



## Chapter 1

# From Objects to Components

Well-written Java programs are both object-oriented and component-oriented. This chapter characterizes the differences between object and component perspectives, and then it demonstrates these differences by taking a program design that is object-oriented and modifying it to be component-oriented as well.

Consider an example problem domain of contact management systems. One important facet of this domain is the ability to find contacts based on a variety of different criteria. Listing 1–1 shows a partial listing for the `Contact` and `FindContact` interfaces.

### Listing 1–1 Contact and FindContact Interfaces

```
package contacts;
public interface Contact {
    public String getLastName();
    public void setLastName(String ln);
    public String getFirstName();
    public void setFirstName(String fn);
    public String getSSN();
    public void setSSN();
    //etc. for other standard fields
}

//contacts/ContactFinder.java
package contacts;
public interface ContactFinder {
    Contact[] findByLastName(String ln);
    Contact[] findByFirstName(String fn);
    Contact[] findBySSN(String ssn);
    //other more exotic search methods...
}
```



Given these interfaces, it is easy to imagine a variety of client applications that access contact information. Listing 1–2 shows a simple `ListByLastName` console client that lists all the contacts with a given last name. Assuming that the `Contact` and `ContactFinder` interfaces have correctly captured the problem domain, this can be judged an adequate object-oriented (OO) design. Notice the complete separation of interface and implementation. `ListByLastName` uses variables only of interface types such as `Contact` and `ContactFinder`. If future versions use a different implementation, only one line of code needs to change.

#### Listing 1–2 `ListByLastName`

```
package contacts.client;

import contacts.*;
import contacts.impl.*;

public class ListByLastName {
    public static void main(String [] args) {
        if (args.length != 1) {
            System.out.println("Usage: ListByLastName lastName");
            System.exit(-1);
        }
        ContactFinder cf = new SimpleContactFinder();
        Contact[] cts = cf.findByLastName(args[0]);
        System.out.println("Contacts named " + args[0] + ":");
        for (int n=0; n<cts.length; n++) {
            System.out.println(cts[n]);
        }
    }
}
```

This design is easily extensible via inheritance. Imagine that each purchaser of the contact management system wants to add a few items of custom data to the basic notion of a `Contact`. They would simply extend the `Contact` interface, creating a different subinterface for each customer. Listing 1–3 shows a sample extension that tracks information of interest to diplomats.

**Listing 1–3 A DiplomaticContact**

```
package contacts.diplomatic;

import contacts.*;
public interface DiplomaticContact extends Contact {
    public float getSpyProbability();
    public void setSpyProbability(float newProb);
    public Contact[] getKnownAssociates();
    //etc.
}
```

The contact management design shown in Listing 1–3 does a good job of modeling the problem domain while preserving the ability to repair and/or enhance specific implementations. This is no ordinary achievement, and the success of object-oriented languages such as Java derives from their support in accomplishing these objectives. But don't start celebrating yet. The current design does not begin to address the issues of component deployment.

A *component* is an independent unit of production and deployment that is combined with other components to assemble an application. There is some conceptual overlap between objects and components. Objects are instances of classes; in fact, object-oriented design might just as well be called class-oriented design. A component is often just a compiled class, or a group of compiled classes.<sup>1</sup> One might ask, if the most important work product of both paradigms is the class, what is the significant difference between object and component approaches? The object approach emphasizes design and development, while the component approach emphasizes deployment.

Object-oriented design emphasizes the development-time relationships between entities in a system. Component-oriented design extends these relationships to other phases of the application lifecycle, particularly production and deployment. An object-oriented approach leads to questions such as the following:

1. Does the design capture the relevant part of the problem domain?
2. Are the interfaces and classes easy to extend and modify?

---

1. A component might also be some other independent unit of deployment: a text file, a graphic image, a data file, or a script.

A component-oriented approach leads to questions like these:

1. How will a client find implementation classes at runtime?
2. What happens if there is more than one version of the implementation classes available at runtime?
3. How will components locate and load necessary configuration information?
4. What happens if a process or container needs to be shut down temporarily? Can work in progress be saved and restored transparently? Can component instances migrate from one container to another?
5. How does the development and maintenance of one member of a family of products impact the other members?
6. Are components bloated by code unrelated to a particular customer's task?
7. An old component is *almost* a perfect fit for a new system. Can the old component be extended in unanticipated new ways without touching the source code?
8. What happens when part of the system must be implemented on a different software platform and seamlessly interoperate with the rest?

These sound like important questions, so why do components get so much less attention than objects do?

Any particular set of tools encourages some kinds of solutions while it discourages others. The friendly environment of a developer's computer discourages the analysis of deployment issues. At any given time, a development machine has a snapshot of a complex, evolving system. The pieces of the snapshot can be proven to fit together by compilers and other development tools. Configuration information is all in the right place, and even if it is not, there is an expert nearby who can tweak things until they work. Everyone has heard the classic refrain "It works fine on my machine!"

The real world is just the opposite of the developer's machine. Different components and different component versions get jumbled together, and applications are expected to load correctly and sort things out on their own. Configuration information is missing or inaccurate, and systems are expected to function anyway. Applications need to grow and evolve without ever shutting down, and systems must be built from disparate components that were never intended to

work together. Programs struggle under the weight of thousands of lines of code that are not related to the task at hand but cannot easily be removed.

Java is not just an OO language; it is also a platform that provides the tools to manage complex deployment. Consider again the component architecture questions raised earlier:

1. *How will a client find implementation classes at runtime?* The class loader architecture (Chapter 2) provides a flexible means for locating classes from different sources. Custom class loaders (Chapter 5) extend this architecture to support arbitrary new strategies for dynamically locating components at runtime.
2. *What happens if there is more than one version of the implementation classes available at runtime?* The class loader delegation model (§2.4.2) defines a search order. Package reflection (§3.6) can discover the version of a loaded class. You can use custom attributes (§5.5) to define a more sophisticated version-reconciliation mechanism.
3. *How will components locate and load necessary configuration information?* Components should rarely load configuration information directly from the file system. Instead, you should use the current class loader (§2.3) or the context class loader (§2.9) to load resources relative to the classes that need them.
4. *What happens if a process or container needs to be shut down temporarily? Can work in progress be saved and restored transparently? Can component instances migrate from one container to another?* Here, you should use Java serialization (Chapter 4) to write a Java instance to a stream and then instantiate an equivalent instance somewhere else. Or, use reflection (Chapter 3) to read and write the state of an object as XML.
5. *How does the development and maintenance of one member of a family of products impact the other members?* Object-oriented design provides inheritance and delegation as mechanisms to share code across a family of products. With a component-oriented approach, you can automate delegation using reflection (§3.2), dynamic proxies (§3.4), or generated code (Chapter 7).
6. *Are components bloated by code unrelated to a particular customer's task?* Straight OO designs may preserve too much flexibility, carrying unneeded code at deployment time or runtime. You can use domain analysis and code

generation (Chapter 7) to generate the exact solution you need, exactly when you need it.

7. *An old component is almost a perfect fit for a new system. Can the old component be extended in unanticipated new ways without touching the source code?* You can use dynamic proxies (§3.4) to transparently layer new functionality over existing interfaces. If you need better performance (§3.5), use reflection to generate static proxies.
8. *What happens when part of the system must be implemented on a different software platform and seamlessly interoperate with the rest?* The Java Native Interface (Chapter 6) is inadequate. You should build a marshalling layer (Appendix A) to encapsulate the details of cross-platform communication.

For those seeking immediate gratification, Listing 1–4 shows a more component-oriented approach to the contact management domain. While this is much more complete than the earlier listings, it leaves plenty of room for improvement. As you read the remaining chapters, consider how you might employ class loaders, reflection, serialization, code generation, and native code to enhance this example.

#### **Listing 1–4 A Component Approach to Contacts**

```
package contacts;
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * Factory class for the <code>Contacts</code> package.
 * Use this instead of instantiating classes directly.
 */
public class ContactFactory {
    /**
     * Process administrators should specify the concrete
     * implementation class to use by setting the
     * <code>contacts.impl.SimpleContactFinder</code> property,
     * and specify the class loader to use by setting
     * the context class loader.
     */
    public ContactFinder getDefaultFinder() {
        try {
```

```

        String className =
            System.getProperty("contacts.FinderClass",
                               "contacts.impl.SimpleContactFinder");
        Class clazz = Class.forName(className, true,
                                     Thread.currentThread().getContextClassLoader());
        return (ContactFinder) clazz.newInstance();
    }
    catch (Exception e) {
        e.printStackTrace();
        throw new Error("Default Finder not available");
    }
}

//contacts.impl.SimpleContactFinder
package contacts.impl;
import contacts.*;
import java.util.*;
import java.io.*;

public class SimpleContactFinder implements ContactFinder {
    /**
     * Default values for JNDI lookups, database table
     * names, etc.
     */
    private static Properties configProps;

    /**
     * Do not assume that a file system is available.
     * Always load co-located application resources by
     * using the class's own class loader
     */
    static {
        try {
            InputStream is = SimpleContactFinder.class.
                getClassLoader().
                getResourceAsStream("contacts/impl/config.properties");
            configProps = new Properties();
            configProps.load(is);
        }
        catch (Exception e) {
            e.printStackTrace();
            throw new Error(
                "Could not load contacts.impl.config.properties");
        }
    }
}

```

```
    }
  }
  //implementation continues...
}

//contacts.impl.SimpleContact.java
package contacts.impl;
import contacts.*;
import java.io.*;

/**
 * Data classes need to be serializable so that instances
 * can be moved from one process to another.
 * If necessary, you can wrap instances in a MarshalledObject
 * to preserve codebase information
 */
public class SimpleContact implements Contact, Serializable {
  private String lastName;
  private String firstName;
  private String ssn;

  public SimpleContact(String lastName, String firstName,
    String ssn) throws ContactsException
  {
    this.lastName = lastName;
    this.firstName = firstName;
    this.ssn = ssn;
    validateNewInstance();
  }

  /**
   * Deserialization must be validated, just like any
   * other "constructor".
   */
  private void readObject(ObjectInputStream ois)
    throws IOException, ClassNotFoundException,
    ContactsException
  {
    ois.defaultReadObject();
    validateNewInstance();
  }

  /**
   * All constructors, plus deserialization (i.e. the
```



```
* readObject method) share validation code. Throws
* application-specific ContactException if instance
* is invalid
*/
private void validateNewInstance() throws ContactsException
{
    //check valid ssn
    //check non-null name, etc.
}
//implementation continues...
}
```

```
;contacts.jar manifest file
;each package is JARred and sealed separately
;all package reflection info is specified
Sealed=true
Implementation-Title=Contacts
Implementation-Version=1.0.0
Implementation-Vendor=Stuart Halloway
Specification-Title=Contacts
Specification-Version=1.0.0
Specification-Vendor=Stuart Halloway
```

