 IoC)

Of these, setter injection and constructor injection are widely accepted and supported by most IoC containers.

How It Works

For comparison, it's better to introduce the DI types in order of their popularity and efficiency, rather than by the type number.

Setter Injection (Type 2 IoC)

Setter injection is the most popular type of DI and is supported by most IoC containers. The container injects dependency via a setter method declared in a component. For example, `ReportService` can implement setter injection as follows:

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator;

    public void setReportGenerator(ReportGenerator reportGenerator) {
        this.reportGenerator = reportGenerator;
    }
    ...
}
```

The container has to inject dependencies by calling the setter methods after instantiating each component.

```
package com.apress.springrecipes.report;
...
public class Container {

    public Container() {
        ...
        ReportService reportService = new ReportService();
        reportService.setReportGenerator(reportGenerator);
        components.put("reportService", reportService);
    }
    ...
}
```

Setter injection is popular for its simplicity and ease of use since most Java IDEs support automatic generation of setter methods. However, there are some minor issues with this type. The first is that, as a component designer, you cannot be sure that a dependency will be injected via the setter method. If a component user forgets to inject a required dependency, the evil `NullPointerException` will be thrown and it will be hard to debug. But the good news is that some advanced IoC containers (e.g., the Spring IoC container) can help you to check for particular dependencies during component initialization.

Another shortcoming of setter injection has to do with code security. After the first injection, a dependency may still be modified by calling the setter method again, unless you have implemented your own security measures to prevent this. The careless modification of dependencies may cause unexpected results that can be very hard to debug.

Constructor Injection (Type 3 IoC)

Constructor injection differs from setter injection in that dependencies are injected via a constructor rather than setter methods. This type of injection, too, is supported by most IoC containers. For example, `ReportService` may accept a report generator as a constructor argument. But if you do it this way, the Java compiler will not add a default constructor for this class, because you have defined an explicit one. The common practice is to define a default constructor explicitly for code compatibility.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator;

    public ReportService() {}

    public ReportService(ReportGenerator reportGenerator) {
        this.reportGenerator = reportGenerator;
    }
    ...
}
```

The container passes dependencies as constructor arguments during the instantiation of each component.

```
package com.apress.springrecipes.report;
...
public class Container {

    public Container() {
        ...
        ReportService reportService = new ReportService(reportGenerator);
        components.put("reportService", reportService);
    }
    ...
}
```

Constructor injection can avoid some of the problems posed by setter injection. You have to provide all dependencies declared in a constructor argument list, so it is not possible for a user to miss any. And once a dependency is injected, it cannot be modified any more, so the careless modification problem does not arise.

On the other hand, constructor injection has a restriction of its own. Unlike setter injection, there is no method with a meaningful name, such as `setSomething()`, that tells you which dependency you are injecting. When invoking a constructor, you can only specify the arguments by their positions. If you want to find out more about different overloaded versions of constructors and their required arguments, you have to consult the javadoc. Moreover, if you have a lot of dependencies to inject for a component, the constructor argument list will be very long, reducing code readability.

Interface Injection (Type 1 IoC)

Alone among the three types of injection, interface injection is seldom used. To apply it, components must implement a particular interface defined by the container, so that the container can inject dependencies via this interface. Note that there are no special requirements or characteristics for the interface. It's simply an interface defined by the container for communication purposes, and different containers may define different interfaces for their components to implement.

For your simple container, you can define your own interface as shown in the next code sample. There's only one method declared in this interface: `inject()`. The container will call this method on each component that has implemented this interface and pass in all the managed components as a map, with the component IDs as the keys.

```
package com.apress.springrecipes.report;
...
public interface Injectable {

    public void inject(Map<String, Object> components);
}
```

A component must implement this interface for the container to inject dependencies. It can get the required components from the map by their IDs. As a result, the components can refer to each other without looking up the container actively.

```
package com.apress.springrecipes.report;
...
public class ReportService implements Injectable {

    private ReportGenerator reportGenerator;

    public void inject(Map<String, Object> components) {
        reportGenerator = (ReportGenerator) components.get("reportGenerator");
    }
    ...
}
```

The container has to inject all of the components, as a map, into each component to build dependencies. Note that this action must be taken after all the components have been initialized.

```
package com.apress.springrecipes.report;
...
public class Container {

    public Container() {
        ...
        ReportGenerator reportGenerator = new PdfReportGenerator();
        components.put("reportGenerator", reportGenerator);

        ReportService reportService = new ReportService();
        components.put("reportService", reportService);

        reportService.inject(components);
    }
    ...
}
```

The shortcoming of interface injection is very obvious. It requires that all components must implement a particular interface for the container to inject dependencies. As this interface is container specific, your components have to rely on the container and cannot be reused outside its scope. This kind of injection is also called "intrusive," because the container-specific code has "intruded" on your components. For this reason, most IoC containers don't support this type of injection.

1-5. Configuring a Container with a Configuration File

Problem

For a container to manage components and their dependencies, it must be configured with the proper information beforehand. Configuring your container with Java code means that you have to recompile your source code each time after modification—hardly an efficient way of configuring a container!

Solution

A better way is to use a text-based, human-readable configuration file. Either a properties file or an XML file would be a good choice. Such files do not need to be recompiled, so they speed things up if you have to make frequent changes.

How It Works

Now you will create a container based on setter injection, the easiest type to configure. For other types of injection, the configuration is much the same. First of all, let's make sure the `ReportService` class has a setter method that accepts a report generator.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator;

    public void setReportGenerator(ReportGenerator reportGenerator) {
        this.reportGenerator = reportGenerator;
    }
    ...
}
```

To configure a container from a file, you must first decide the file format. This time you will choose a properties file for simplicity's sake, although XML is more powerful and expressive. A properties file consists of a list of entries, each of which is a key/value pair of string type.

If you analyze the programming configuration in `Container`, you will find that there are only two kinds of configuration for your simple container. You can express them as properties in the following ways:

New component definition: You use the component name as the key and the fully qualified class name as the value.

Dependency injection: You join the component name with the property name to form the key, with a dot as a separator. Remember that a setter method for this property must be defined in the component class. Then the value is the reference name of another component to be injected.

To change the preceding programming configuration to properties-based configuration, you create the `components.properties` file with the following content:

```
# Define a new component "reportGenerator"
reportGenerator=com.apress.springrecipes.report.PdfReportGenerator

# Define a new component "reportService"
reportService=com.apress.springrecipes.report.ReportService
# Inject the component "reportGenerator" into property "reportGenerator"
reportService.reportGenerator=reportGenerator
```

Then your container has to read this configuration file and interpret its contents as component and dependency definitions. It also has to create component instances and inject dependencies as specified in the configuration file.

When implementing the container, you will have to manipulate component properties via reflection. To simplify matters, you can take advantage of a third-party library called Commons BeanUtils. This is part of the Apache Commons (`http://commons.apache.org/`) project that provides a set of tools for manipulating the properties of a class. The BeanUtils library requires another library from the same project, called Commons Logging.

■ **Note** You can download Commons BeanUtils and Commons Logging from the Apache Commons web site. Then include the downloaded JAR files `commons-beanutils.jar` and `commons-logging.jar` in your classpath.

Now you are ready to reimplement `Container` with this new idea. The first step is to load the properties file into a `java.util.Properties` object to get a list of properties. Then iterate over each property entry, which is made up of a key and a value. As mentioned before, there are two possible kinds of configuration:

- If there's no dot in the entry key, it is a new component definition. For this kind of configuration, you instantiate the specified class via reflection and then put the component into the map.
- Otherwise, the entry must be a dependency injection. You split its key into two parts, before and after the dot. The first part of the key is the component name and the second part is the property to set. With the help of the `PropertyUtils` class provided by Commons BeanUtils, you can refer that property to another component by the name specified in the entry value.

```
package com.apress.springrecipes.report;
...
import org.apache.commons.beanutils.PropertyUtils;

public class Container {

    private Map<String, Object> components;
```

```
    public Container() {
        components = new HashMap<String, Object>();

        try {
            Properties properties = new Properties();
            properties.load(new FileInputStream("components.properties"));
            for (Map.Entry entry : properties.entrySet()) {
                String key = (String) entry.getKey();
                String value = (String) entry.getValue();
                processEntry(key, value);
            }
        } catch (Exception e) {
           throw new RuntimeException(e);
        }
    }

    private void processEntry(String key, String value) throws Exception {
        String[] parts = key.split("\\.");

        if (parts.length == 1) {
            // New component definition
            Object component = Class.forName(value).newInstance();
            components.put(parts[0], component);
        } else {
            // Dependency injection
            Object component = components.get(parts[0]);
            Object reference = components.get(value);
            PropertyUtils.setProperty(component, parts[1], reference);
        }
    }

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

Note that in the `processEntry()` method, you have to split the entry key with a dot. In regular expressions, a single dot matches any character, so you have to quote it with a slash. Again, a slash has to be quoted in a Java string, which results in two slashes.

With these enhancements, your container has become highly reusable, as it reads a text-based configuration file for component definition. Now not even a single line of Java code has to be modified when you change your component definition. Although this container is not powerful enough for production use, it has fulfilled its purpose by demonstrating the core principle and mechanism of an IoC container.

1-6. Summary

In this chapter, you have learned to use a container to manage your components. This can help reduce coupling between different components. A component can look up other dependent components through the container. However, direct lookup will bind your components to the container with complex code. You can write a service locator to encapsulate the lookup logic and delegate the lookup requests to it.

Active lookup is a direct and sensible approach for resource retrieval. However, it requires your components to know about resource retrieval. When using the IoC principle, your component has only to choose a way of accepting resources and the container will deliver resources to your components. DI is a concrete design pattern that conforms to the principle of IoC. There are three main types of DI: setter injection, constructor injection, and interface injection.

You can configure your container using Java code or a text-based configuration file. The latter is more flexible because there's no need to recompile your source code each time you change it.

You have written a simple IoC container to read a properties file. This container is conceptually similar to the Spring IoC container.

In the next chapter, you will get an overview of the Spring framework's architecture and how to set it up.