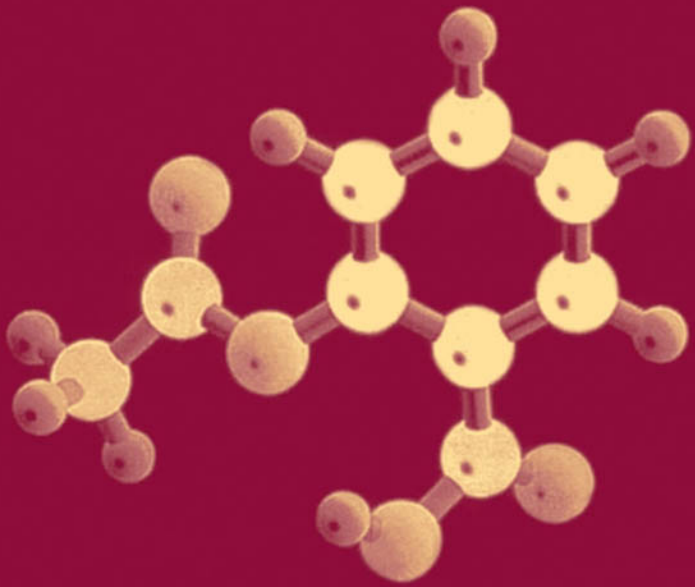
WILEY

Developing Chemical Information Systems

An Object-Oriented Approach Using Enterprise Java

Fan Li



DEVELOPING CHEMICAL INFORMATION SYSTEMS

AN OBJECT-ORIENTED APPROACH USING ENTERPRISE JAVA

Fan Li
Merck & Company, Inc.
Rahway, New Jersey

WILEY-INTERSCIENCE
A John Wiley & Sons, Inc., Publication

DEVELOPING CHEMICAL INFORMATION SYSTEMS

THE WILEY BICENTENNIAL–KNOWLEDGE FOR GENERATIONS

Each generation has its unique needs and aspirations. When Charles Wiley first opened his small printing shop in lower Manhattan in 1807, it was a generation of boundless potential searching for an identity. And we were there, helping to define a new American literary tradition. Over half a century later, in the midst of the Second Industrial Revolution, it was a generation focused on building the future. Once again, we were there, supplying the critical scientific, technical, and engineering knowledge that helped frame the world. Throughout the 20th Century, and into the new millennium, nations began to reach out beyond their own borders and a new international community was born. Wiley was there, expanding its operations around the world to enable a global exchange of ideas, opinions, and know-how.

For 200 years, Wiley has been an integral part of each generation's journey, enabling the flow of information and understanding necessary to meet their needs and fulfill their aspirations. Today, bold new technologies are changing the way we live and learn. Wiley will be there, providing you the must-have knowledge you need to imagine new worlds, new possibilities, and new opportunities.

Generations come and go, but you can always count on Wiley to provide you the knowledge you need, when and where you need it!

William J. Pesce

WILLIAM J. PESCE
PRESIDENT AND CHIEF EXECUTIVE OFFICER

Peter Booth Wiley

PETER BOOTH WILEY
CHAIRMAN OF THE BOARD

DEVELOPING CHEMICAL INFORMATION SYSTEMS

AN OBJECT-ORIENTED APPROACH USING ENTERPRISE JAVA

Fan Li
Merck & Company, Inc.
Rahway, New Jersey

WILEY-INTERSCIENCE
A John Wiley & Sons, Inc., Publication

Published by John Wiley & Sons, Inc., Hoboken, New Jersey
Published simultaneously in Canada

Library of Congress Cataloging-in-Publication Data is available.

ISBN-13: 978-0-471-75157-1
ISBN-10: 0-471-75157-X

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

For Yingduo, Melodee, and Michael

PREFACE

Although I have published several scientific articles throughout my academic career spanning 15 years, this is my first book. I consider it both an opportunity to share my experience of developing chemical information systems for the pharmaceutical industry and an opportunity for me to learn. Therefore, I do not expect this book to be perfect. I welcome feedback from the readers so that I can improve on the material for my next book.

Hundreds of books are in the marketplace about object-oriented analysis, design, and programming. A handful of books are about cheminformatics. But no book exists about how to apply object technology to the cheminformatics domain. This book is an attempt to fill that gap.

For a long time, chemical information systems have been considered special and have been dominated by a few vendor proprietary solutions. The costs for development and support of these systems are extremely high. I strongly believe that era is over. More and more cheminformatics software vendors provide open APIs for their proprietary implementations or develop their software using open technologies altogether, which offers tremendous opportunity for organizations to acquire or develop their cheminformatics solutions at a much reduced cost and with increased productivity. There is no need to rely on a single vendor to provide end-to-end solutions. This book shows how to apply the software industry's best practices, principles, and patterns while effectively integrating vendor tools to solve chemical informatics problems.

Chemical information systems are complex. This book does not cover every aspect of them. However, it uses a chemical registration system as an example of how to use an object-oriented approach to develop systems in the cheminformatics domain.

This book assumes the reader has basic knowledge of object-oriented analysis, design and programming, UML, Java, and concepts of chemical registration and searching.

FAN LI

Edison, New Jersey
fan_li_1129@yahoo.com

ORGANIZATION OF THE BOOK

Chapter 1 gives an introduction to the book: some historical background, the purpose of the book, and some basic information on chemical information systems.

Chapters 2–8 provide some general information and guidance for developing enterprise chemical information systems using object technology and the agile iterative process. I firmly believe that both object-oriented analysis and design principles and the agile iterative process are important to the success of any software development projects. The combination of the two helps a team to do the right things and to do the things right.

Chapters 9–15 use the chemical registration system as a case study to illustrate how to develop chemical information systems using the object-oriented approach and the Java technology. Chapter 9 presents an example of capturing functional requirements using a use case specification document. Other chapters talk about the implementations of each layer of the chemical registration system. Many analysis and design techniques are presented in great detail, and there are many code examples and UML diagrams in these chapters.

Chapter 16 summarizes the key points of the whole book.

ACKNOWLEDGMENTS

During my career at Merck, I received support from many people. I would first like to thank the management team of Merck Research Information Services for Basic Research: Dr. Ingrid Akerblom, Dr. Allan Ferguson, Dr. Gary Mallow, and Dr. Sanjoy Ray who supported my idea of writing this book. Without their encouragement, this book would have not been possible.

Special thanks to the Merck Chemical Informatics Application Engineering Team: Rachel Zhao, Arik Itkis, Xiping Long, LiMiao Chang, Sean Morley, Vaniambadi Venkatesan, Jarek Pluta, Irene Fishman, Jeanette Cabardo, and Dr. Hank Owens, without whom much of my research at Merck would not have been possible. A lot of information in the book is inspired by their work. I also thank Dr. Christopher Culberson of Molecular Systems of Merck Research Laboratories, who helped tremendously during the development of the Merck compound registration system.

Also, I thank my other colleagues at Merck: John Simon, Dr. Yao Wang, Dr. Annie Samuel, Dr. Jay Mehta, James Goggin, Andrew Ferguson, and Marianne Malloy. They were all part of the Merck Chemical Registration System Project Team, and many of them shared invaluable knowledge about compound data management with me.

POSTSCRIPT

I made a career change after I finished this book. I am now working at Goldman Sachs as a Technical Lead. This book was in the production phase when I joined Goldman Sachs. I am grateful to Allen Hom and Johnathan Lewis, managing directors at Goldman Sachs, for their support. Thanks to Sue Su, who helped me to establish contact with Wiley. Also, I thank the editorial and production team at Wiley: Dr. Darla Henderson, Senior Editor, Rebekah Amos, Editorial Assistant, and Kris Parrish, Production Editor.

CONTENTS

CHAPTER 1

Introduction

1.1 BACKGROUND

In 1999, I was asked by my manager to lead an application development team to lay out a strategic plan for the next generation of chemical information systems for Merck Research Laboratories. Back then, Java technology was entering its fifth anniversary, and the J2EE 1.0 specification was just launched by Sun Microsystems. However, almost all chemical information systems used by chemical, pharmaceutical, agricultural, and biotech companies were developed using vendor proprietary technologies such as MDL ISIS, which is the de facto industry standard. Although many people recognized that the cost of licensing, developing, and maintaining these legacy systems was high, an alternative to those systems was unclear. I have to admit that there was probably no viable alternative at all back then.

Since its inception 30 years ago, object-oriented technology has been successfully applied in software development in many industries for many years. However, it is a new beast even now in the chemical informatics domain. Many chemistry software vendors have been slow in reacting to technology evolution. As a user or developer, not many technological choices are available. As an employer, it is difficult and costly to find and recruit developers who have experience in those vender proprietary development platforms. There is also a fear factor in many organizations; moving away from existing technologies to new ones, no matter how promising they may be, is risky. This risk is true even though many of the limitations of the existing technologies justify the changes: performance and flexibility are low, whereas development, maintenance, and licensing costs are high.

From the middle to late 1990s, the situation changed when major chemistry software vendors started migrating their chemical information databases from proprietary formats to Oracle-based relational databases. Another positive move was that these vendors also started releasing chemical structure

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

data cartridges using the Oracle® Extensibility Framework. These products included Accelrys® Accord for Oracle, CambridgeSoft® Oracle Cartridge, Daylight® DayCart, Tripos® Auspyx for Oracle, and MDL® MDLDirect. These changes were caused at least in part by the competition among these vendors. These cartridges enable people to use direct SQL to query and update chemical databases, something that could only be done using vendor proprietary programming interfaces in the past. Software developers in the chemical informatics field now have the opportunity to use open, industry standards and more interesting technologies to do their work (like it or not, having fun is one of the biggest factors of software development productivity).

Having programmed in Java since its inception, I was a firm believer that Enterprise Java could be one alternative to vendor proprietary technologies. I proved to my managers that I was right when we finally released the first compound registration system using J2EE at Merck in 2003.

Chemical information systems are complex because they process chemical structures–a very special and complex sort of data. Indexing and querying chemical structure data require special techniques, and a handful of software vendors that have the domain expertise have come up with data storage and query solutions. The complexity also deterred many organizations from developing customized chemical information systems in-house. Instead, they hire outside consultants to implement these systems on their behalf. Many software developers in these consulting firms are not professional software devolopers by training but ended up becoming programmers for one reason or another. I remember during the technology boom in the 1990's, many "seasonal" programmers wanted to find IT jobs. Many of them did so simply because they were tired of what they were doing and believed IT jobs were easy and less stressful. People were under the impression that one could become a good programmer by just attending a two-week programming training course and learning how to write a "Hello World" program—a gross misperception. Software development projects are challenging and costly. They require special skills and disciplined practices, or they may fail badly.

The advantage for chemists in developing chemical information systems is obvious: they know the domain subject e.g., chemistry and what the systems are supposed to do very well. The disadvantage is that they do not necessarily know what it takes to develop enterprise strength software systems. There are certain people who know both very well, but it is not always the case. The consequence is that the systems developed can be hard to maintain and debug and are not as good in performance and scalability as you may expect. In many cases, only the person who wrote the code can understand and maintain it. I do not mean to offend anybody because this is purely due to a lack of training and experience and has nothing to do with talent. Neither am I suggesting that

being trained in software engineering automatically makes a person a good software developer. In fact, many chemists working in the pharmaceutical and chemical industries have advanced degrees and have trained themselves to be good software developers. I was a physicist by training initially myself and acquired a computer science degree later in my career. I learned low coupling and high cohesion principles in graduate school. They turned out to be the two most important principles in software development that have guided me since then. Software development is both an art and an engineering discipline, which in my mind requires formal training, years of practice, and continuous learning and exploration of new and better techniques.

Chemical informatics may mean different things to different people. I am not here to provide an authoritative definition. However, as it is the topic of this book, I will give a definition from the IT aspect. Chemical informatics is about capturing, storing, querying, analyzing, and visualizing chemical data electronically. Modern chemical information systems are challenged to facilitate industry's productivity growth by effectively handling a huge amount of data. Making sure these systems are robust and high-speed is crucial to the competitive advantage of any discovery research organization. Chemical information systems usually require the following tools.

1.2 CHEMICAL STRUCTURE ENCODING SCHEMA

One of the most widely used chemical structure-encoding schemas in the pharmaceutical industry is the MDL® Connection Table (CT) File Format. Both Molfile and SD File are based on MDL® CT File Format to represent chemical structures. A Molfile represents a single chemical structure. An SD File contains one to many records, each of which has a chemical structure and other data that are associated with the structure. MDL Connection Table File Format also supports RG File to describe a single Rgroup query, rxnfile, which contains structural information of a single reaction, RD File, which has one to many records, each of which has a reaction and data associated with the reaction, and lastly, MDL's newly developed XML representation of the above—XD File. The CT File Format definition can be downloaded from the MDL website: http://www.mdl.com/downloads/public/ctfile/ctfile.jsp.

Other structure-encoding schemas are developed by software vendors and academia such as Daylight® Smiles, CambridgeSoft® ChemDraw Exchange (CDX), and Chemical Markup Language (CML), and they all have advantages and disadvantages. The MDL CT File Format is the only one that is supported by almost all chemical informatics software vendors.

Figure 1.1 is the structure of aspirin.

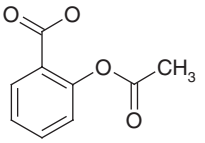


Figure 1.1 Structure of the aspirin molecule.

The Molfile representation of the above structure is as follows.

```
-ISIS-  07240513032D

 13  13  0  0  0  0  0  0  0  0  0999  V2000
-1.1556   -0.1291    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
-1.1568   -0.9565    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
-0.4419   -1.3694    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
 0.2745   -0.9560    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
 0.2716   -0.1255    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
-0.4437    0.2836    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
-0.4462    1.1086    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
-1.1667    1.5250    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
 0.9846    0.2897    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
 1.7006   -0.1201    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
 2.4135    0.2951    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
 1.7037   -0.9451    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
 0.2677    1.5221    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
 1  2  2  0  0  0  0
 6  7  1  0  0  0  0
 3  4  2  0  0  0  0
 7  8  2  0  0  0  0
 5  9  1  0  0  0  0
 4  5  1  0  0  0  0
 9  10  1  0  0  0  0
 2  3  1  0  0  0  0
 10  11  1  0  0  0  0
 5  6  2  0  0  0  0
 10  12  2  0  0  0  0
 6  1  1  0  0  0  0
 7  13  1  0  0  0  0
M  END
```

The Smiles representation of the same structure is far simpler: C(=O)(O)c1ccccc1OC(=O)C.

1.3 CHEMICAL STRUCTURE RENDERING AND EDITING TOOLS

MDL® ISISDraw and CambridgeSoft® ChemDraw are probably the most widely used structure editing tools. Both companies have a Web browser

plug-in version of these structure editing tools—MDL® ChimePro Plug-in and CambridgeSoft® ChemDraw Plug-in. MDL ChimePro also includes a JavaBean component, which can be used either as applets or in Java Swing based client applications.

Other products on the market include Daylight® Depict Toolkit, Accelrys® Discovery Studio ViewerPro, and Chem Axon® Marvin Bean.

1.4 CHEMICAL INFORMATION DATABASES

Data storage and querying are the most fundamental requirements of all informatics systems. Thanks to the Oracle® Extensibility Framework (a.k.a. Oracle Data Cartridge Technology), chemical structure data can be stored and queried using direct SQL and special query operators, such as substructure search, flexmatch search, similarity search, and formula search. Also, some indexing techniques make these otherwise slow searches fast. Detailed discussions about these databases and cartridges are beyond the scope of this book. Please refer to the vendor's website and product documentation for more information.

1.5 CHEMISTRY INTELLIGENCE SYSTEMS

These tools perform structure validations, making sure molecule structures follow certain conventions that are defined by an organization, property calculations such as molecular weight, molecular formula, pK_a, and so on, and salt handling. Many chemistry software vendors provide chemistry intelligence software. Some vendors may encapsulate chemical intelligence components in their data cartridge products. Some may bundle it with their structure editing tools. Some may offer it as independent products. MDL, for example, used to have it as part of its ISIS product suite. Now it has a product called Cheshire that is independent of ISIS and can be integrated with both Microsoft and Java platforms.

Since each organization has unique business rules, it is highly desirable that the chemistry intelligence software is flexible to allow customized implementations of chemistry rules handling. MDL Cheshire does a pretty good job from that perspective.

The above tools provide fundamental building blocks of chemical information systems. With these tools in place, you can pretty much develop customized solutions that meet your specific technical and business needs.

CHAPTER 2

Software Development Principles: High–Low Open–Closed Principles

One of the biggest challenges of all software projects is managing changes. This is true for several reasons. First, most programmers prefer developing new systems over maintaining existing systems because they feel the former is more challenging and creative and has a better sense of achievement than the latter. Developers do not want to spend most of their time supporting existing systems. Second, many software systems are poorly documented and hard to understand. Changes in one place may have unpredictable side effects in other places. Many software systems are poorly designed such that it is impossible to make changes without breaking the system.

However, no matter how much you hate it, changes in software systems are inevitable. Usually software systems that cannot be changed are short-lived and cannot survive when the business evolves, which happens all the time in drug discovery research. Isn't it nice that you could always add new behaviors to or alter the existing behavior of your software by adding new code without even touching the existing code? Wouldn't it be even nicer if there were proven solutions that could help you achieve this? This is exactly what software design principles and design patterns are about.

There are four fundamental and yet important software design principles—*low coupling, high cohesion, open for extension,* and *closed for changes*. We can simply call them *high–low open–closed principles*.

2.1 LOW COUPLING

The low coupling principle tells us that a software module should be loosely coupled with other modules in the system. Coupling is a measure of how strongly one module is connected to, has knowledge of, or depends on other

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

modules (Larman, 2005). High coupling makes the system hard to understand, change, or extend.

Low Coupling Principle: Complexity can be reduced by designing the system with the weakest possible coupling between modules.

There are two aspects to coupling: one is the number of modules to which one module is coupled; the other is how rigid these couplings are. The low coupling principle says both of them should be low. Low coupling reduces the impact of changes in one module on the rest of the system. A good analogy to this is a business organization that requires collaborations between employees. A well-organized and efficient business requires only a few collaborators for an employee to do his or her work; whereas in a poorly organized business, each employee needs many collaborators to do his or her work. In such an organization, there is a greater chance that things will break.

In object-oriented software systems, there are two types of couplings. One is inheritance (also referred to as Is-A relationship). The other is composition (also referred to as Has-A relationship). Inheritance is a more rigid coupling than composition and should be avoided if possible. In an inheritance hierarchy, changes in the interface or in the base class impose the same changes in all the subclasses. This is not necessarily a bad thing as long as all classes in the same class hierarchy share the same behaviors. (I mean behavior at the interface level, not at the implementation level, because each class in the hierarchy can have its own implementation of the behaviors.) In fact, inheritance gives you the benefits of code reuse. However, if classes in a class hierarchy do not always have the same behavior, then inheritance is not a good choice; in which case, you should consider using composition.

Figure 2.1 shows coupling by inheritance and how changes in Base propagate to all its concrete subclasses.

In a composition relationship, one object can shield changes in another object that it "owns." In Figure 2.2, Class1 owns Class2. Changes in Class2 are hidden to the clients of Class1 because Class1 wraps Class2. Figure 2.2 shows coupling by composition.

Composition is a very powerful technique and is used in many Gang of Four (GoF) design patterns (Gamma et al., 1995) such as *Strategy, State,* and *Command*. You can further reduce coupling by having Class1 referencing an interface or an abstract class instead of a concrete class as in Figure 2.3. This design enables the system to dynamically swap implementation Class2 and Class3 at runtime. Figure 2.3 shows coupling by composition through interface.

This kind of reduced coupling has direct benefits to the goals of open-closed principles as you can see later in this chapter.

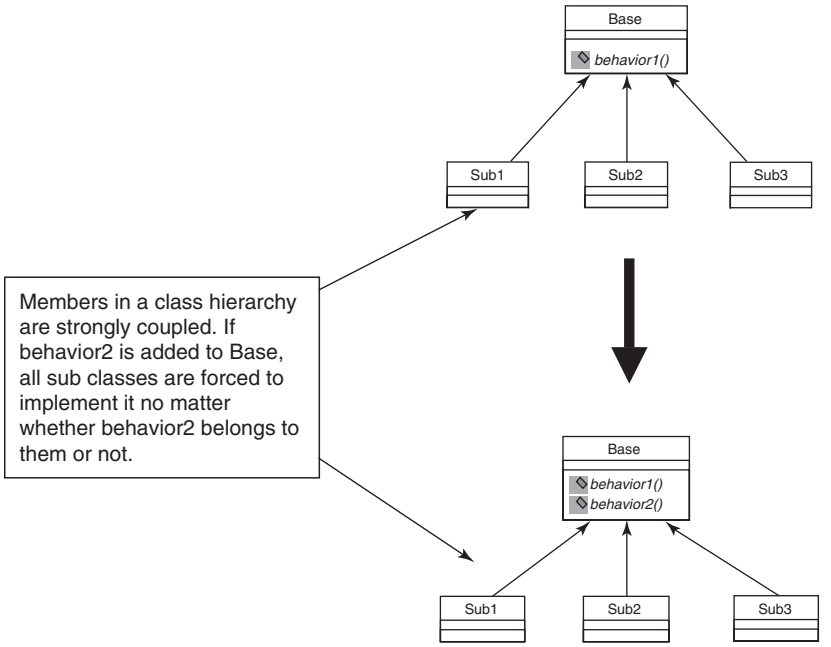


Figure 2.1 Coupling by inheritance.

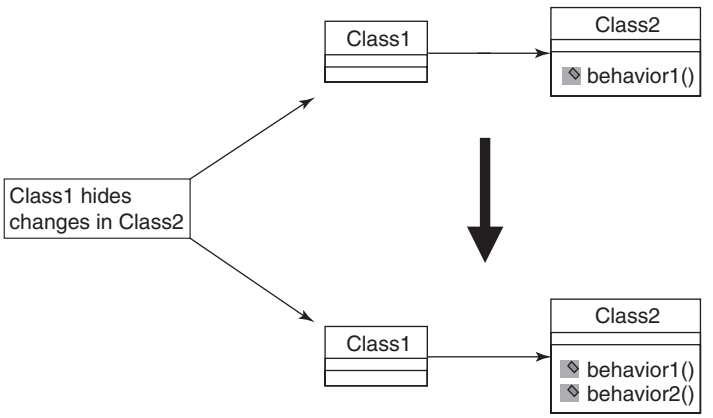


Figure 2.2 Coupling by composition.

2.2 HIGH COHESION

Cohesion is a measure of how strongly related or focused are the responsibilities of a module. A module is highly cohesive if its responsibilities are highly focused, which can be translated to the notion that a module's responsibilities should all be related. Or to be more extreme, a module should have only one

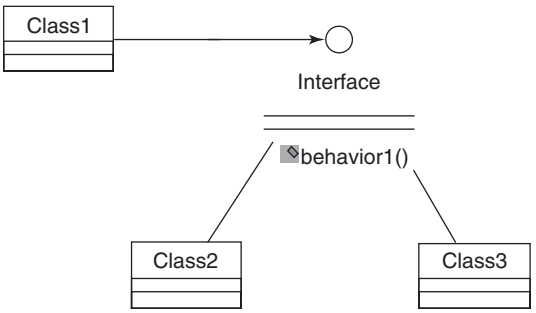


Figure 2.3 Coupling by composition through interface.

responsibility or one reason to change. Robert Martin's (2003) book has very good explanations about the high cohesion principle.

High Cohesion Principle: Responsibilities of a module should be highly related and focused so that the module has only one reason to change.

Some techniques can help you to achieve high cohesion—one of which is to use descriptive names for your classes and methods. Descriptive names can help you to keep the classes and methods focused. When you add responsibilities to your classes or methods, think about whether these responsibilities have any relevancy to the names of the class and method. If not, most likely it does not belong there. Never use ambiguous names for your classes and methods because they make the code hard to understand and most likely lead to low cohesive design. The same rule applies to member and local variables. Here are some bad names: MyClass and myMethod. These names should never be used in your code (although I use these names in this chapter to describe some concepts, they are not recommended in the real world). Here are some good names: Molstructure, ChemistryConventionChecker, and CompoundRegistrationService. Another technique is to keep the module short. If the size of a class or a method is large, usually it is a bad sign indicating the class or method is not focused enough and you should consider moving some of the responsibilities out of the class or method.

High cohesion makes the system easy to understand, reuse, and extend.

2.3 OPEN FOR EXTENSION AND CLOSED FOR CHANGES

These two principles are closely related.

Open (for Extension) – Closed (for Changes) Principles: Modules should be open for extension and adaptation and closed for modification in ways that affect its clients.

Here is a real-world example for illustrative purposes. Suppose you have a chemical information system that has to support both Molfile and Smiles structures and a business method in a business object has to get the molweight and molformula from the Molstructure objects to fulfill its responsibilities. A naive design is to have two versions of the business method: one takes a Molfile structure object as input and another takes a Smiles structure as input (Figure 2.4).

With this design, if a new structure format (e.g., CML) is added to the system, another version of the business method has to be added to the BusinessObject. This design obviously violates open–closed principles. Figure 2.5 shows a better design.

First, we create a higher level of abstraction—an abstract class Molstructure—and make MolfileStructure and SmilesStructure subtypes of Molstructure. Instead of having two or more versions of businessMethod, each one takes a different format of Molstructure as input; now BusinessObject only has one business method that takes the base type Molstructure as input, and dynamically, it invokes the calculateMolweight and calculateMolformula methods of either MolfileStructure or SmilesStructure depending on which type of object is passed in at runtime. With this kind of design, when a new structure format (e.g., CML) is introduced to the system, all we need to do is

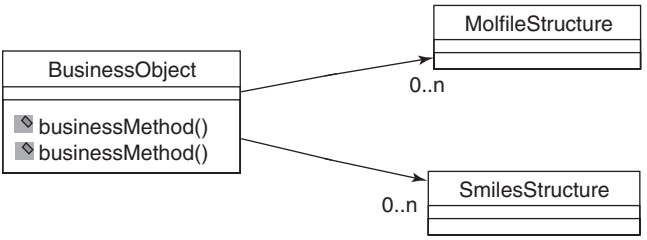


Figure 2.4 A design that is against open–closed principles.

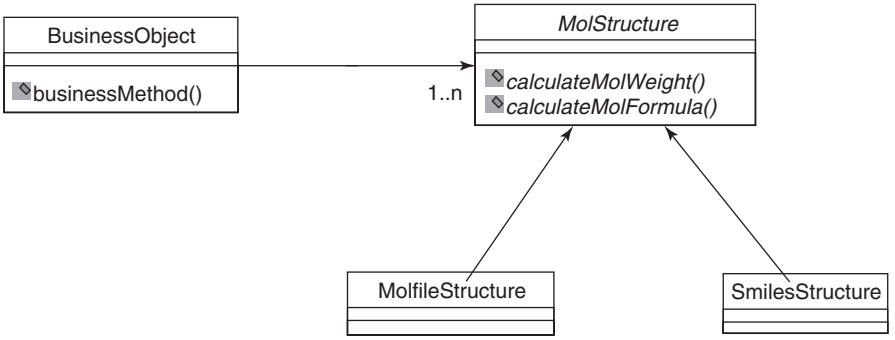


Figure 2.5 A design that is open for extension and closed for changes.

implement another subtype of Molstructure and everything else still works without any changes. The above design approach is described as **Strategy Pattern** in the GoF design pattern book (Gamma et al., 1995).

> **The Strategy Pattern: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients who use it.**

High–low open–closed principles should be applied in accordance. They are independent and yet related. Applying one principle can usually help to achieve other principles. Their goal is to manage changes, and you will find that many design patterns are realizations of these principles.

CHAPTER 3

Introduction to the Object-Oriented Approach and Its Benefits

Most high-level programming languages can be categorized into one of the four following paradigms: procedural (e.g., Basic, C, FORTRAN, MDL ISIS PL, and Pascal), scripting (e.g., JavaScript, VBScript, Perl, and Cheshire), 4-GL (e.g., Visual Basic, and PowerBuilder), and object-oriented (e.g., C++, C#, Java, Ruby, and SmallTalk). Each of these paradigms has advantages and disadvantages, and many software developers program in all of them during their careers. I am an object fan although I have used all four of the above paradigms depending on the systems I develop. As you can see from its title, this book advocates an object-oriented approach.

As described in Chapter 2, managing changes is one of the biggest challenges of software development. Most of the design principles and techniques are aimed at making software systems easy to change. Object-oriented programming provides the following four features that help software professionals to achieve good design.

3.1 ABSTRACTION AND ENCAPSULATION

Abstraction, along with encapsulation, is a technique that hides the internal structure and implementation details of an object or some other software unit with its external interfaces. In this chapter, I focus on objects. Other software units include components, subsystems, and services. They will be discussed in subsequent chapters. Abstraction is about what a software module looks like to the outside world. Encapsulation uses these "looks" to hide the module's implementation details. At first glance, abstraction may not sound like a big deal. Quite the opposite, a system with well-designed abstractions greatly reduces couplings between its building blocks and is much easier to understand, maintain, and extend. Because highly coupled software systems

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

are difficult to change, all software designers must find ways to reduce couplings between the building blocks of the systems.

To illustrate an object's interface versus its implementation, I would like to borrow a concept from an ancient Chinese philosophy of Taoism. Taoism believes that all objects in the universe are governed by two balancing forces—Yin and Yang. Yin represents the passive, introvert, and hidden aspects of an object. Yang represents the active, extrovert, and exposed aspects of an object. Therefore, we can consider an object's implementation detail that is hidden from the outside world as its Yin and its interface that is exposed to the outside world as its Yang.

All object-oriented programming languages provide a feature called "access modifier" that facilitates separations between the interfaces and the implementations of an object. The way to achieve this result is to define member variables of the object as private or protected and define methods that provide services to the object's client as public. Only the public elements of an object can be accessed by its clients, and these are all that its clients care to know. In Java and C#, we can go even further by creating interfaces that have only method signatures. The implementations of these methods are provided by the classes that implement these interfaces. Although there is no interface in C++, you can, however, create an abstract class by declaring one or more of the class's methods as pure virtual. Abstract class is also supported by Java and C#. Interfaces and abstract classes are useful software constructs for defining abstractions in a software system.

In object-oriented programming, use of global variables should be avoided although it is not forbidden. You can still do so by declaring public member variables in a public class. However, the difference is that in object-oriented programming, you do not need to use global variables and still achieve your programming goals. The way to do it is to declare all member variables private or protected and yet provide public methods that access these variables.

Figure 3.1 The balance of Yin (black) and Yang (white).

The fact that a programming language facilitates encapsulation and encapsulation is a good programming practice does not guarantee that all developers know how to do it correctly. There is a fine line between knowing a language and knowing how to design software. People often do not distinguish the differences between the two skill sets—language syntax and design skills—and wrongfully believe that knowing language syntax is more important than knowing how to design software. During many job interviews, interviewees are grilled much harder with language syntax questions than with design questions. Putting the incidental before the fundamental ways of dealing with software development is in my mind one of the reasons why many software projects fail.

Although the following example has been used by other authors in different contexts, I do not hesitate using it here again to demonstrate how to build systems with better abstractions. The reason is it uses the Java Collection Framework—a Java class library that is familiar to many developers and the framework itself is a good example of encapsulation. Suppose you want to design a compound library class that contains a list of individual compounds. Also suppose that the clients of the CompoundLibrary class need read-access to the compound list and the developer decides to use ArrayList to hold the compounds inside the CompoundLibrary class. A naive implementation of the CompoundLibrary class is as follows:

```
public class CompoundLibrary {
   ArrayList compoundList = new ArrayList();
   public ArrayList getCompounds() {
      return compoundList;
   }
}
```

It is naive because the getCompounds method of CompoundLibrary class returns a concrete data type—ArrayList—instead of an abstraction. If for some reason the developer of CompoundLibrary class decides to switch compoundList to a Vector, or a LinkedList, or some other customized List implementation, the clients of CompoundLibrary class all need to change because none of these implementations are interchangeable. This kind of coupling between the internal data structure of a class and its clients is not desirable and can be avoided by using better abstractions to hide CompoundLibrary's implementation details with their interfaces.

A better solution is as follows:

```
public class CompoundLibrary {
   ArrayList compoundList = new ArrayList();
   public List getCompounds() {
      return compoundList;
   }
}
```

Now the method getCompounds returns an interface—List, which is a super type of all possible concrete List classes. No matter what kind of list CompoundLibrary uses to hold its compounds, its clients do not need to care any more because what they get is the common abstraction: List. Another way to achieve this is to have getCompounds to return an iterator. Please note the iterator() method in Java Collection Framework creates a new iterator object every time it is called and therefore is an expensive operation and should be used with discretion.

Note: Whether the member variable compoundList of CompoundLibrary should be declared as an interface—List or a concrete type—ArrayList should be determined on a case-by-case basis. If the concrete class has methods that are not defined in the interface or the abstract class, you are better off defining the variable as the concrete type. Otherwise, you need to explicitly cast the variable to the concrete type every time you use those methods. Either way, the clients of CompoundList are no longer affected by the decision made by the developer of CompoundLibrary with regard to the data type of compoundList variable, which is what abstraction or encapsulation is all about.

In fact, CompoundLibrary has another problem—the compound list that getCompounds method returns is modifiable by its clients—the clients can add and delete elements in the list. This problem still breaks encapsulation and may introduce many undesired side effects. What if CompoundLibrary needs to apply some business rules when new compounds are added to the compound list, for example, certain structure conventions have to be followed by the compounds, molecular weight has to be in a specific range, or the compounds have to be added in chronological order? If the clients are allowed to add new compounds, these rules might be violated, which is against the principle of encapsulation. Even if there are no such business rules at the initial phase of development, it is still a good idea to protect the data inside a class from being modified directly by its clients. Otherwise, changes to the class may propagate to many different places in the system, and hidden side effects are very difficult to debug at a later phase. The following is a better solution in which the getCompounds method returns an unmodifiable list. Another method, addCompound, is added to the class for adding compounds to the CompoundLibrary object.

```
class CompoundLibrary {
   ArrayList compoundList = new ArrayList();
   public List getCompounds() {
      return Collections.unmodifiableList(compoundList);
   }

   public void addCompound(Compound aCompound) {
      //...some business rules
      compoundList.add(aCompound);
   }
}
```

Notice that the clients of CompoundLibrary class have no knowledge about how these methods are implemented. They do not know of any business rules that are included in these methods. Neither do they know how compounds are kept within the CompoundLibrary class. This knowledge belongs to the *Information Expert (*Larman, 2005), which in this case is the CompoundLibrary class and is hidden from its clients. All a client can do is to send a message by invoking the methods of a CompoundLibrary object and expect something will happen as the result of the method invocation. Everything else is left to the CompoundLibrary to decide. This is the power of encapsulation.

Information Expert: Assign a responsibility to the information expert—the class that has the information to fulfill the responsibility.

3.2 CODE REUSE THROUGH INHERITANCE

There are different types of code reuse. Here we focus on code reuse using a class hierarchy—in other words, through inheritance.

Suppose we want to develop a module that represents the chemical structure of compounds. A structure is the signature of a compound that, in most cases, uniquely defines all chemical properties of the compound such as molweight, molformula, stereo chemistry, pKa, and logP. Suppose a structure can be represented in many different formats—Molfile, Chime, Smiles. The algorithms of calculating the chemical properties are different depending on the structure format, and our application has to support all of them. A naive solution is to develop a class for each structure format and repeat every common attribute and method in all of them.

The Molstructure class for the Molfile format is as follows:

```
public class MolfileStructure {
   private String format = "MOLFILE";
   private String structure = null;
   public MolfileStructure(String structure) {
      this.structure = structure;
   }

   public String getMolstructure() {
      return structure;
   }

   public String getFormat() {
      return format;
   }
```

```
    public float getMolweight() {
        float molweight = 0f;
        //... some calculation logic specific to molfile format
        return molweight;
    }

    public String getMolformual() {
        String molformula = null;
        //... some calculation logic specific to molfile format
        return molformula;
    }
}
```

The Molstructure class for the Smiles format is as follows:

```
public class SmilesStructure {
    private String format = "SMILES";
    private String structure = null;
    public SmilesStructure(String structure) {
        this.structure = structure;
    }

    public String getMolstructure() {
        return structure;
    }

    public String getFormat() {
        return format;
    }

    public float getMolweight() {
    float molweight = 0f;
        //... some calculation logic specific to smiles format
        return molweight;
    }

    public String getMolformual() {
        String molformula = null;
        //... some calculation logic specific to smiles format
        return molformula;
    }
}
```

The above two classes have a lot of duplicated code. Not only is this against productivity, but it also makes the application difficult to change. A better way is to introduce a common base class and to *refactor* the common code to the base class.

The base class Molstructure is as follows:

```
abstract public class Molstructure {
        public static String SMILES_FORMAT = "SMILES";
        public static String MOLFILE_FORMAT = "MOLFILE";
```

```
    public static String CHIME_FORMAT = "CHIME";

    private String format = null;
    private String structure = null;

    public Molstructure(String format, String structure) {
      this.format = format;
      this.structure = structure;
    }

    public String getMolstructure() {
      return structure;
    }

    public String getFormat() {
      return format;
    }

    abstract public float getMolweight();
    abstract public String getMolformual();
}
```

The Molstructure class is defined abstractly for two reasons:

1. You would not create an instance of Molstructure without knowing its format.
2. The algorithm of the molweight and molformula calculations depends on the actual format. The Molstructure class does not know how to calculate them because it does not know the format until runtime. Hence, these two methods are declared as abstract. One can argue that you can implement getMolweight and getMolformula according to the format member variable that is specified when the constructor is called using the if-else conditions. However, that will require that these two methods get changed any time when a new format is introduced to the system and therefore is against the closed for changes principle that is discussed in Section 2.3.

The new definition of MofileStructure and SmilesStructure classes is as follows:

```
public class MolfileStructure extends Molstructure{

    public MolfileStructure(String structure) {
      super(MOLFILE_FORMAT, structure);
    }

    public float getMolweight() {
      float molweight = 0f;
```

```
        //... some calculation logic specific to molfile format
        return molweight;
    }

    public String getMolformula() {
        String molformula = null;
        //... some calculation logic specific to molfile format
        return molformula;
    }
}

public class SmilesStructure extends Molstructure{
    public SmilesStructure(String structure) {
        super(SMILES_FORMAT, structure);
    }

    public float getMolweight() {
        float molweight = 0f;
        //... some calculation logic specific to smiles format
        return molweight;
    }

    public String getMolformula() {
        String molformula = null;
        //... some calculation logic specific to smiles format
        return molformula;
    }
}
```

Notice that now each of these two classes extends Molstructure class and all common code is removed from them. This result is because the common behaviors are inherited from the common superclass—Molstructure. Also notice that even some logic in the constructor is inherited from the superclass. Now we have achieved some code reuse through inheritance by having a class hierarchy.

There are other types of reusability, one of which is software components. Software components are typically executables distributed as .jar (Java), .dll (windows), or .so (Unix) files. Components with well-designed abstractions can provide reusability for many different software systems. Many commercially or freely available reusable components are developed using object-oriented technologies. The Java Collection Framework is a good example.

Service-oriented architecture (SOA) is another software reusability enabler that has become very popular these days. SOA is not limited to object-oriented technologies. Web service is the most talked about SOA that uses XML-based messaging between the service provider and the service consumer. In SOA, a service consumer uses some services somewhere in the network to do its own work. In most cases, service provider and service

consumer run on different hardware. The consumer looks up service providers from a service registry (e.g., UDDI) and requests services from the provider via remote method calls. This kind of architecture determines that in SOA all parties should be loosely coupled. The consumer's core functionality should not be compromised even if the service provider is not available at runtime, or at the very least, asynchronous messaging between the consumer and provider has to be possible. Also, the service provider can be swapped out and replaced by a new provider without impact to the comsumer. More detailed discussion about SOA is beyond the scope of this book.

A very important and yet less commonly talked about reusability is applying various software patterns. The difference between patterns and other types of reusability is that patterns provide reusability through knowledge and experience sharing rather than through code sharing. Patterns will be discussed more throughout the book.

3.3 POLYMORPHISM AND DYNAMIC PLUG-AND-PLAY

Being able to extend or alter the functionality of the system without changing, recompiling, and redeploying the existing code is a dream of all programmers. Object-oriented programming achieves this capability by leveraging polymorphism and dynamic binding (also known as method overwriting, late binding, or runtime binding). The idea is to keep the coupling between software modules at the interface level rather than at the implementation level so that at development time, the system does not know or does not care which implementation is used at runtime. The binding of the implementation to the system happens at runtime, and hence, the actual behavior of the system is realized at runtime.

Suppose you want to implement a class CompoundRegistrationService that has a register() method that registers compounds into your compound database. Also suppose the compound being registered can be in Molfile, Smiles, or some other format, and molweight and molformula need to be calculated during the registration process. A naive solution is to have the register method take a concrete structure type as an argument in the method signature:

```
public class CompoundRegistrationService {
   public void register(MolfileStructure structure){
      //... do something
      structure.getMolformula();
      structure.getMolweight();
      //... do something
   }
}
```

This implementation restricts CompoundRegistrationService to work only with MolfileStructure. To support Smiles structures, either an overloaded register method or another CompoundRegistrationService class for Smiles structure format has to be implemented. The latter solution may require code duplications that are not desirable.

A better solution is as follows:

```
public class CompoundRegistrationService {
   public void register(Molstructure structure){
      //... do something
      structure.getMolformula();
      structure.getMolweight();
      //... do something
   }
}
```

Notice that the new register() method takes the abstract class Molstructure as input. At compile time, it does not care whether MolfileStructure or SmilesStructure is bound to it when the application runs. At runtime, depending on what concrete type is passed into the register() method by its caller, CompoundRegistrationService behaves according to the implementation details of MolfileStructrure or SmilesStructure.

One may wonder who decides whether to create MolfileStructure or SmilesStructure objects. Well, the answer is that it depends. Either it can be configured at deployment time by an application configuration file that tells the application what type of structure format is used, or it can be generated by the structure drawing tool being used at runtime, or it can be created by a factory object that makes the decision according to the runtime environment. In any case, the application logic no longer cares what type of structure it processes. It works on behalf of the system according to what is given to it at runtime.

Polymorphism is one of the most unique and yet powerful features of object-oriented programming. Many design patterns are based on the ideas of polymorphism. If used properly, it can greatly improve the design of the system and reduce the maintenance cost.

Encapsulation, inheritance, and polymorphism are the most well-known features of object-oriented programming. These features are provided by all object-oriented languages.

3.4 PATTERNS: SOLUTIONS TO RECURRENT PROBLEMS

Many object-oriented patterns are codified and published by experienced object experts and thought leaders. The most well-known ones are the GoF design patterns (Gamma et al., 1995), Martin Fowler's analysis patterns

(Fowler, 1997), Patterns of Enterprise Architecture (P of EAA) (Fowler, 2003a), Craig Larman's UML and Patterns (Larman, 2005), Robert Martin's Principles, Patterns, and Practices (Martin, 2003), and the J2EE Patterns (Alur et al., 2003). Patterns are proven solutions to recurrent problems that can be applied in various contexts. When a problem arises, keep in mind that there might be solutions that have been applied again and again by others to the same problem. Programmers do not need to reinvent the wheel if they understand these patterns and know how to customize them to serve their needs. Patterns can be combined to build application frameworks.

No matter how good a particular technology is, it does not provide assurance for good design. It is still up to the architects and developers to get things right. Some people write procedural-like code using object-oriented language. You can see from the code examples in this chapter how things can be programmed differently using the same language. Educating developers on good design principles and techniques remains a challenge. Many development tool vendors try to incorporate patterns into their integrated development environments (IDEs) to help average developers write better code. In my opinion, none of the tools can yet replace humans. It will be interesting to see how Object Management Group's (OMG) Model Driven Architecture (MDA) works out.

CHAPTER 4

Build Versus Buy

Every software project has to answer this tough and sometimes very political question: Should we buy or build? I do not intend to give readers definitive answers. But I will share some advice based on my experience.

The most cost-effective solution for a software project is to buy a good product that meets your needs off the shelf. Unfortunately this is not always possible. Vendor products are often too generic and require significant customization to be useful to your organization. Outsourcing is another way of buying software solutions, and they will be discussed in this chapter.

Software development is still a risky business, and its failure rate is high. To make things worse, the complexity of software systems still grows. One developer said to me that the large systems that they develop always go wrong when they are deployed to production. You should not build if there is lack of expertise in your organization, which includes both technical and domain knowledge expertise. For example, if you want to develop chemical information systems in-house, you need architects and developers who know the technologies and the vendor software being used, as well as people who have a strong organic chemistry background and who fully understand the chemistry conventions of your company. More ideally, you should have at least some people who have both domain and technical expertise. In most cases, you should not build if you do not have business and executive support. On the other hand, is there a guarantee that you can find good outsourcing or consulting firms to do the jobs for you? The answer is unfortunately "No." Odds are if there is lack of in-house expertise, it is impossible to find good outsourcing firms or consultants yourself simply because you are not able to judge whether they are good or bad. If that is the case, you either look for firms that have good reputations in your business domain or hire yet another third party to do the screening for you. Unlike other industries, many choices are not available in the chemical information area in the market.

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

If you do have in-house expertise and executive support, and if members of the project team work together as a team, in-house development has a better chance to succeed than outsourcing. The reason is an in-house development team is much closer to the end users and therefore is better positioned to be responsive to end users' feedback and to be more agile and adaptive to changes. In other words, compared with outsourcing, in-house development can easily adopt the agile development process—a methodology that is proven to be much better than waterfall. Chapter 5 will discuss the agile iterative development process in much more detail. Some consulting companies embrace agile methodologies. They do so by full-time deployment of their developers to customer sites.

Even if you do want to develop in-house, you should still balance between what you have to build and what you can buy. Not everything should be developed in-house.

Those that you should consider buying are as follows:

1. Structure rendering and editing tools: There are so many mature commercial products from which to pick. MDL and CambridgeSoft are dominant in this arena. MDL offers proprietary solutions such as ISISBase/ISISDraw, a Web browser plug-in Chime, and, recently, Java and .NET based MDLDraw. ChemDraw from CambridgeSoft is another popular product.
2. Structure representation and chemistry intelligence engine: As described, commercial tools are available and there is no need to reinvent the wheel. Plus, developing these tools requires significant domain expertise and resources. However, influencing the tool vendors to continually improve their products is a good idea.
3. Molecule database and data cartridge products: Similar to the above two, commercial software is available and there is no need to reinvent the wheel.

Several issues need to be taken into consideration when choosing commercial products:

1. Compatibility: Not all commercial software packages are compatible with each other. Although many vendors provide tools to convert between structure formats, there might be information losses when the conversion takes place. You probably have to stick with one vendor for your entire system or at least most of the system. However, it is still possible to pick other vendors for some special purposes.
2. Performance, functionality, and complexity: Not all products are equal. Some vendors provide better performance, whereas another vendor

may have better structure representation and chemistry intelligence. For example, MDL provides better chemistry intelligence, whereas Daylight® products may perform better. One has to balance these different aspects when picking a vendor solution according to need.

Some final notes: projects for which most requirements can be clearly defined upfront and have a low level of uncertainty are good candidates for outsourcing. Software upgrade is one example. In my opinion, a project like this has clearly defined requirements—upgrade the software infrastructure (OS, Oracle, ISIS, etc.) from one version to another—and yet it is time consuming, resource intensive, and does not provide a huge competitive advantage. On the other hand, a new development that has a high level of uncertainty, represents the uniqueness of the research organization, and requires a quick solution is a good candidate for in-house development provided the expertise exists. Doing so avoids the risks of lengthy contract negotiations, inflexibility to changes once the contract is signed, and lack of business knowledge by outsiders, and yet preserves the competitive advantages. Keep in mind that projects like these are subject to frequent requirement changes and uncertainties that may require re-negotiation of contracts if outsourced. Usually a contract requires many back and forth negotiations and therefore is time consuming and difficult to respond to business in a timely manner. This type of project also requires significant exploration and close interaction with business areas throughout the development cycle due to a high level of uncertainty and change rates. In many cases, this interaction cannot be easily done if outsourced.

Can outsourcing succeed in a new development project? The answer is yes. But it requires good management practices. Never expect to fully specify and freeze all requirements up front, to give the requirements and a deadline to the vendor, and, on the project end date, to receive a system that meets business needs. Even if the vendor believes the above is enough for them to deliver the system, you should not let the project go that way. Ask the vendor to deliver a partial system periodically and incrementally (see Chapter 5 on the agile iterative development process) and give end users and business people the opportunity to try the partial system and provide feedback. Also, ask the vendor to deliver quick prototypes at the beginning of each iteration and let the end users or business people review them. The contract should state that requirement changes are expected during the development process based on feedback and should be factored into the pricing and timelines. These changes may include newly added features, elimination of no longer needed features (research shows about 65% of planned features are never used or rarely used) (Larman, 2005), and functional and nonfunctional changes of planned features. The above practices are not optional but mandatory in order to make sure the system meets business needs.

CHAPTER 5

The Agile and Iterative Development Process

The reason I include this seemingly irrelevant chapter in this book is because it is important. What are the factors that make a software project succeed? One could say the project has to provide good business values. In addition, the project team has to be technically competent. Although these factors are all necessary for the project to succeed, they are not sufficient. Especially for a software project that has some level of complexity, tight timelines, and resource constraints, a good development process is also critical. Unless your development process is ad hoc, most likely you will use either the waterfall or the iterative development process. Many researchers show that the latter is a much better choice than the former, and many thoughtful leaders advocate its use in most software development projects.

5.1 BUSINESS CASE AND PRINCIPLES

Craig Larman (2004) presented the rationales and business cases for the agile iterative development process in his well-written book, *Agile and Iterative Development Process*: *A Manager's Guide.* The book also provides guidance on how to apply the agile iterative process to software projects and common mistakes people tend to make. Although interested readers can get a copy and read the book to get a more thorough coverage of the subject, I want to spend some time discussing it in this book because it is so important to software projects.

If you are a software professional, most likely you have run into situations when, after months or even years of development work, you find that your users do not like the product because either the system does not meet requirements (this may be caused by ill-stated and incomplete requirements by the

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

users or misunderstood requirements by the development team), the system does not pass test specs, or the requirements have changed so much that the system you developed has become irrelevant to the business. This occurs because there is a missing link between the software under development and the business organizations that sponsor the software project throughout the development cycle. In a waterfall development process, user involvement happens only at the very beginning and at the very end of the development cycle. There is no ongoing feedback from business areas as to whether the project is on or off target.

Figure 5.1 illustrates the waterfall process. Most software development projects deal with moving targets—changing business, changing requirements, and changing priorities. The waterfall model is not suited for this type of software development.

In contrast, in an agile iterative development process, the project is divided into short, manageable, and timeboxed iterations. Timeboxing means the starting date and end date are fixed and not changeable. Each iteration implements a small subset of features with most architecture and business significant features being implemented in early iterations. At the end of each iteration, the partial system is delivered to end users for testing and feedback. These systems are partial but with production quality, not throwaway prototypes. The feedback influences what the next iteration should do. If business requirement changes happen at any point of time in the project and some features that were not planned at the beginning of the project become important, they can be included in the forthcoming iterations. This kind of continuous delivery and feedback makes it possible for a project team to always do what is most important at any point of time during the project and to make sure what is delivered truly meets

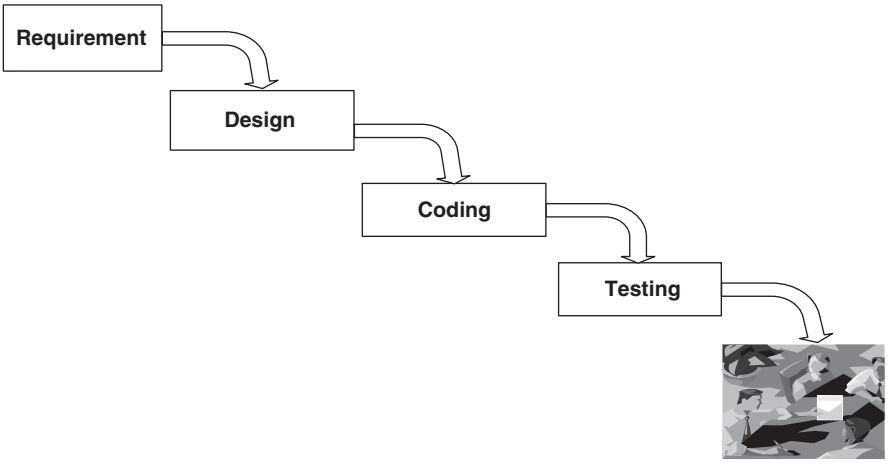


Figure 5.1 Waterfall development model.

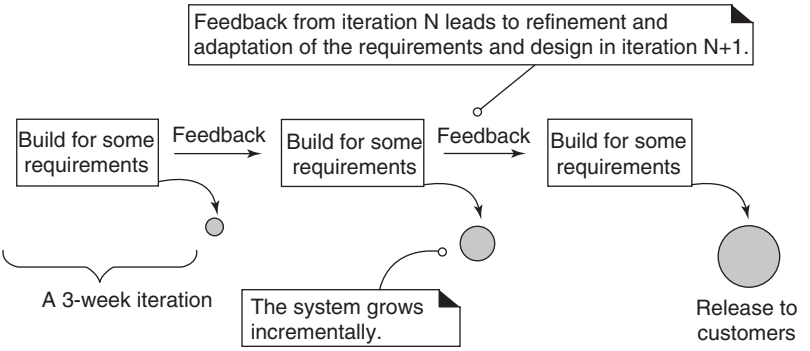


Figure 5.2 Iterative and incremental development (From C. Larman, *Agile and Iterative Development: A Manager's Guide*, p. 10, Figure 2.1. Copyright © 2004 Pearson Education, Inc. Reproduced by permission of Pearson Eduction, Inc. All rights reserved.)

requirements and business needs. Figure 5.2 illustrates how the iterative process develops and delivers software incrementally (Larman, 2004).

The agile iterative development process is an opponent to upfront detailed planning and complete requirement and design specifications because studies show that software projects are unpredictable by nature. There is no way requirements can be predefined and frozen before design starts. Likewise, there is no way that design can be complete, accurate, and frozen before coding starts. The agile iterative process expects and is willing to incorporate changes and to alter the project plan accordingly. It does so by constantly delivering the partial system and soliciting feedback. As opposed to the upfront complete requirement specification, in a typical iterative project, a list of functional and nonfunctional requirements is developed with a name and a brief description with no details before the first development iteration starts. Before an iteration starts, pick a few features that are most important, have the highest business values, or are the most risky. At the beginning of the iteration, a detailed requirement discussion is held only to discuss the features that are being implemented in that iteration; design, implementation, and testing all happen at roughly the same time throughout the iteration.

We recently successfully applied the agile iterative process in a compound registration project at Merck. We broke the project into short iterations of 3 to 6 weeks. We implemented the most risky and architecturally significant features in the early iterations. Those features were also of high business value. We allocated more weeks for the early iterations because they were more complex and required more time in order to deliver something that is significant enough for getting meaningful feedback. After the first two iterations, the entire architecture was laid out, which only needed small changes

later on when the remaining features were implemented. Later iterations were shorter and for less-significant or low-risk features. After each iteration, we conducted both QA and user testing to get feedback on the quality, usability, and functionality of the system. We also did load and performance testing after the early iterations when most architecturally significant features were implemented to test the performance and reliability of the architecture. Even though during the early load and stress testing we discovered database cursor leaks, we had enough time to fix the problems. Although there were concerns about the implications to the QA and business teams at the beginning of the project because they needed to be involved early and more frequently in the project, because each iteration only delivers a small portion of the features, the amount of testing at the end of each iteration was manageable. The outcome was great. The development team was in high morale because they could constantly see the results of their work and get feedback from the end users. We were able to deliver a nine-month project on time and within budget, and incorporate new requirements late in the development process.

However, there are some lessons learned. It is very important to communicate the ideas and benefits of the agile iterative development process to the entire project team and have buy-ins not only from the developers, but also from business analysts, the QA team, and end users. We only have a small QA team that supports many projects and end users are busy with their priorities. It is important to have some rough iteration planning upfront and have some idea about when QA and end users need to be involved in testing. If the plan has to change, inform QA and end users as early as possible.

5.2 KEY PRACTICES

There are several flavors of the agile iterative process. The most popular ones are the Rational Unified Process (RUP) (Kruchten, 2003) and Extreme Programming (XP) (Beck, 2004). Other flavors include Scrum (Schwaber and Beedle, 2001), Evo, and Feature-Driven Development (Palmer and Felsing, 2002). The following list of key practices is found in all agile iterative development processes that in my view are the most important. For details, please refer to Larman (2004) and Beck (2004).

1. Short, timeboxed iterations (Larman, 2004): Each iteration is 1 to 6 weeks long. Longer iteration is not favorable because you lose the benefits of early, frequent results and feedback. Project planning is about assigning features in each iteration. Early iterations should implement features with high business values or features that are most architecturally significant and risky. In many cases, these two coincide, which

is ideal because architecture can be laid out early, thus reducing risk. If they do not coincide, you have to make a choice. My suggestion is that if the product is large, complex, and long term with relatively low change rates, implementing architecturally significant features first makes the most sense. Otherwise, features with high business values should take precedent. At the end of each iteration, a partial, production quality system is delivered to business areas for testing and feedback. An important point is that this deliverable is not a throwaway prototype; it is a real system that will be extended and enhanced in the later iterations. The feedback shapes and steers forthcoming iterations.

2. Early and frequent developer testing (Beck, 2004), which includes both unit and integration testing: Early and frequent integration testing is especially worth emphasizing because many people believe integration testing is not needed until the last moment of a development cycle. Late integration has more chance to cause integration problems that require a longer time to fix and often cause delay of the timeline. For the Merck project we had at least weekly integration testing at the beginning of each iteration and daily integration testing in the last week of the iteration. Automating test scripts is a very powerful way to increase testing productivity because these test scripts are executed again and again every time the code is changed when code change is made.
3. Communications: A software project is most likely to fail if the project team does not communicate well. Face-to-face discussion is the best communication medium. Ideally, all developers should sit in one office and they should exchange design ideas frequently. A team where people feel comfortable expressing and exchanging ideas freely is a healthy team and is more likely to succeed. The development team should also communicate directly with business analysts and end users. Having a project manager sitting in the middle as a communication buffer is not a good idea because this often causes misinterpretation and lost information lost. The best approach, therefore, is to let the technical team communicate directly with the end users. Developers must be involved in requirement meetings with the end users, because this is the opportunity for them to understand what users want the system to do and ask for clarification. Relying on the project manager to sit in the meeting and then translate the information to developers is not effective. User demo and feedback meetings are the same, and developers have to be involved in these face-to-face meetings directly.
4. Adaptive planning—changing the project plan based on the feedback from each iteration: Please note that it is perfectly OK to either add new features or remove already planned features in the forthcoming iterations. This

process makes sure that the project is always doing what is most important and delivers the highest value to the business at any point in time.

5. Daily project meeting: At this time, each team member updates progress and expresses problems to which the project team needs to react quickly. The problems can be requirement ambiguities, technical difficulties, delays in task completions, and so on. Also, the project manager and the technical lead have to step out and take responsibility for helping the project team to solve these problems.
6. Refactoring (Fowler, 1999): In an agile iterative process, there are no upfront complete requirement and design specifications. Requirement changes are expected to occur due to misunderstandings early in the process and due to feedback after users see the partial system. These changes in the requirements may make the earlier design decision less optimal or obsolete. The development team has two choices. They either do nothing to the design but address requirement changes by adding patches to the code or they refactor the design when changes come to keep the design clean. If the first approach is taken, the design will soon become very bad and the cost of maintenance will grow exponentially and the whole notion of accepting changes becomes unrealistic. Therefore, the agile iterative development process demands the second choice. When changing the design, the developer has to make sure that the working code does not break. The term "refactoring in software" is a technique of changing the internal structure of the code without changing its external behaviors (Fowler, 1999). To do so, the changes should be done in small steps and always be challenged by regression testing. Martin Fowler (1999) has described many refactoring techniques in his well-crafted book. People may think that refactoring has high costs. In fact, the cost of not doing refactoring is much higher because the system will soon require a complete redesign.

A software project is hard to predict. The best you can do is to make sure that the delivered software has the most business value and the highest quality possible at any point in time. The agile iterative process helps you to achieve that.

5.3 TESTING

In many organizations, testing—at least system and integration testing—is not considered the responsibility of developers but the QA group. Although it is a good idea to have an independent QA group to perform a sanity check of

the system before it is deployed to end users, the agile development methodology demands early and frequent developer testing instead of relying on the QA group for most testing activities. Kent Beck (2004) calls this developer accountability.

In most organizations, QA is a shared resource that supports many projects. Having the QA resource available when you need it is not realistic especially if your project is iterative and requires frequent testing. In an iterative process, testing should be done all the time along with coding. The system is not considered done if it is not fully tested by the developers before it gets delivered to QA or end users for further testing. When we talk about delivering a partial system at the end of each iteration, it means the partial system is fully tested and is of production quality—if the business decides to deploy it to production as is, it is ready to be deployed. A misconception is that every iteration has to complete coding of all features that are planned for the iteration but can skip testing if necessary. This is totally wrong. Quite the opposite, the agile iterative process says it is OK to cut features if they cannot all be implemented in the iteration, but what has been implemented has to be of production quality.

The following list shows the types of testing that developers should be responsible for:

1. Unit testing: Every module, class and method, has a unit testing code that is rerun every time when changes are made to the module. If a bug is found not as a result of unit testing, it becomes a new test case and is rerun in the future. The open source XUnit (JUnit, NUnit, HttpUnit) family of testing tools can be used to automate unit testing.
2. Integration testing: This should be done daily or at least weekly. Because modules work as a unit does not mean they work together. Every day, the unit tested changes of all developers are checked into the source control, a new build is created and deployed to the test environment, and integration testing is performed to make sure the changes made do not break integration points.
3. Functional testing: Make sure the newly built system does what it is supposed to do—bugs are fixed and new features work according to the specifications.
4. System regression testing: Make sure things that already worked before the changes were made still work.

If any of the above testing fails, the developer is notified right away so that the bugs get fixed in the next build. Some open source tools can help to automate the testing. CruiseControl is a tool that automatically builds, deploys, and runs regression testing every time a code is checked into source control and notifies the person who checked in the code should problems be found.

FitNess is an acceptance testing tool that captures requirements as test cases. It scripts the test cases and executes them automatically.

In the chemical informatics development team that I led at Merck, the developers were told to make changes in small steps, test the changes, check them in, and then move on to the next task. At end of every working day at about 4:30 PM, a new build was created that included all changes that were checked in that day and deployed to the test server. The team leader, myself, was responsible for doing integration, functional, and regression testing. This process worked out quite well for us.

CHAPTER 6

UML Modeling

UML is the acronym for the Unified Modeling Language. It is a visual language for modeling requirements, design, and deployment of software systems. The term "Unified" comes from the fact that it unifies the pre-UML notations such as Booch (Booch, 1991, 1994), OMT (Rumbough et al., 1991), and Use Case modeling (Jacobson et al., 1992).

Now in version 2, UML is an OMG standard. For a good introduction and an excellent usage guide, please see Martin Fowler's *UML Distilled* (Fowler, 2003). In fact, Fowler has done such an excellent job that I find this book of less than 200 pages covers most of the usefulness of UML. For detailed instructions about UML, please use *The Unified Modeling Language User Guide* (Booch et al., 1999). Craig Larman's *Applying UML and Patterns* book is a very practical reference to UML and its applications to OOAD (Larman, 2005). Another good UML guide is Scott Ambler's *The Elements of UML Style* (Ambler, 2005).

It is worth pointing out that UML is neither a software development process nor an object design technique. It is merely some graphic notations from which you can create various diagrams that help you to visualize requirements and design ideas. It is a great tool that can help you to capture and analyze requirements. It can also help you to visualize and analyze your design and therefore allows you to more easily apply design principles and techniques to improve the design. It can also be used to communicate design ideas with your colleagues and sometimes customers. What it does not give you is the object design principles and techniques that require a totally different skill set. UML is also process independent in that you can use UML in any software development process whether it is waterfall or iterative.

UML defines a dozen diagrams. I personally find Class Diagram, Sequence Diagram, Activity Diagram, Package Diagram, and Component Diagram the most useful ones. One misunderstanding is that the above diagrams are for

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

modeling the design rather than for requirements. In fact, all of the above except Package and Component diagrams can be used to model both requirements and design. The following diagrams show how UML Sequence Diagram can be used to model both requirements and design. Figure 6.1 shows how a user interacts with a Compound Registration System (CRS) to accomplish compound registration. In the diagram, the CRS is represented by a single box CRS. It does not show the design detail. It does show, however, from a user's perspective, the steps it takes to register a compound using the system and how the system responds to every user action. This is called the System Sequence Diagram (Larman, 2005).

Figure 6.2, on the other hand, is a sequence diagram that shows a sequence of message sending between the objects inside the CRS. This design sequence diagram is the basis for the implementation that is created.

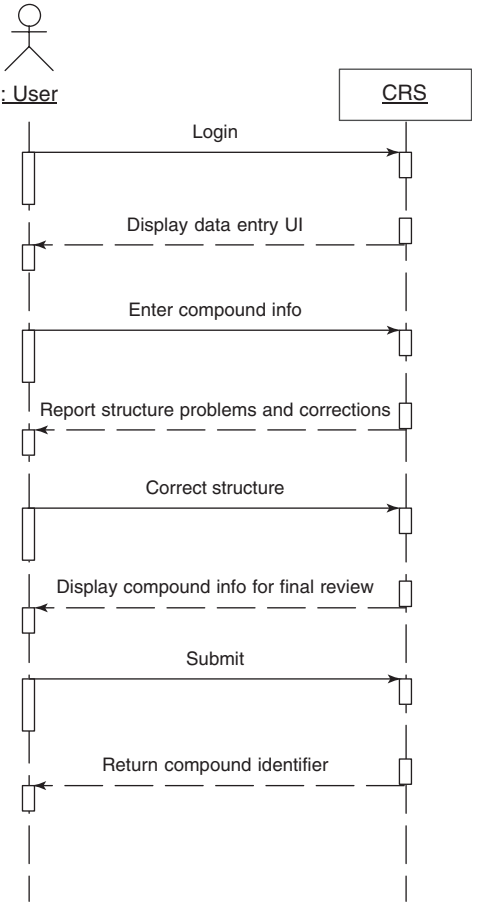


Figure 6.1 A typical CRS compound registration scenario.

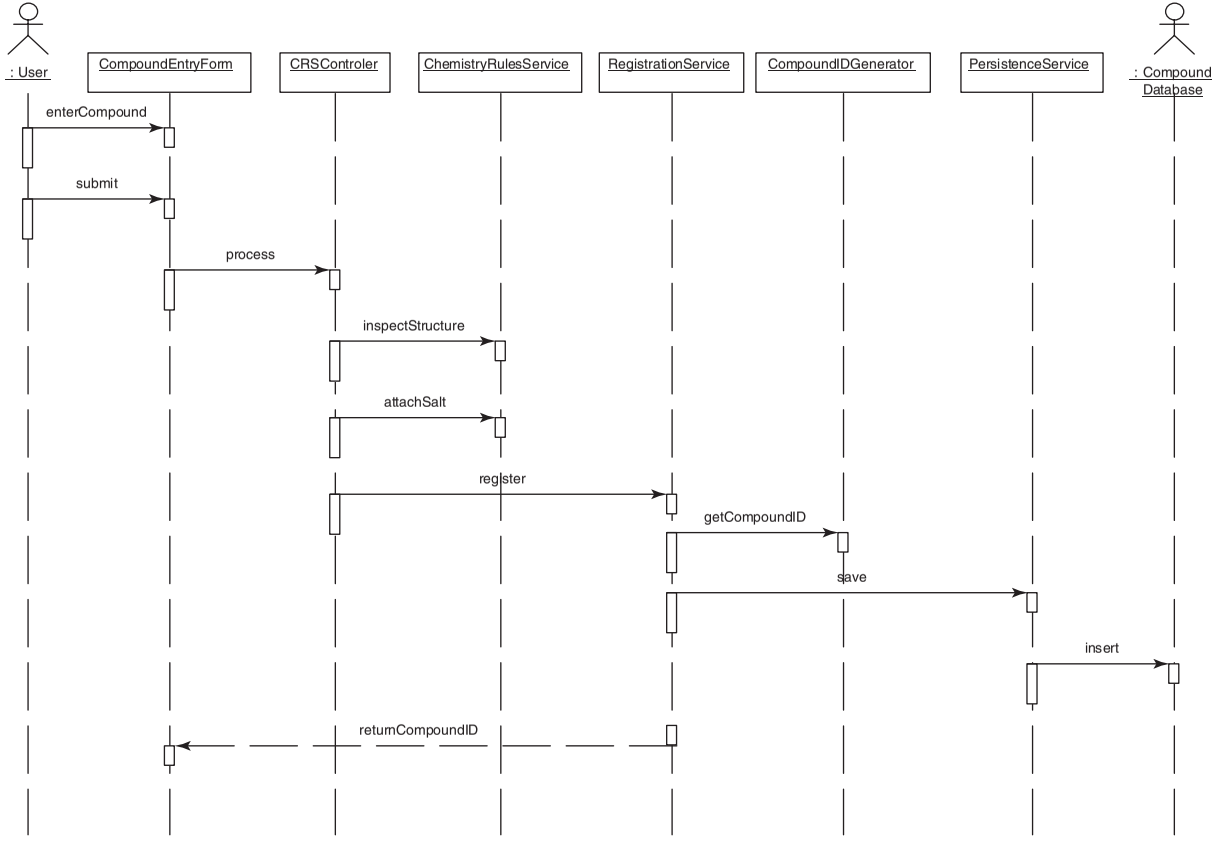


Figure 6.2 A CRS design sequence diagram.

UML diagrams will be used throughout this book.

You may wonder about the use case diagram. In fact, the use case diagram is not as useful as the use case specification, which is a text document. Spending too much time getting the use case diagrams "right" does not provide much value. However, some type of use case diagram can give you a high level idea of what functionality the system is supposed to provide. Figure 6.3 is an example of a use case diagram.

Do not spend too much time trying to get use case diagrams accurate because that is not the point of use case modeling. Use case text, on the other hand, is a very good tool for capturing functional requirements. I often find that ad hoc requirement specifications are vague, lacking details, and not testable. Use case specifications describe scenarios in which the system interacts with a user to fulfill some user goals. It clearly describes inputs and expected outputs, the dialogues between the user and the system, or a sequence of events (interaction steps) between the user and the system. It can also help you to develop test cases. Another advantage of use case specifications is that compared with ad hoc requirement specifications in which information is scattered all over the place, each use case focuses on very specific and related usage scenarios that fit in very well in an iterative development process in which the development team focuses on very specific feature(s). I strongly recommend use case specifications (text) as functional requirement specifications. Cockburn (2001) is one of the best resources about how to write effective use cases. Larman (2005) also has some use case examples. You will also see some examples in Chapter 9 of this book.

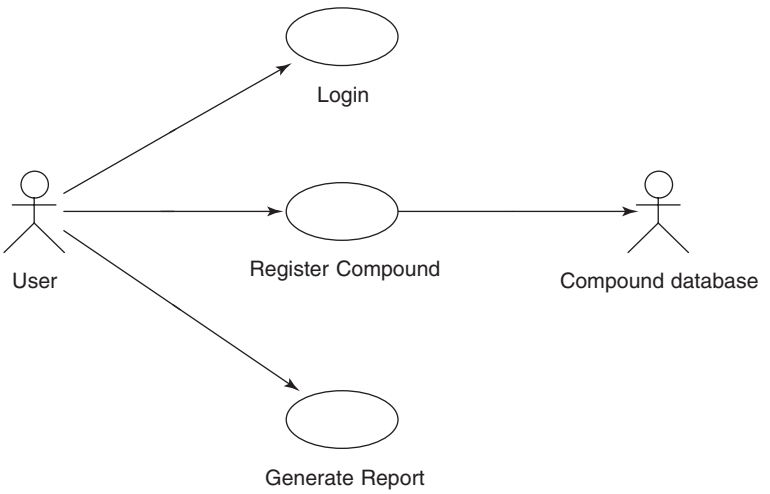


Figure 6.3 CRS use case diagram.

CHAPTER 7

Deployment Architecture

For most organizations, the following three options exist in regard to deployment architecture.

7.1 TWO-TIERED CLIENT–SERVER ARCHITECTURE

In this type of architecture, two computer nodes are connected by a network to do the work. The node that an end user uses is called Client. It provides graphical user interfaces (GUIs) and event handling for user interactions. In many cases, it also has some business logic. This type of client is called rich client (or thick client, fat client). Client machines are typically desktop or notebook PCs with a Microsoft Windows operating system. Some organizations still use Macintosh or Unix workstations, and Linux is an emerging alternative. The backend node in the client–server architecture is the database server. The chemical information database resides in this node. It may also host some business logic that is shared by all clients in a client–server architecture. The business logic on the server node is typically implemented as database triggers and stored procedures. In a chemical information application, a chemical registration application puts compound data into the database, and a querying and browsing application gets the data out of the database and presents the data to end users.

Figure 7.1 illustrates a client–server architecture.

The biggest advantage of client–server architecture is probably its simplicity. Because the client process can be single threaded, you do not need to worry about issues such as thread safety and deadlocks. The reliability requirements are also not as stringent because although crashing or hanging of a single client machine is not desirable, it is not fatal to the overall system as long as the server node is safe.

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

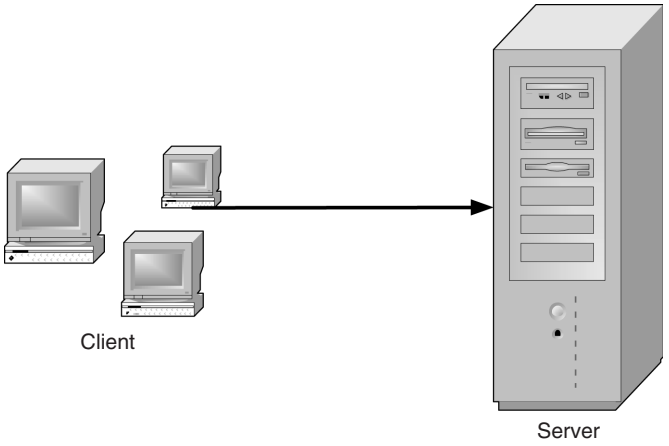


Figure 7.1 A client–server architecture.

The client-server architecture imposes several limitations. First, the computing power (CPU, memory, IO) of the client machine is limited. A designer has to carefully balance what business logic should reside on the client machine versus what business logic should reside on the server machine. Second, the resource utilization is not efficient because every client keeps a connection to the database server. Third, deployment is a big burden because every upgrade requires a new installation on all client machines. Fourth, although it does not happen very often in the chemical information space because Oracle is overwhelmingly dominant, business logic implemented as database stored procedures and triggers is not portable from one database management system (DBMS) to another.

7.2 THREE-TIERED (OR MULTITIERED) ARCHITECTURE

A three-tiered architecture includes one more node between the client and the database server—the middle tier. In a three-tiered architecture, business logic is offloaded from the client and the database server nodes to the middle tier. In fact, you can choose to further distribute the business logic among more than one middle tier node and still call it a three-tiered (or *n*-tiered) architecture because the idea is similar. Note that the tiers do not have to be physically separated. You can have both the middle tier server and the database server collocated on the same physical computer but running in different processes with separate memory spaces. Modern hardware architecture can partition a single hardware box into multiple virtually separate computers or domains. Typically, a three-tiered architecture supports a Web-based thin client although it can also work with a rich client.

Figure 7.2 illustrates a three-tiered architecture.

The three-tiered architecture turns the disadvantages of the client–server architecture into its advantages:

- The computing power of the middle tier is virtually unlimited. When it reaches its limit, by simply adding more hardware, the system should get back on track assuming the scalability of the software is good.
- Expensive system resources can be more effectively managed. For example, the middle tier can create a pool of database connections that is shared by all clients. It can also implement a complex data caching capability to boost performance.
- It provides better performance for systems with high user load.
- If a Web-based thin client is used, there is no client deployment needed. The updated system is deployed to the middleware only, and the clients can access it easily with a URL from a Web browser.

These advantages are not free. Three-tiered architecture has many challenges:

- The three-tiered architecture is much more complex than a client–server architecture.
- The infrastructure and development cost of three-tiered architecture is higher than client–server.
- Because the middle tier handles concurrent requests from many user sessions, architects and developers are forced to consider threading issues that are considered difficult for many junior developers.
- Session management has to be implemented to make sure every user state is maintained between method calls.

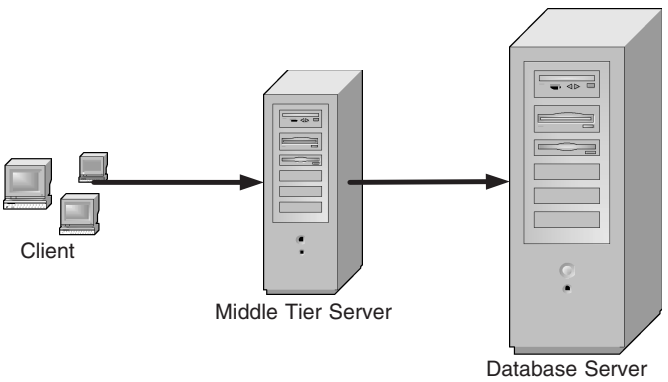


Figure 7.2 A three-tiered architecture.

- Because more networked layers are introduced, performance may suffer if the architecture is not designed properly.

A more robust solution is to introduce a load balancer between the clients and a farm of identical middle tier servers. The load balancer routes requests to one of the middle tier servers based on the load and netweork topology. If one middle tier server fails, it can reroute the request to another to prevent single point of failure (fail-over). This architecture requires that the servers in the middle tier farm synchronize user sessions with each other in case one has to take over a client from another.

Figure 7.3 illustrates a three-tiered architecture with a load balancer.

By the same token, the database server can also be clustered, which is not illustrated in Figure 7.3.

Both client–server and three-tired architectures can be developed using either Java or .NET technology, although .NET is a more natural choice for a rich client on Windows-based client PCs. It is also possible to take a hybrid approach—develop rich client in .NET but business logic in Java, and integrate both using SOAP-based Web services. MDL's Isentris, for example, takes the hybrid approach.

Which architecture to choose should be determined on a case-by-case basis. No single architecture fits all situations. Each organization should carefully evaluate the pros and cons of each architecture and make decisions based on requirements, budget, technical expertise, infrastructure, technology standards, and so on.

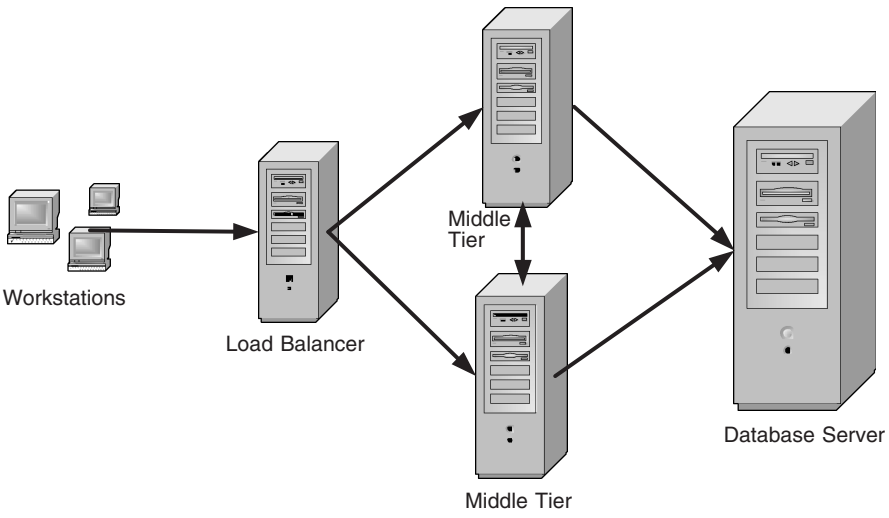


Figure 7.3 A three-tiered architecture with a load balancer.

This book focuses on a three-tiered architecture, on which Enterprise Java is based.

7.3 SERVICE-ORIENTED ARCHITECTURE

Service-oriented architecture (SOA) is a hot topic these days and is considered by many people to be the enterprise computing framework of the future. In SOA, each software unit runs on a piece of hardware as a service that can be called by many different consumers. For example, a compound registration service can be called by a library enumeration tool and a chemistry e-notebook to fulfill compound registration tasks. The most popular SOA is Web services that are based on HTTP and XML or SOAP standards although SOA as a concept has existed for awhile and is not limited to Web services. SOA has a lot of advantages, the most important of which is code reuse and improved productivity. However, it also presents a lot of challenges.

There are two types of code reuse; one is code libraries that are deployed with every single application that uses them. Another is services that are deployed and run independently somewhere in the network. The notion of designing with SOA in mind is very important. Service is not about simply wrapping a piece of code that is designed for a standalone application with a service API and exposing it to the public. It represents unique requirements and challenges such as the following:

- Security: Transparent authentication and authorization across application boundaries.
- Performance: In general, a service performs more poorly than an in-process method call due to network latency and bandwidth constraints. To make things worse, a service may call other services to fulfill its responsibilities—a chain of services. Performance has to be a design consideration throughout the development cycle.
- Availability and scalability: A service does not know how many clients will use it and when. It has to be very resource efficient and able to handle unpredictable user load. Caching and resource pooling should be used more widely in these systems. It should also be deployed into a clustered environment. There also must be a monitoring and alerting mechanism to make sure it is always in a healthy state.
- Make sure distributed transactions are atomic.
- A service also poses challenges to its clients: The clients have to be able to recover themselves if the service or network goes down.

All of these requirements require diligent design and rigorous testing. SOA will not be a reality without these issues being addressed.

CHAPTER 8

Software Architecture

Modern information systems provide many layers of abstractions to ease the development and increase the portability of the software. The operating system is an abstraction layer that hides hardware architecture. For example, both Windows and Linux can run on Intel and AMD hardware, and application developers usually do not care what underlining hardware is being used. Virtual machines are a layer of abstraction that hides operating systems. For example, Java Virtual Machines are available for almost all kinds of operating systems so that in most cases Java developers can write portable code without even thinking about the underlining operating systems that his/her code has to run on. Microsoft Common Language Runtime (CLR) is a similar concept, although its implementations on operating systems other than Windows remains to be seen. At least CLR is language independent in that you can write your code in any language that is supported by .NET, and they can call each other and interoperate seamlessly within the CLR. Application server specifications are another layer of abstraction with containers in which business components are deployed. For example, if you adhere to J2EE standard APIs, your J2EE components should be easily portable from one application server implementation (Weblogic) to another (JBoss) or vice versa.

These abstraction layers offer tremendous benefits to the software development process with reduced development complexity and costs and increased productivity. Application server platforms and blueprints also provide software development frameworks to help the software fit into specific architecture patterns. One of the most commonly adopted software architecture patterns for enterprise systems is the layered architecture (Buschmann et al., 1996; Fowler, 2003a). It is also the heart of the J2EE blueprint (Alur et al., 2003).

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

The Layered Architectural Pattern: This helps to structure applications that can be decomposed into groups of subtasks, in which each group of subtasks is at a particular level of abstraction.

In a layered architecture, the software system is divided into layers of subsystems in which lower layers provide services to upper layers. A classic example of the layered architecture is the ISO's network protocol (Figure 8.1).

Please note that layers and tiers are two different concepts. Tiers mean the physical separation of subsystems—each subsystem runs on a different hardware or the same hardware but in different processes. In a multitiered system, the interaction between the subsystems is accomplished through remote procedure calls (RPCs). Any RPC involves network overhead and therefore has a performance penalty whether the remote procedure is on a separate hardware or on the same physical hardware but in a different process. Layers, on the other hand, are logical separations of the subsystems. Each layer can run on a different physical tier, or all layers can run on a single tier. The purpose of physical tiers is to leverage distributed hardware resources or to reuse a piece of software that is deployed on a different hardware that your system wants to leverage. The purpose of layered software architecture is to separate the system into highly cohesive and loosely coupled modules (see Chapter 2 for software development principles).

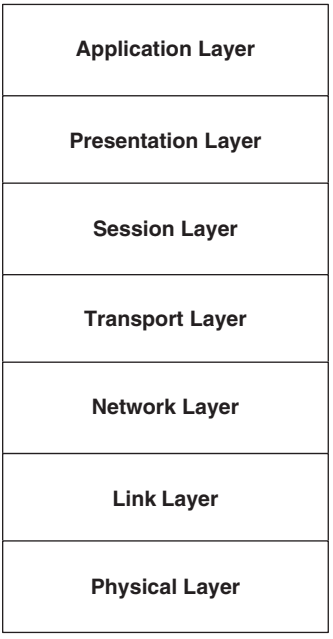


Figure 8.1 ISO network reference model.

Figure 8.2 is a typical layered architecture in a Web application. It also shows how the layers are typically distributed among the physical tiers.

From the top, the client layer resides on an end user's desktop, laptop, or handheld device, which is typically, but not limited to, a Windows PC with a Web browser. Usually a Web-based client layer is called thin client, which is lightweight. The programs that run inside the browser are typically JavaScript, VB Script, Java Applets, ActiveX, or Web browser plug-in. Using a rich client such as .NET or Java Swing is another choice, although this book focuses on a Web-based architecture.

The next three layers reside on an application middleware server, although in some systems, there is a further physical separation between the presentation layer, which runs on a different hardware from the domain and data access layers. If EJB is used in a J2EE application, the presentation layer runs on a Web container and the domain layer runs on an EJB container. With the EJB local interface in J2EE 1.3, the separation becomes unnecessary, which eliminates the network overhead between the two.

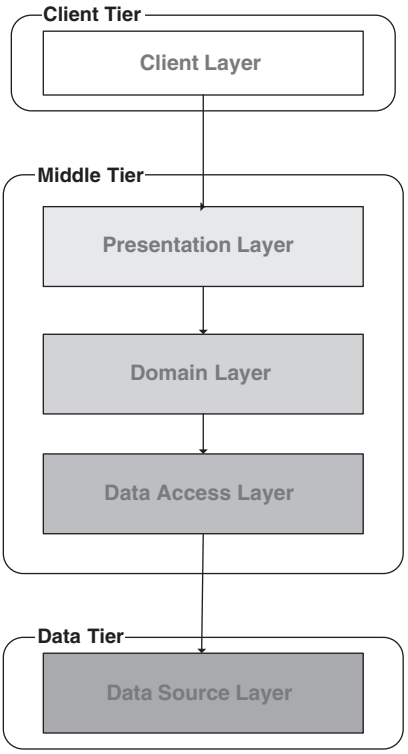


Figure 8.2 A layered architecture in a Web application and how typically these layers are distributed among the three physical tiers.

Although the J2EE application usually implies a Java Servlet, JSP, and Enterprise Java Beans (EJB) based Web application, it does not mandate the use of EJB. In fact, not using EJB gives you some performance advantages and some programming freedoms. On the other hand, EJB, if used effectively, can ease the development and deployment efforts because the EJB container provides a lot of low-level services to allow you to focus on business logic. However, being an effective EJB developer does not mean just understanding the APIs. You have to understand how the EJB container works to write robust and fast EJB objects. For example, Stateful Session Beans are more expensive than Stateless Session Beans and should be avoided when possible. Entity Beans are far more expensive than session beans and therefore should be used to represent "first class" entities (e.g., Employee) in the database. Dependent objects should be used to represent "second class" entities (e.g., Address).

The presentation layer is responsible for receiving and "interpreting" requests from the client layer, delegating the request to the domain layer, and generating and presenting responses back to the client layer. Please note that the presentation layer should not actually process the requests. It should delegate the requests to the domain layer. This separation of the responsibilities increases the cohesion of each layer and makes changes easier—a single domain layer can support multiple flavors of presentation layers and vice versa.

The business layer (or domain layer) is the center of the system that does the real work. It implements all business logic and workflows. In J2EE, EJB can be used to implement the Business layer. However, you can also use Plain Old Java Object (POJO) with an object-relational mapping tool or direct JDBC API to do the job.

The data access layer (or data persistence layer) encapsulates interactions (select, insert, update, and delete) with the backend databases. The purpose of this layer is to hide database schemas from the business objects in the domain layer so that when the database schemas change, the domain layer is not affected. The data access layer can be implemented in Entity Bean or POJO using the JDBC API. Entity Bean is not recommended for several reasons. First, as was discussed, there is a huge performance impact when Entity Beans are used. Second, if you use MDL RCG Oracle Gateway, you will not be able to use Container Managed Persistence (CMP). In J2EE 1.3 and 1.4, the Enterprise Java Beans Query Language (EJB QL) does not support MDLDirect operators. I do not know if it ever will. Not being able to use CPM inhibits one of the biggest advantages of Entity Bean. An alternative is to use an object-relational mapping tool such as Hibernate. In the chemical information domain, there is the MDL Isentris Integrated Data Source Framework. It does similar work that other object-relational mapping tools

do, but it has a high license fee. Hibernate, on the other hand, is an open source tool and is free.

The very bottom layer in the architecture is the data storage layer. This is where the compound data are stored when a registration or update is committed. Almost all chemistry database vendors use Oracle as the data storage DBMS, including MDL, Daylight, Accelrys, Tripos, and CambridgeSoft. They provide some kind of chemistry data cartridge that allows you to query, insert, update, and delete compound data using direct SQL. I have experience using MDL's MDLDirect Data Cartridge version 2.0 with the MDL RCG database, and I am very satisfied with it. Storing compound data in an Oracle database allows you to query across chemical and biological data easily, which is a huge advantage.

An obvious benefit of the layered architecture is that you can easily swap out a particular layer and replace it with a different one without impacting the service consumer layer above it provided that the service consumer layer is dependent on the interface of the layer being swapped out rather than its implementation. For example, the chemistry intelligence component resides in the domain layer in Figure 8.2. Assume the component has an implementation independent interface on which the presentation layer is dependent. Today you are using vendor A's implementation of that interface. For some reason (maybe the vendor is going out of business; another vendor has a better implementation or provides a better price; or you have developed a better in-house implementation) you want to replace it with a different implementation; all you have to do is to swap out the component and replace it with the other. To achieve easy plug and play, the higher level layer must be dependent on the abstraction of the lower level layer, not its implementation. This is called The Dependent Inversion Principle (Martin, 2003), which is discussed further in Chapters 10–12.

This book demonstrates how the layered architecture can be used in an enterprise chemical information system.

MDL's new architecture Isentris is based on a layered architecture. It provides services that a standard J2EE application server provides such as session management, object lifecycle management, messaging, object pooling, and object-relational mapping. It also provides chemical informatics functionality such as chemistry rules, compound registration, and a standard query language for both chemical structure and alpha-numeric data. In my view, Isentris is still young and needs some time to become mature. It is also not cheap compared with J2EE application server products such as BEA Weblogic and IBM Websphere. However, if your organization does not have J2EE or .NET expertise in-house, it is worth considering as a post-ISIS architecture.

There are other architecture patterns, one of which is Pipe and Filter (Buschmann et al., 1996). Pipeline Pilot of SciTegic (now part of Accelrys) is a good application of the Pipe and Filter Pattern and is widely used in the chemical information domain.

Pipe and Filter Pattern: Data flows between the filters via pipes, and the filters apply some logic to the data so that the data that flows out from a filter is the data needed by the next filter.

CHAPTER 9

A Case Study: Develop a Chemical Registration System (CRS)

Chemical registration is a task that all pharmaceutical and chemical companies have to fulfill in their chemical informatics systems. A robust chemical registration system makes sure that the intellectual properties a research organization produces are preserved and can later be queried and retrieved along with other data.

Chemical registration is about storing chemical entities into permanent data storage. Typically, the chemical registration process involves the following steps:

- Structure editing: Structure is properly drawn and entered into the system, which can be done using commercial chemical structure editing tools such as MDL ISISDraw (it has a Web browser plug-in, Chime), MDLDraw, which is aimed at replacing ISISDraw by MDL, and CambridgeSoft's ChemDraw.
- Structure quality checking: Make sure the entered structure is compliant with a set of business conventions from your organization such as stereo chemistry and sprout hydrogen. These conventions can be coded in commercially available chemistry intelligent software such as MDL Cheshire and Accelrys Accord.
- Property calculations: Molecular weight, molecular formula, pKa, and so on.
- Enter ancillary data: Usually notebook information, chemical synthesis methods, the research project for which the compound is synthesized or purchased, and so on. These are alpha-numeric data and are easy to handle.
- Uniqueness checking and sample identifier generation: Although different organizations have different rules of sample identifier generation, a

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

sample identifier usually consists of two parts. One is the compound identifier (compound id)—all samples of the same compound structure should have the same compound id. The other part is called lot or batch number, which identifies the physical sample of different synthesis or acquisition of the same structure because a structure can be synthesized or acquired multiple times (the source of a sample is usually identified by a prefix in the sample identifier). Please note that the sample id we are talking about is different from the primary key of a record in the database. The primary key is usually internal to the DBMS, generated from a sequence and hidden from the end user. No business rules are associated with it. The sample identifier, on the other hand, is generated with a set of business rules and, once generated, is associated with some business meanings.

- Data persistence: The compound data and the sample identifier are inserted into the database and are ready to be searched.

There are three types of data that are involved in the chemical registration process:

- Primary (and the most important) data: The chemical structure.
- Derivative data: Molecular weight, molecular formula, pKa, and so on. These are calculated from the structure and rely on the structure being correctly entered by the user and represented by the system.
- Ancillary data: Notebook information, synthesis methods, purpose of the compound, and so on.

As such, any chemical registration system should be structure centric. It must assist the user to get the structure right—make sure the structure is compliant with all business conventions and salts are added correctly and charge balanced. The whole process of structure handling must be optimized to be as efficient, fast, and accurate as possible.

9.1 USE CASE MODELING

This section presents use case modeling for the CRS. As mentioned, you should spend most of your time getting the use case text right rather than the diagram. Figure 9.1 is a use case diagram for CRS. It is shown as an example and not meant to be complete.

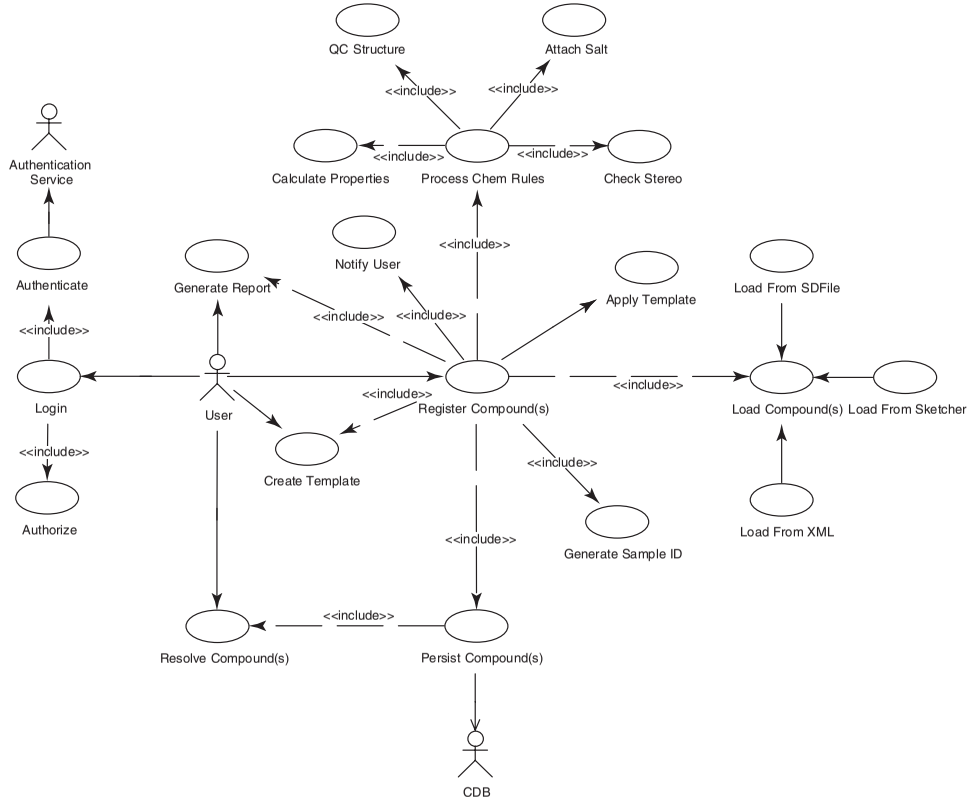


Figure 9.1 Use case diagram of CRS.

Each oval in the diagram represents a use case. Usually a use case is an end-to-end process of a specific usage scenario. This diagram looks too granular in that respect. It has many *include* relationships (Fowler, 2003b). Usually *include* relationships can be used to separate a complicated step in a use case or to represent a step that is used by many use cases to eliminate duplications. For instance, a registration template contains common data that can be applied to many registrations again and again. It can be created before registration starts or during a registration process. In other words, a Create Template use case can be part of the Compound Registration use case or can be a standalone use case. Another example is the Process Chemistry Rules use case, which is very complex and includes other sub-use cases. Another type of relationship in the diagram is **generalization**—a line with a triangle at the end. To register a compound, the compound data have to be loaded into the CRS. Compound data can be loaded from different sources—an SD File, an XML file (MDL is working on an XML schema XDFile to replace the SDFile or RDFile), or a sketcher. A Load Compound(s) use case is a generalization of Load From SDFile, Load From XML File, and Load From Sketcher.

9.2 ITERATION PLANNING

In an agile iterative development framework, project planning is about iteration planning—what features are implemented in what iteration based on priority of business values, risks, and complexity. In some iterative process such as XP, there is not even an upfront iteration planning. It is not until right before the iteration starts that the project team decides what features go into that iteration and how long the iteration lasts (usually 1–2 weeks). In a large project where cost and timeline estimate are required, iteration planning is necessary. However, the iteration plan is just a rough estimate and is subject to change based on how early iterations go and user feedback after each iteration. This process is called adaptive planning (Larman, 2004). The planning should become more and more accurate as the project moves forward and the project team learns more. It is perfectly OK to admit that early interation plans were not accurate; and it is still far better than the waterfall approach, where the upfront project plan is never met and therefore useless. There are hardly any middle-and large-size waterfall projects that do not experience delay and overspending (Larman, 2004). The biggest problem with waterfall is that the project team does not know the project plan is not met until very late so there is no chance to mitigate the effects.

We said that iteration planning should be based on business values, risks, and complexity. Using our case study, CRS as an example, here is a

sample of a high-level iteration plan (based on a team of six full-time developers):

Iteration	Features	Time Frame	Business Value	Complexity	Risk
1	User login and register single compound (no salt) with structure convention checking	6 weeks	High	High	High
2	Register a list of compounds (could be a library) plus salt handling	4 weeks	High	High	Medium
3	Resolve compound(s)	2 weeks	High	Medium	Medium
4	Support SD file and XML file as input	1 week	High	Low	Low
5	Create and apply template	2 weeks	Medium	Low	Low
6	User notification and reporting	1 week	Medium	Low	Low

Please note that in an agile process, the initial iteration plan may not be the actual path the project takes. At the end of each iteration, the plan should be revisited based on feedback and requirement/priority changes and adjusted accordingly—that is, adaptive planning (Larman, 2004). At every point of time during the project, the project team must make sure it is doing the most important things. Please also note that the time frame of each iteration is not changeable after the iteration starts. What can be changed is the number of features being implemented by the iteration. The project team can reduce the scope of the iteration but has to make sure what is delivered at the end of the iteration must be of production quality.

9.3 USE CASE SPECIFICATION

Enough has been said about the use case diagram and iteration planning. Now let's look at how to write use case specifications—text specification of use cases.

The following use case specification is for the Register Compound(s) Use Case. This book is not about how to write use cases. For more detailed discussion about how to write effective uses case specifications, please see Cockburn (2001) and Larman (2005).

Use Case UC1: Register Compound(s)

Scope: CRS.

Level: User goal.

Primary Actor: Chemist.

Preconditions: System has to be available. Chemist workstation has to be connected to the network. Chemist is authenticated and authorized.

Postconditions: Compound data are saved into the database. Sample identifiers are generated. Compounds are retrievable from the database.

Brief Description: This use case allows a Chemist to register single or multiple compounds interactively (not start from an SD File). At the end of the use case, the compounds are registered into the database and the Chemist receives sample identifiers for each compound. The Chemist must be authenticated and authorized to access this use case.

Basic Flow

1. **Chemist** starts CRS to register **compound**(s).
2. **System** allows the Chemist to input compound **structure**(s).
3. Chemist inputs a structure by either drawing the structure from scratch or importing the structure from a **molfile**. If the compound has salt(s), the Chemist also specifies salt type(s) from a pick list and their coefficient(s) (which defaults to 1 and must be a positive integer). The salt(s) can also be included in the structure that is drawn or in the molfile that is imported.
4. System decomposes the structure into **fragments** and displays them to the Chemist. System displays salt structures that were picked, drawn, or imported in step 3 as separate fragments.
5. Chemist verifies each fragment's **role (parent** or **salt)** and coefficient. Chemist updates the roles and coefficients if necessary.
6. System performs quality checking (QC) on the parent structure using a set of **chemistry business rules** (see QC Structures Use Case Spec). If the structure is corrected by the QC process, System displays the structure in its original form and the corrected form side-by-side and highlights what has been changed. If the structure requires manual correction, the System displays the structure and highlights what must be corrected. Otherwise, the structure passes QC.

7. Chemist manually corrects the structure.

Repeat steps 6 and 7 until the structure passes QC.

8. System attaches salt(s) to the parent structure (see Attach Salt Use Case Spec) and displays the structure with salt(s) attached.

Repeat steps 3 to 8 until all structures are entered, pass QC, and salt(s) attached.

9. Chemist enters ancillary data such as **notebook** info, **research project** info, **formulation** info, and **distribution** info. These data can be entered either manually or by applying a template. Chemist submits the compounds to register them.
10. System makes sure all required data are entered (see Compound Registration Required and Optional Data Supplement). System performs uniqueness check against the database (see Uniqueness Check Business Rules Supplement). If the structure is unique, System registers the compound with a new **sample identifier** (see Sample Identifier Generation Rules Supplement). If it is not unique, System displays the compound to be registered and the hit compounds from uniqueness search and prompts the Chemist to resolve (see Resolve Compounds Use Case Spec).
11. Chemist resolves the compound.

System repeats steps 10 and 11 until all structures are registered.

12. System displays the sample identifiers.

Alternative Flows

2 – 9a. At any point between steps 2 and 9, Chemist decides to abort the **registration**. System aborts the registration. System does not save the data.

3 – 9a. At any point between steps 6 and 9, Chemist decides to save the data and come back later. System saves the data and aborts the registration.

4a. Structure has only one fragment. Go to step 6.

6a. Structure passes QC. Go to step 8.

8a. Some salt(s) cannot be attached. System warns the Chemist.

8b. Structure requires salt to be charge balanced, but salt is not specified in steps 3 and 5. System flags an error.

9a. Chemist wants to add more structures. Repeat steps 2 and 9 until all structures are added. All previously entered compounds are preserved.

9b. Chemist chooses some structures to be excluded from the registration. System removes the structures from the registration.

9c. Chemist further modifies a structure. Repeat steps 4 to 8.

9d. Chemist submits the registration in an asynchronous mode. System reports that the registration is being processed. System processes the

registration in the background. When the registration finishes, System sends a message (e.g., e-mail) to the Chemist reporting registration outcome—sample identifiers for those that are registered, errors for those that failed, and tracking numbers for those that are not structurally unique and require resolution.

10a. System finds missing required data. System prompts the user to enter the data.

10b. Registration fails. System displays an error message.

Special Requirements: Chemist's workstation has to have at least 256 MB of memory and 800 MHz CPU. The supported operating system is Microsoft Windows XP Professional with Internet Explore 6.0 Web browser, MDL ISISDraw 2.5, and MDL ChimePro 2.6 SP6 Plug-in.

The following outline is a brief description of what each section is about (Cockburn, 2001; Larman, 2005).

Scope: The system or business process for this use case. In our example, we are describing usage scenarios of a software system under development—CRS.

Level: A use case can be a user goal level or subfunction level. A user goal level use case describes a set of scenarios for fulfilling some kind of user goals. Register Compound(s) is a user goal, and therefore, it is a user goal level use case. A subfunction level use case describes substeps for fulfilling some user goals. In Figure 9.1, Process Chemistry Rules is an example of a subfunction use case.

Primary Actor: The primary user (or user role) of the system whose goal must be fulfilled.

Preconditions: Conditions that have to be satisfied before the use case starts.

Postconditions: Conditions that have to be satisfied after the success scenario of the use case is completed.

Basic Flow: A typical success path of the use case; at the end of the flow, the user goals for which this use case are fulfilled.

Alternative Flows: Other paths of the use case. They include both successes and failures.

Special Requirements: Nonfunctional requirements such as system requirements, performance requirements, and load requirements. Some of these requirements can also be included in supplements.

The above use case text has reference points that lead to other user goal and subfunction use cases. These points are the ***includes*** in the use case diagram (Figure 9.1).

There is no standard format for use case specifications. The above is the Alistair Cockburn format, which can be found at his website — *alistair.cockburn.us* or in his book (Cockburn, 2001). The template is available for download from the website. Another popular format is the IBM Rational RUP format that can be downloaded from http://www 128.ibm.com/developerworks/ rational/library/4152.html.

Please note that use case modeling is not associated with any development process. It is even independent of object-oriented technologies. It is a generic tool for capturing functional requirements. I have used use case specifications in ISIS PL projects, and it worked perfectly.

In an object-oriented world, in addition to capturing functional requirements, use case specifications can be used as a starting point for developing a domain object model, which is discussed in the next chapter.

Other types of requirements, usually nonfunctional, may not be captured by use case specifications such as performance, scalability, availability, usability, security, and software–hardware constraints. These requirements can be included in the supplementary specifications. Please see Larman (2005) for examples.

Use case specifications can be visualized using UML Activity Diagrams and System Sequence Diagrams (Larman, 2005). I always find a graphic model easy to understand and communicate. However, these visualized models should be used as supplements, not as a replacement for use case specifications in text.

Figure 9.2 is a system sequence diagram (SSD) of the above use case. Figure 9.3, on the other hand, is the activity diagram of the above use case. SSD is an excellent way of illustrating actor–system dialogue. However, it is cumbersome to show both Basic Flow and Alternative Flows in one SSD. The activity diagram, on the other hand, can be used to easily visualize both Basic Flow and Alternative Flows in one diagram. Which one to use is a matter of personal preference. I find both of them useful in my work, and therefore, the SSD and activity diagram often complement each other for me.

9.4 WHEN TO DEVELOP USE CASE SPECIFICATIONS

In an agile iterative process, not all use case specifications need to be fully developed before the design and implementation starts. Usually, a scope and a vision document are developed with a list of features that should be included in the system, including a brief description of each feature as in the Brief Description section of the above use case specification. These features are prioritized and planned according to their business values, complexity, and architectural significance (please see Chapter 5). At the beginning of an

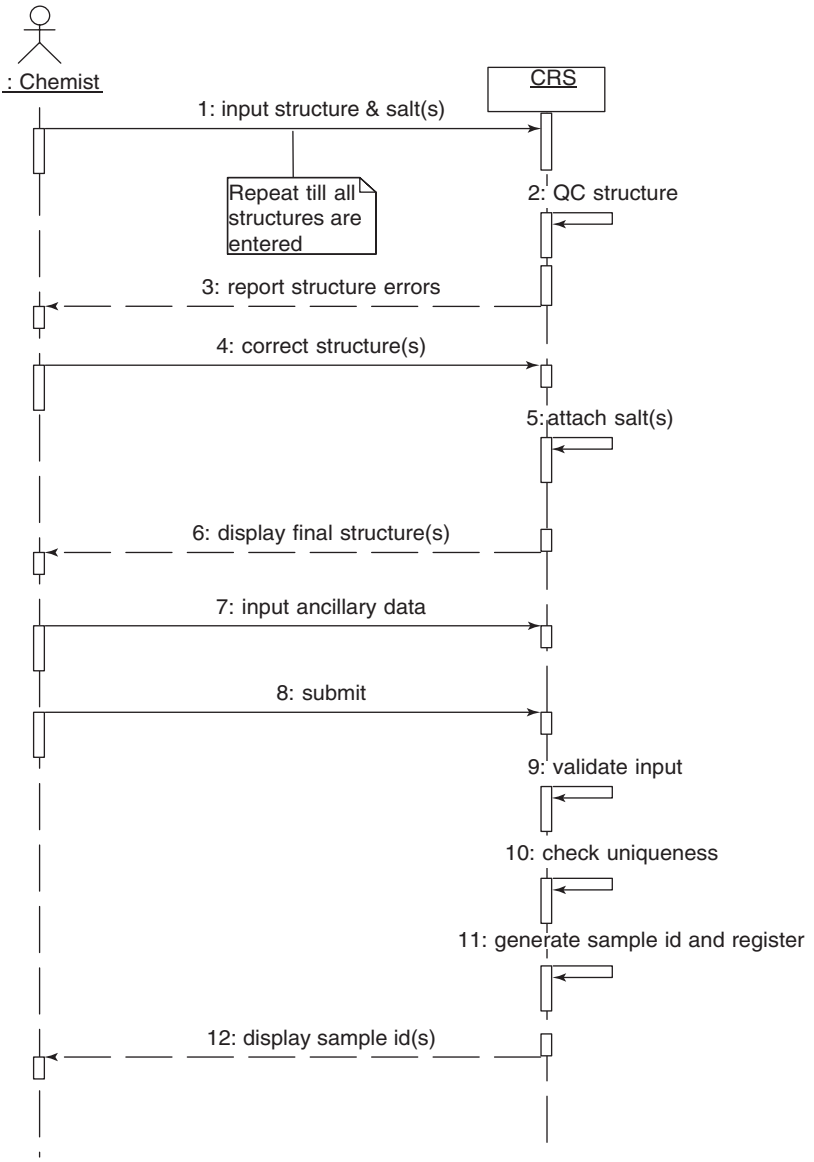


Figure 9.2 The SSD of the Register Compound(s) Use Case.

iteration, the features being implemented by the iteration are analyzed and use case specifications for those features are developed.

Never expect that use case specifications are complete and accurate and do not change before the design and implementation work starts. No matter how much time is spent on requirement analysis, there will be omissions, misunderstandings, and unknowns. Often users may not even know how the

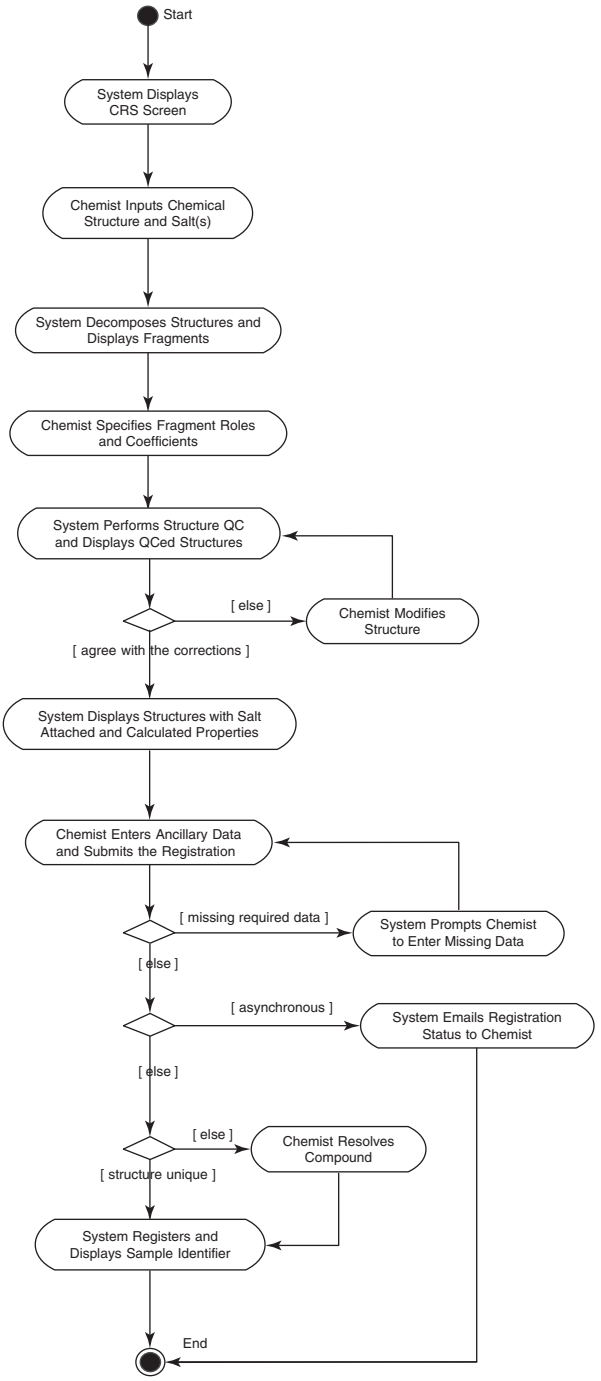


Figure 9.3 The activity diagram of Register Compound(s) Use Case.

system should work before they see a prototype or a partial system. Therefore, do not overengineer use case specifications upfront. Keep in mind that use case specifications provide enough information to get the development work started. New discoveries will emerge during the development and feedback process. On the other hand, not spending any time on requirement analysis at the beginning of the iteration is also not a good practice. It is the upfront analysis combined with ongoing feedback that makes sure the system meets business needs.

CHAPTER 10

A Chemical Informatics Domain Analysis Object Model

Use case specifications document functional requirements. The next step is to design the partial system that the current iteration is supposed to deliver. The gap between requirements and design is not trivial, and a bridge between the two is desired. This bridge is what object-oriented analysis is about. The domain analysis object model is not the final design. However, it provides a starting point for the design process.

A domain object model is a model that describes key domain concepts and their relationships. Many of these concepts come from "tangible" objects in the real world of the problem domain. In the chemical informatics space, these are the "objects" that chemists are dealing with on a daily basis, such as compounds, structures, notebooks, and libraries. The domain analysis model being presented here focuses on those objects that are involved in the compound registration process.

One technique that can help you to capture domain objects is to look for nouns in the use case specifications—those terms in bold in UC1 of Chapter 9. The following are the key domain objects from UC1.

- **Registration:** This is an abstraction that represents a compound registration transaction. A registration can be for a single compound, a group of discrete compounds, or a compound library.
- **Compound**: The object that represents a chemical entity. It has properties such as structure, molweight, molformula, chemical name, stereochemistry description, and toxicity. In an ideal world, the chemical structure should define all other properties.
- **Structure**: This is the most important object in CRS. Getting structures correct is critical to the compound registration process. After being registered, the structures have to be searchable and linkable to biological

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

data for analysis. The registration process has to make sure the structure is represented correctly and the correct sample identifier is generated based on the uniqueness of the structure in the database. Structure can also be used to derive chemical properties such as molweight, molformula, LogP, and pKa. In MDL technology, a structure is represented by a Connection Table. The string representation of the MDL Connection Table is called Molfile—an open standard that is supported by all vendors. In Daylight technology, a structure is represented by a Smiles string. In CambridgeSoft technology, a structure is represented by an XML format—CML.

- **Structure Component** (or fragment): A structure being registered may have one to many fragments. Some of them (at least one) are parent fragments, and some of them are salt fragments.
- **Sample**: A sample is a physically synthesized compound—the product that a chemist produces. A compound represents a structure on paper before it is synthesized through chemical reactions. The compound is instantiated when a physical sample is produced. CRS registers physically synthesized compound samples.
- **Sample Identifier**: Each compound sample has a laboratory identifier. Although companies use different sample identifier formats, they are usually composed of three parts: a base that identifies the parent structure (can be a mixture of multiple parent fragments); a form that identifies the salt form, formulation, or radiolabel; and a lot or batch number that identifies the actual synthesized sample.
- **Library**: A compound library is a group of related compounds synthesized by some parallel synthesis or combinatorial chemistry process.
- **Notebook**: Chemists use a notebook to record information about the synthesis—date, chemist name, notebook page number, and so on.
- **Research Project**: The research project for which the compound is synthesized.
- **Registration Template**: A template stores common information that can be applied to compound registrations multiple times. For example, a chemist may synthesize many compounds or libraries for a research project. These compounds share the same attributes such as project name, chemist, and the assays against which these compounds are screened. The chemist can create a registration template that contains these common attributes and then apply the template when registering these compounds and libraries without having to reenter them again and again.

Figure 10.1 is an analysis domain object model of CRS.

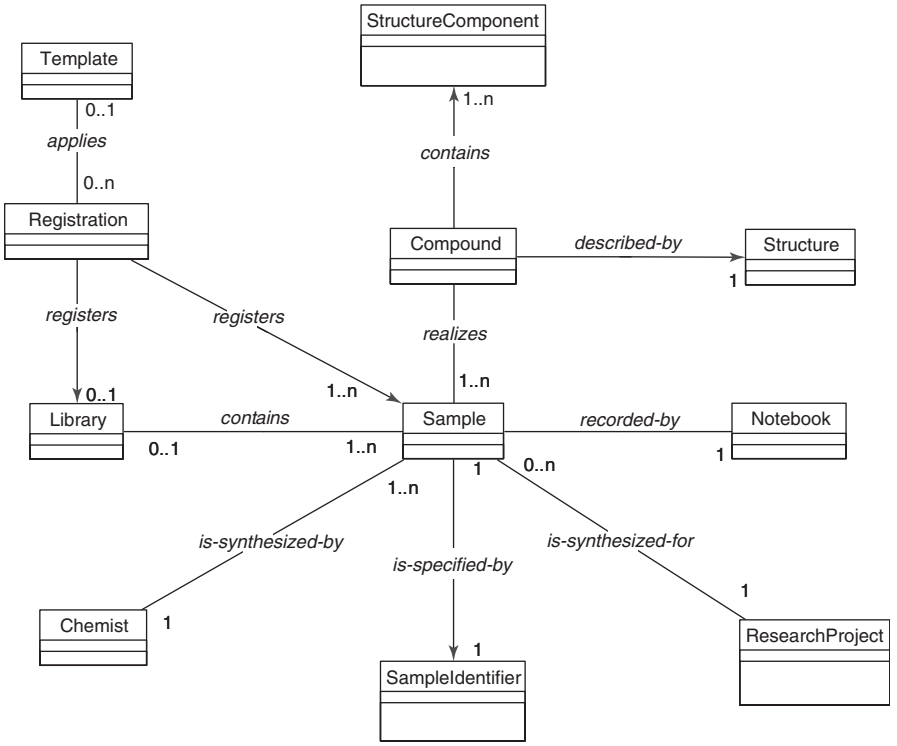


Figure 10.1 An analysis domain object model of CRS.

These conceptual objects are not necessarily software objects that will be in the design model, although many of them may become software objects in the design model. Usually the design model has more objects than in the analysis model when software design principles are applied to introduce more levels of abstractions and in-directions.

Usually the analysis model should include only classes and attributes and not methods. Adding methods to objects is part of the object design process called assigning responsibilities, which is another reason the conceptual model in Figure 10.1 is not a design. The benefit of creating a domain conceptual model is that the model serves as a bridge between the terms we use in the real world and the software objects. Craig Larman (2005) has some guidelines for creating such domain conceptual models.

Figure 10.1 shows conceptual classes (the boxes), the associations (the lines between the boxes), and multiplicity (the numbers or number ranges at the end of associations). Some of these associations have directions that are specified by the navigation arrows, and some of them do not. The arrows indicate that one object has a reference to another object. There are situations where the navigation is bidirectional or cannot be determined at the analysis phase.

For example, it makes sense that the library has a reference to the compound samples it contains. Also, each compound sample should have a reference to the library to which it belongs. Multiplicity specifies the number of objects of one class that can be associated to an object of another class. For example, a compound sample can have only one notebook, and so the multiplicity is 1 at the Notebook end of the Sample–Notebook association. A library can have 1 to many compound samples, and so the multiplicity is 1..n at the Sample end of the Library–Sample association, and so on. If multiplicity is obvious during the analysis phase, then specify it. Otherwise, defer this decision to the design phase.

The domain object analysis phase must be very brief in an iterative process. It should be a $\frac{1}{2}$–1 day activity after the use case specification is developed. Do not expect the analysis model to be complete, accurate, and unchanging. Many decisions can be, and should be, deferred to the design phase.

CHAPTER 11

Presentation Layer

Although the Presentation Layer is not a main focus of this book, several options and some guiding principles are discussed in this chapter.

Although the first generation of chemical information systems used punch cards as the user interface, nowadays almost all of them use some type of graphical user interface (GUI). MDL ISISBase, Visual Basic, .NET, Java Swing, and Web pages are some of the most popular options from which organizations can choose. MDL is developing its next-generation chemical informatics GUI framework called MDLBase, which is based on Microsoft .NET technology. CambridgeSoft and Accelrys GUI tools also use Microsoft technologies. The Web has not been a mainstream GUI framework for chemical information systems, partly due to a lack of Web-based tools in the chemistry domain. At the time of this writing, Merck had successfully developed a Web-based compound registration application using the J2EE technology combined with MDL tools such as MDL Direct, Cheshire, and ChimePro Plug-in.

Web-based systems offer several advantages over rich client: easy to deploy and access, easy to scale (by adding more hardware resources to the application server), and shared computing resources (CPUs, memory, database connections). However, most of the information this book presents is not limited to Web-based applications. It promotes the loosely coupled Presentation Layer and Domain Layer so that business logic can be reused no matter what GUI technology is being used.

A common pattern of GUI design is the Model-View-Controller (MVC) framework. MVC is not about graphic design. It is about separating business logic from graphic widgets so that they are decoupled and can easily be replaced and reused. Avoiding monolithic code that has presentation and business logic intermingled provides a huge advantage. Monolithic code makes the application difficult to change, test, replace, and reuse.

In an MVC framework, Model is the data or business logic that produces the data. View is the graphic components such as windows, text fields, and buttons.

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

Controller is the event handler and dispatcher. In J2EE, there are two possible ways of implementing the MVC framework. In Model 1 MVC, Model can be Java Beans, Plain Old Java Object (POJO), or EJB. JavaServer Pages play the roles of View and Controller. Figure 11.1 is a typical Model 1 MVC.

In Model 2 MVC, JavaServer Pages only play the role of View. The Controller role is played by a Java Servlet. Figure 11.2 is a typical Model 2 MVC.

Perhaps the most popular MVC framework for J2EE Web-based applications is the Apache Struts framework. It is the de facto standard for Java-based Web application development. The key components in the Struts framework are Action Servlet, Action Class, Action Form, and the view objects such as JSP, Velocity Template, and XSLT. The Action Servlet comes with the Struts class library, which can be downloaded from http://struts.apache.org. It plays the Controller role in the Model 2 MVC, which maps the HTTP Request to an Action Class that is responsible for handling the request. The mapping is defined in a configuration file called struts-config.xml. The struts-config.xml configuration file also defines to what view objects (e.g., JSP) the action object should forward. The Action Class is responsible for processing the request and forwarding the flow to the next view object in the flow. The Action Class must be implemented by the developer. However, the Struts class library provides an abstract base action class that all custom action classes extend. This action class hierarchy is an application of the Command Pattern in the GoF book (Gamma et al., 1995). It is recommended that the Action object delegates to a business object (POJO,

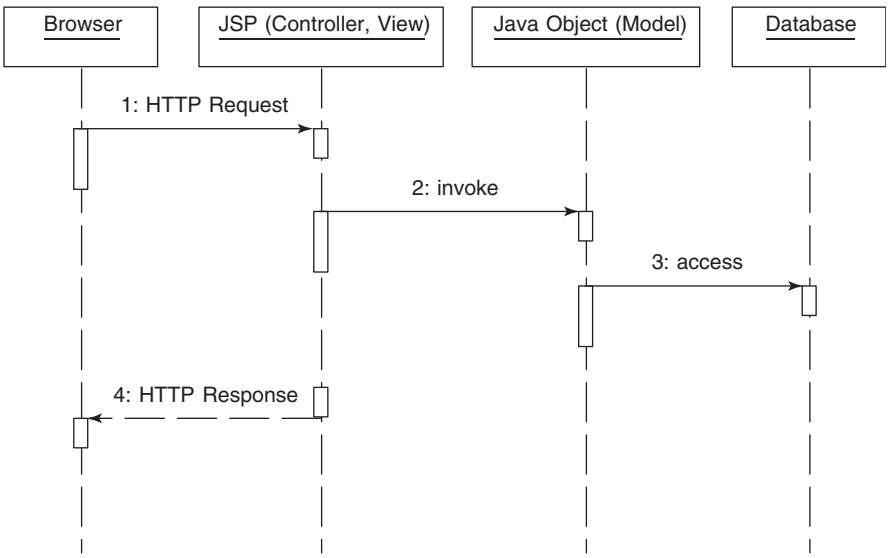


Figure 11.1 A typical Model 1 MVC.

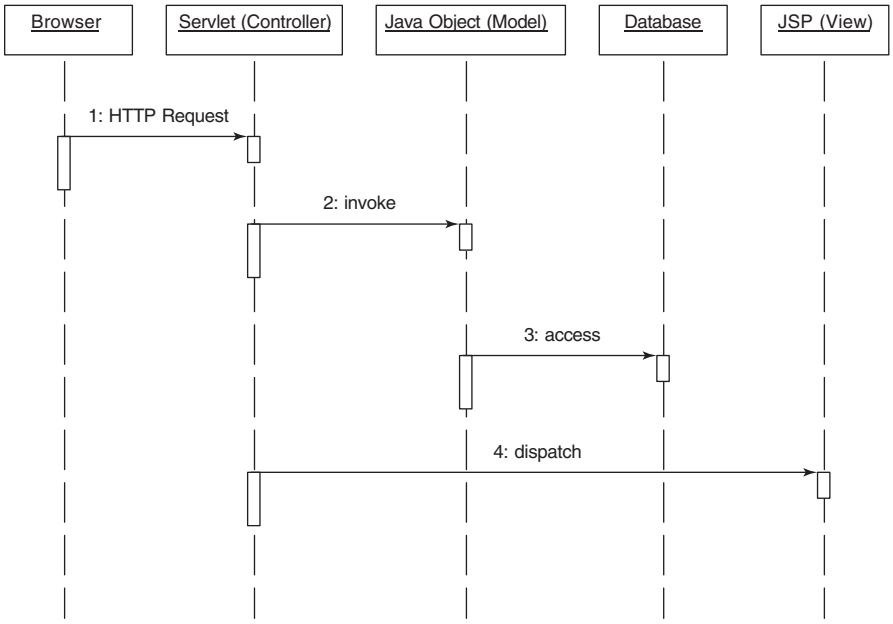


Figure 11.2 A typical Model 2 MVC.

EJB, etc.) to do the actual work, because the action class is a Web component. It takes HTTP Request as an argument into its execute() method. Coding business logic in the action class makes it less reusable and difficult to test. Action Form is a Java Bean that holds user input from the HTML form. It has the setter methods that Struts uses to set its properties from the HTTP Request object. It also has getter methods that the action class uses to access these properties. Figure 11.3 illustrates a hypothetical compound registration transaction using the Struts framework.

The following XML is the struts-config.xml example that shows how to start from a compound registration data entry form - /pages/ Compound Registration. jsp, submit the form - CompoundRegistration.do, map to crs.Compound RegistrationAction which uses action form CompoundRegistrationForm, and if the compound registration succeeds, display /pages/ CompoundRegistration Report.jsp, otherwise display /pages/Error.jsp:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
  <!DOCTYPE struts-config PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
        "http://struts.apache.org/dtds/struts-config_1_2.dtd">
  <struts-config>
    <form-beans>
      <form-bean
```

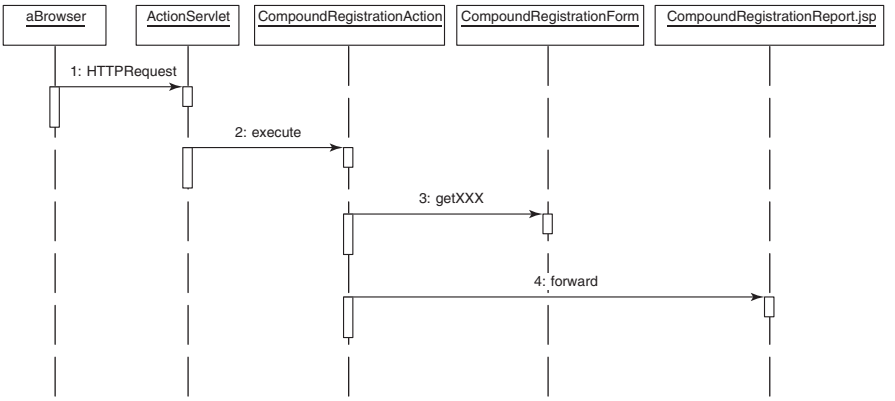


Figure 11.3 Compound registration in the Apache Struts framework.

```
        name="compoundRegistrationForm"
        type="crs.CompoundRegistrationForm"/>
    </form-beans>
    <action-mappings>
      <action
        path="/CompoundRegistration.do"
        type="crs.CompoundRegistrationAction"
        name="CompoundRegistrationForm"
        scope="request"
        validate="true"
        input="/pages/CompoundRegistration.jsp">
        <forward
          name="success"
          path="/pages/CompoundRegistrationReport.jsp"/>
        <forward
          name="failure"
          path="/pages/Error.jsp"/>
      </action>
    </action-mappings>
    <message-resources parameter="resources.application"/>
  </struts-config>
```

Struts provides an excellent framework for implementing Model 2 MVC. It helps developers make the system easier to maintain and change. The caveat is its learning curve and complexity to people who are new to it. However, if your system is large and complex, the overhead will definitely pay off. For a small project with a handful of JSPs and Java objects, Model 1 MVC might be more cost effective. To learn more about Struts, go to http://struts.apache.org. Many books on Struts can be found at http://www.amazon.com. Also see Agarwal (2004).

CHAPTER 12

Business Layer

The Business Layer is in the center of CRS where the core business functions reside. Any business application has a workflow—the steps it takes to accomplish one or many tasks or transactions. Clearly analyzing and understanding the workflow is essential to designing the Business Layer correctly. Use case specifications capture workflow from a user's perspective. The System Activity Diagram is an excellent tool to capture system workflow from a system perspective. Figure 12.1 is the System Activity Diagram (CAD) of CRS.

The registration process starts when a user enters compound data into the system. The input can be either an SD File, an XML File, or manually drawn structures. The system then converts these data into domain objects that can easily be processed by the system. The next few steps involve chemical intelligence. The system decomposes structures into fragments to allow the user to specify parents versus salts and their coefficients (relative counts). The system then combines the parent fragments into the parent structure and performs QC checking to see whether the parent structure is compliant with chemistry business rules. The reason salt structure is not included in the QC process is that salts can be picked from a predefined salt dictionary and the structures can be pre-QCed. After QC, the user can verify and manually correct the structures; in which case, the QC process will kick in again until both the user and the system agree with the structures. The system then attaches salts to the parent structures, calculates the chemical and physical properties, generates sample identifiers, and stores the data into the backend chemical database.

Each activity in Figure 12.1 involves one to many groups of domain objects. Each of these object groups is a cohesive software component or service. From the above CAD, we come up with the following core components in the Business Layer.

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

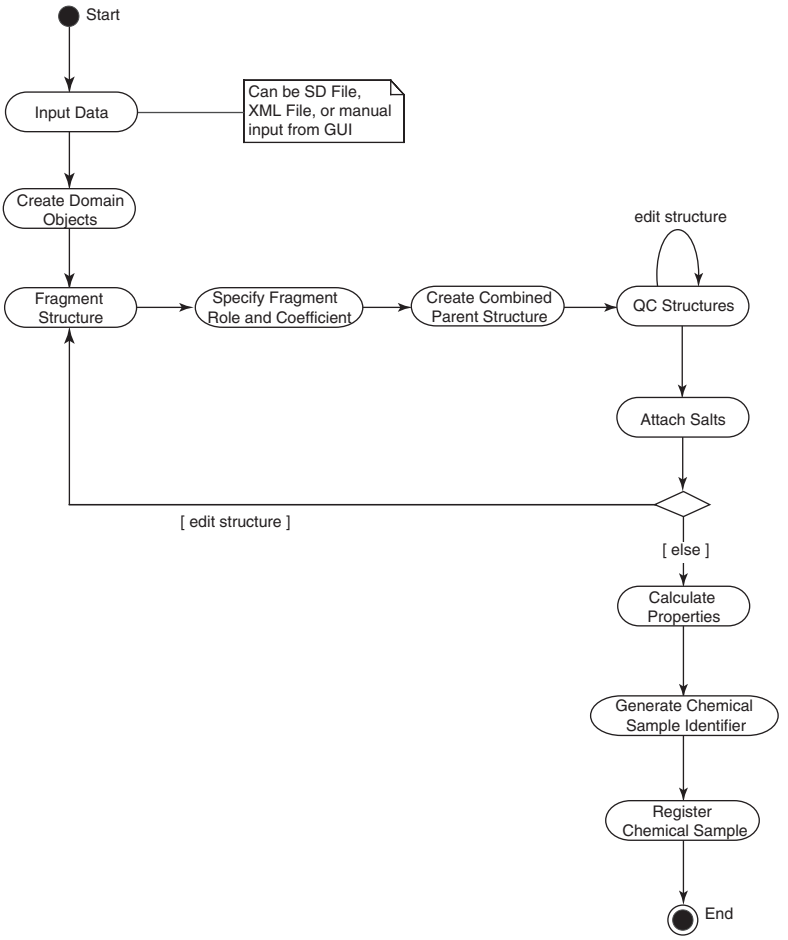


Figure 12.1 The System Activity Diagram (CAD) of CRS.

- **Domain Object**: The design domain objects that are derived from the conceptual model in Chapter 10.
- **Data Binder**: Since we are developing an object-oriented system, if the input is an SD File or an XML File, it must be converted to Java objects for processing efficiency reasons. This conversion is done by the Data Binder objects.
- **Application Controller**: The central controller of the system workflow.
- **Chemistry Intelligence**: Objects that implement chemistry business rules and salt handling logic.
- **Property Calculator**: Objects that are responsible for calculating chemical and physical properties.

- **Chemical Laboratory Sample Identifier (LSI):** An identifier that uniquely identifies a chemical sample. Although the format of sample identifier differs from organization to organization, it usually consists of five parts: a prefix that specifies the sample's source (e.g., synthesized internally or acquired from external sources); a base that uniquely specifics parent structure; a form that indicates whether a chemical sample is a free base or with salt, radiolabeled, or a formulation; a parity bit checksum that is derived from the combination of prefix, base, and form using a check-sum hash algorithm; and a batch or a lot number that identifies the actual physical sample.
- **Registration Service**: Objects that are responsible for uniqueness checking, compound sample identifier generation, and persisting compound data into the chemical database.

Figure 12.2 is the component diagram that shows the above components and their relationships. We will discuss the design of each of these components in this chapter.

12.1 DESIGN BY INTERFACE

A good software designer or architect cares about getting the interfaces right first and foremost before thinking about implementations. Jumping right into implementation details without carefully designing the interfaces first often leads to a rigid and hard-to-extend design for the reasons that were presented

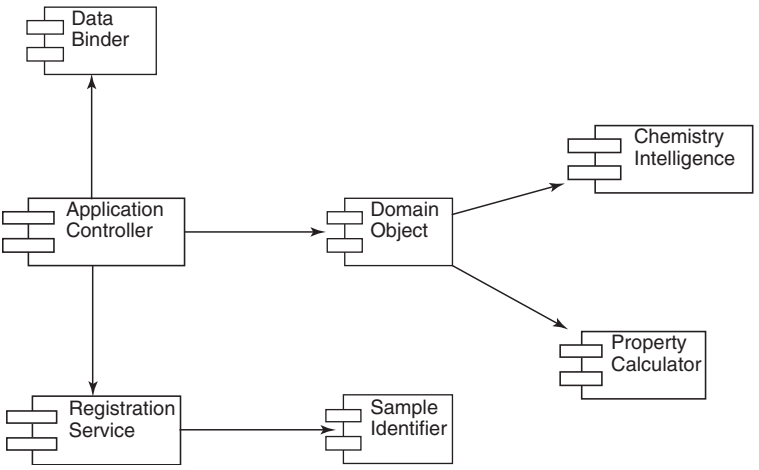


Figure 12.2 The component diagram of the business layer.

in Chapters 2 and 3, one of which is that dependencies between the software modules are at the implementation level rather than at the abstraction level. Software architecture is primarily concerned about dividing the system into subsystems and defining the interfaces and the interaction protocols between them. This technique is called *Design by Interface*.

Interface design and implementation design focus on two different aspects of software development. The interface of a module should be stable, simple, and intuitive for its clients to use. The latter two qualities are very much similar to graphic user interface (GUI) design, except that client here means other software modules rather than human users. On the other hand, the implementation of the module has to deal with issues such as data structure, algorithm, memory efficiencies, performance, thread safety, and network. These implementation complexities should be hidden from clients because the clients do not care about how the component does the work internally. We often talk about user friendliness when designing GUI. The same concepts apply to application programming interface (API) design. Often the interface is used by your peer developers in the same team or sometimes developers of other teams. Many organizations promote code sharing for productivity reasons. Nevertheless, three obstacles make code sharing difficult:

- Lack of useful documentation.
- Complexity of the API.
- Changes of the reusable component may have big impacts for clients.

Design by Interface helps to remove the second obstacle by hiding the complexity of the component with simple and intuitive interfaces. *Design by Interface* helps to remove the third obstacle by keeping the dependencies at the abstraction level.

A good API must have the following three characteristics:

- Stability: It should not change very often.
- Simplicity: Its parameter set should be as small and as cohesive as possible, and it should have a simple return type that its clients expect. It should also only throw exceptions that the clients care about.
- Clarity: Its name should clearly describe what it does or what services it provides.

When designing an API, put yourself into the position of a potential user of the API and think about how you would want the API to look and whether you would be willing to use it if it is designed the way it is. If the answer is "no," try to change it until you are satisfied as a potential user.

For the CRS system, we have identified seven core components. Our first task is to design their interfaces and the relationships between the interfaces. Figure 12.3 is the interface diagram of the CRS business layer.

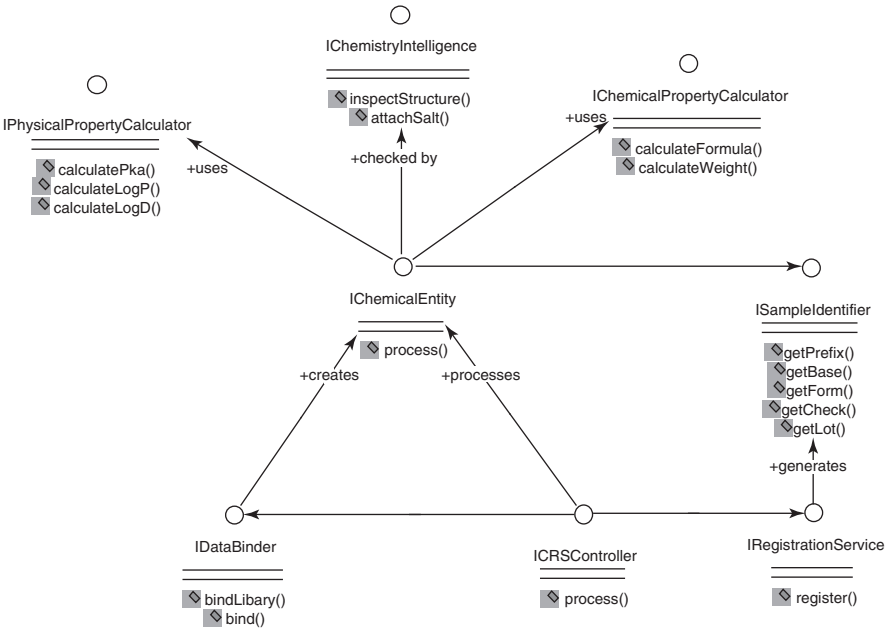


Figure 12.3 The interface diagram of the CRS business layer.

ICRSController serves as the Application Controller (Fowler, 2003a) that controls the application workflow. It uses IDataBinder to create domain objects from an SD file, an XML file, or any format of input. IChemicalEntity is the core domain object that can be either a chemical library or a chemical sample. IChemicalEntity uses IChemistryIntelligence for structure convention checking and attaching salts to the structure. IChemicalEntity also uses IChemicalPropertyCalculator and IPhysicalPropertyCalculator to calculate chemical and physical properties. Finally, when the IChemicalEntity is ready to be registered, ICRSController passes it to IRegistrationService, which stores the IChemicalEntity into the database.

These interfaces can be coded as either Java interfaces or abstract classes. Each of these interfaces and their implementations are discussed in more detail in this chapter.

12.2 DOMAIN OBJECTS

In Chapter 10, we presented a conceptual domain object model of CRS. The objects in the conceptual object model represent real-world concepts of the problem domain with which we are dealing. We said that these conceptual objects are candidates of real software objects. We also said that the design model will be more complex than the conceptual model because software

design principles and patterns introduce layers of abstractions and indirections. Let's see how the conceptual object model in Chapter 10 evolves to a design object model.

12.2.1 The Chemical Entity Object Design Model

The use case specification presented in Chapter 9 describes steps in a compound registering process:

- Enter structure
- Fragment structure
- Specify parent(s), salt(s), and coefficients
- QC structure
- Attach salt(s)
- Enter ancillary data
- Register

The goal of object design is to support the above workflow. As the Compound object is in the center of this process, it makes sense to further analyze and design the Compound object first.

The above workflow can very well be represented by a finite state machine—the compound object goes through a series of state transitions in the compound registration process. A finite state machine can be modeled using the UML State Diagram. Figure 12.4 is the UML State Diagram of a compound object.

When a compound object is created, its structure may have multiple fragments. The system should first decompose the structure into individual fragments and let the chemist specify their roles—parent, salt, and their coefficients. This is done at the Created State, which transitions to the Fragmented State when the fragmentation finishes. Next, the system must make sure the structure is fully compliant with structure conventions. This is the structure QC process. There are three possibilities as a result of this process:

1. The structure is fully compliant with structure conventions and the compound transitions to the Valid State.
2. The structure is not compliant with chemical conventions and the compound transitions to the Invalid State.
3. The structure is not compliant with chemical conventions, but the chemistry intelligence logic can automatically correct them and the compound transitions to the Corrected State.

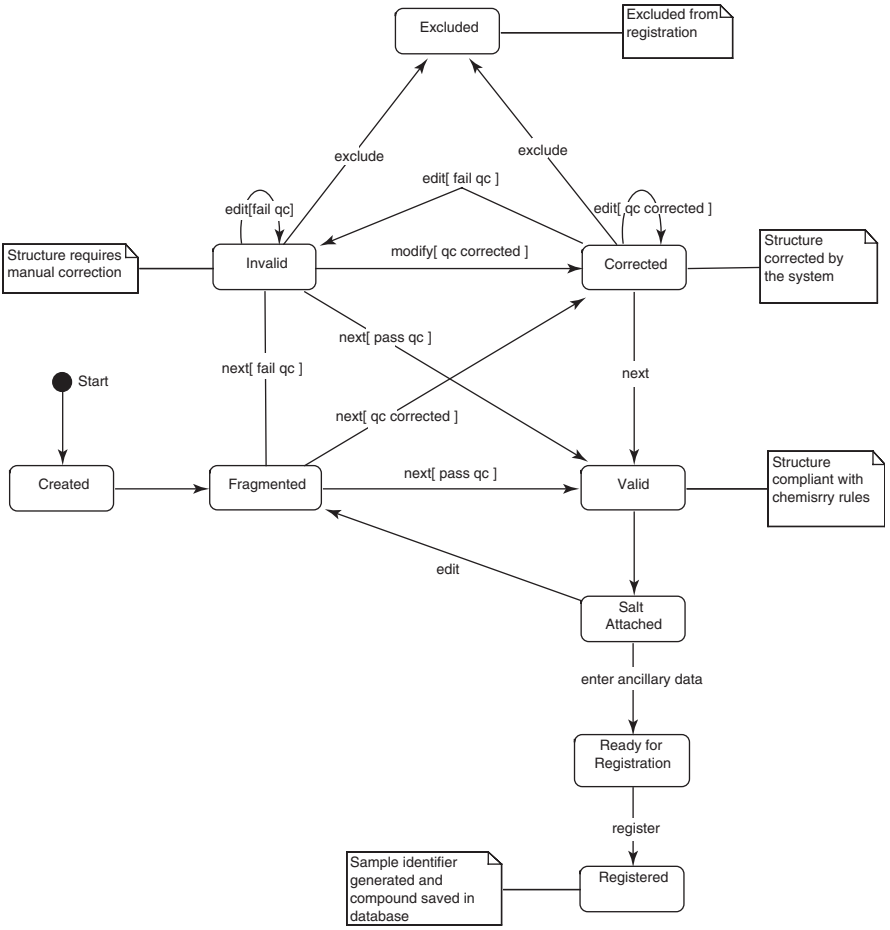


Figure 12.4 The state transition diagram of the Compound object.

If the structure is invalid, it requires manual correction that again leads to three possible state transitions:

1. The corrected structure becomes fully compliant with structure conventions and the compound transitions to the Valid State.
2. The corrected structure is not compliant with chemical conventions and the compound stays in the Invalid State.
3. The corrected structure is not compliant with chemical conventions, but the chemistry intelligence logic can automatically correct them and the compound transitions to the Corrected State.

If the structure is automatically corrected by the chemistry intelligence logic, the compound moves into the Corrected State. At this point, the

system should display the original and corrected structures to allow the chemist to review the changes. The chemist can further edit the corrected structure or accept the change. Depending on what the chemist does to the structure, the compound at Corrected State can have three possible state transitions:

1. The structure is edited by the chemist and becomes invalid and the compound transitions to the Invalid State.
2. The structure is edited by the chemist and is not compliant with chemical conventions, but the chemistry intelligence logic can automatically correct them and the compound remains in the Corrected State.
3. The structure is accepted or edited by the chemist and becomes valid and the compound transitions to the Valid State.

When the structure becomes valid, the system attaches salt to the parent structure and the compound transitions to Salt Attached State. At this point, if the chemist edits the structure, the system has to decompose the structure into fragments again and the whole structure process will start over. Otherwise, the system makes sure all required ancillary data are entered and the compound enters the Ready for Registration State. The final state of the compound is Registered State—compound stored in the database.

Another state the compound can transition to is the Excluded State. Compounds in this state are excluded from registration by the chemist.

The state transition analysis leads to an important design decision—the GoF State Pattern (Gamma et al., 1995).

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Figure 12.5 is the class diagram of compound states.

CompoundState is an abstract class that defines an abstract method process(). CreatedState, FragmentedState, ValidState, CorrectedState, InvalidState, and ReadyToBeRegisteredState are all concrete subclasses of CompoundState and must implement the process() method. They define the behaviors of a compound depending on its state.

Start with the CompoundState class, which is an abstract class. It has an attribute—compound. This compound is the one that the CompoundState describes. It is set in the constructor—when a CompoundState object is created (CompoundMemento object is described later in this chapter). CompoundState also defines an abstract method process(). Its source code is as follows:

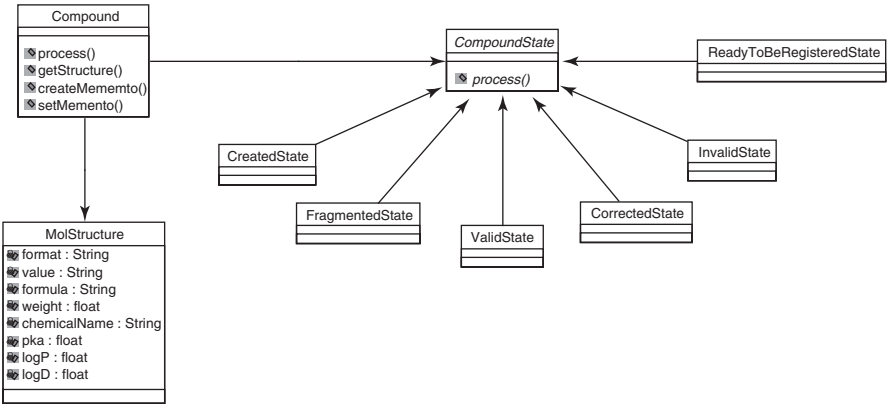


Figure 12.5 The State Pattern of the Compound object.

```
package com.abcpharma.crs.entity;

public abstract class CompoundState {
   protected Compound compound = null;
   protected CompoundMemento memento = null;

   public CompoundState(Compound compound){
      this.compound = compound;
   }

   public abstract void process();
}
```

CreatedState is a concrete class that extends CompoundState. Its constructor takes a Compound object as input and calls the constructor of its superclass—Compound State. Its process() method uses a chemistry intelligence object—Molstructure Inspector's fragment() method to decompose the structure of the compound object into structure fragments (Molstructure Inspector will be discussed in much more detail in Chapter 14). It then adds salts and parent fragments to the Compound object and sets its state to FragmentedState according to the state diagram in Figure 12.4. The roles of the fragments are assigned based on some basic rules, for example, by comparing each fragment with a salt dictionary. If a match is found, it is a salt; otherwise it is a parent. This is, of course, pending for the chemist to verify.

```
package com.abcpharma.crs.entity;

import com.abcpharma.crs.chemintell.*;
import com.abcpharma.crs.Molstructure.StructureFragment;
```

```
public class CreatedState extends CompoundState{

   public CreatedState(Compound compound){
      super(compound);
   }

   public void process() {
      MolstructureInspector inspector = MDLCheshire Molstructure Inspector
      Impl. getInstance();
      StructureFragment[] fragments = null;
      try{
         fragments = inspector.fragment(compound.getMolstructure());
         if(fragments != null){
         for (int i = 0; i < fragments.length; i++){
            if(fragments[i]. getRole().equals (Structure Fragment. PARENT_ROLE)){
               compound.addParent(fragments[i]);
            }
            else{
               compound.addSalt(fragments[i]);
            }
      }
   }
   compound.setState(compound.fragmentedState);
   }catch(ChemistryRulesException e){
      e.printStackTrace();
      }
   }
}
```

Now CRS should display the parent and salt fragments to allow the chemist to confirm or alter the fragment roles and specify their coefficients.

The current state is the FragmentedState. After the chemist has confirmed or altered the fragment roles and coefficients, an event will trigger the application controller to invoke FragmentedState's process() method. The process() method first uses a chemistry intelligence object—MolstructureInspector—to combine all parent fragments of the compound object to create a combined parent structure. It then uses the MolstructureInspector object to inspect the structure using structure convention rules. According to Figure 12.4, the structure QC logic determines to what state the compound transitions. If the structure is compliant with structure conventions, it transitions to the Valid State. If the structure has problems but is corrected by the structure QC logic, it transitions to the Corrected State. If the structure has problems and requires manual correction, it transitions to the Invalid State. The source code of the Fragmented State class is as follows:

```
package com.abcpharma.crs.entity;

import com.abcpharma.crs.chemintell.*;

public class FragmentedState extends CompoundState {

   public FragmentedState(Compound compound){
      super(compound);
   }

   public void process(){
      MolstructureInspectorinspector = MDLCheshireMolstructureInspector
      Impl.getInstance();
      String[] message = null;
      try{

         compound.setMolstructure(inspector.combineParentFragments(
            compound.getParentFragments()));
         message = inspector.executeAllRules(compound.get Molstructure());
         if(message != null){
            compound.setState(compound.correctedState);
            compound.setStateDescription(message);
         }
         else
            compound.setState(compound.validState);
   }catch(ChemistryRulesException e){
         compound.setState(compound.invalidState);
         message = new String[1];
         message[0] = e.getMessage();
         compound.setStateDescription(message);
      }
   }
}
```

InvalidState is another concrete class that extends CompoundState. Just as the FragmentedState, its constructor takes a Compound object as input and calls the constructor of its superclass—CompoundState. Its process() method uses a chemistry intelligence object—MolstructureInspector—to inspect the structure of the compound object. Depending on the return of the MolstructureInspector object's executeAllRules method, the InvalidState object alters the state of the compound object according to the state diagram in Figure 12.4.

```
package com.abcpharma.crs.entity;

import com.abcpharma.crs.chemintell.*;

public class InvalidState extends CompoundState {
```

```
    public InvalidState(Compound compound){
        super(compound);

    }
    public void process() {
        MolstructureInspector inspector = MDLCheshireMolstructureInspector
        Impl.getInstance();
        String[] message = null;
        try{
            message = inspector.executeAllRules(compound.get
            Molstructure());
            if(message != null){
                compound.setState(compound.correctedState);
                compound.setStateDescription(message);
            }
            else
                compound.setState(compound.validState);
        }catch(ChemistryRulesException e){
            message = new String[1];
            message[0] = e.getMessage();
            compound.setStateDescription(message);
        }
    }
}
```

Likewise, the following class implements CorrectedState—another concrete CompoundState:

```
package com.abcpharma.crs.entity;

import com.abcpharma.crs.chemintell.*;

public class CorrectedState extends CompoundState {

    public CorrectedState(Compound compound) {
        super(compound);
    }

    public void process() {
        MolstructureInspector inspector = MDLCheshireMolstructureInspector
        Impl.getInstance();
        String[] message = null;
        try{
            message = inspector.executeAllRules(compound.get
            Molstructure());
            if(message != null){
                compound.setStateDescription(message);
            }
            else
```

```
            compound.setState(compound.validState);
        }catch(ChemistryRulesException e){
            compound.setState(compound.invalidState);
            message = new String[1];
            message[0] = e.getMessage();
            compound.setStateDescription(message);
        }
    }
}
```

When the compound becomes valid, salts can be attached to its parent structure, which is accomplished by the ValidState object:

```
package com.abcpharma.crs.entity;

import com.abcpharma.crs.chemintell.*;

public class ValidState extends CompoundState {

    public ValidState(Compound compound) {
        super(compound);
    }

    public void process() {
        SaltHandler handler = MDLCheshireSaltHandlerImpl.getInstance();
        String[] message = new String[1];
        message[0] = handler.executeAddSalt(compound.getMolstructure(),
            compound.getSaltFragments());
        compound.setState(compound.saltAttachedState);
        compound.setStateDescription(message);
    }
}
```

If the compound has salt(s), the ValidState object attaches the salts to the parent structure using a SaltHandler object (SaltHandler will be discussed in much more detail in Chapter 14). The compound transitions to the Salt Attached State.

The last state of a compound in the compound registration context is Ready To Be Registered State. A compound is ready for registration when its structure passes chemistry convention rules and salts have been attached, and all required ancillary data, such as project and notebook information, have been entered.

Each organization may have different business rules with regard to compound state transition, but the above discussion captures the essence of how such a system can be designed.

Now look at the Compound class. The Compound class has all its states as its instance variables. These variables are instantiated when the compound object is created (see its constructor), not the first time when they are needed.

This process is called *eager instantiation*. Another approach is not to create these state objects until the first time when they are needed, which is called *lazy instantiation*. *Lazy instantiation* is more resource efficient because the objects are not created before they are needed and are never created if they are never needed. In our situation, if we take the lazy approach, the state objects will be responsible for creating other state objects, which is against the Expert Principle (Larman, 2005).

Expert Principle: Assign a responsibility to the class that has the information needed to fulfill it.

If we take the *lazy instantiation* approach, the process() in the Fragmented State class will appear as follows:

```
public void process(){
   MolstructureInspector inspector = MDLCheshireMolstructureInspector
   Impl.getInstance();
   String[] message = null;
   try{

   compound.setMolstructure(inspector.combineParentFragments(
      compound.getParentFragments()));
   message = inspector.executeAllRules(compound.getMolstructure());
   if(message != null){
      if(compound.qcedState == null)
         compound.qcedState = newQCedState(compound);
      compound.setState(compound.qcedState);
      compound.setStateDescription(message);
   }
   else{
      if(compound.validState == null)
         compound.validState = newValidState(compound);
compound.setState(compound.validState);
   }
}catch(ChemistryRulesException e){
   if(compound.invalidState == null)
      compound.invalidState = new InvalidState(compound);
      compound.setState(compound.invalidState);
      message = new String[1];
      message[0] = e.getMessage();
      compound.setStateDescription(message);
   }
}
```

The highlighted changes will be in all state classes, which is awkward.

Every design has trade-offs. We use eager instantiation to trade resource efficiency for clarity. Which approach to take should be determined on a case-by-case basis.

The other instance variables in the Compound class include a Molstructure object, a list of salt structures, a list of parent structures, a String array that contains information about what has been changed or what the problems are in the structure as a result of the chemistry rules checking, and a CompoundState object representing the current state of the Compound object. The process() method in the Compound object simply delegates the call to its current state object, which knows how to process the Compound object depending on its current state as described above.

```
package com.abcpharma.crs.entity;

import java.util.*;

import com.abcpharma.crs.Molstructure.*;

public class Compound {

   private Molstructure structure = null;
   private List salts = new ArrayList();
   private List parents = new ArrayList();

   private String[] stateDescription = null;

   CompoundState createdState = null;
   CompoundState fragmentedState = null;
   CompoundState correctedState = null;
   CompoundState validState = null;
   CompoundState invalidState = null;
   CompoundState saltAttachedState = null;
   CompoundState readyForRegistrationState = null;
   CompoundState state = createdState;

   public Compound(){
      createdState = new CreatedState(this);
      fragmentedState = new FragmentedState(this);
      correctedState = new CorrectedState(this);
      validState = new ValidState(this);
      invalidState = new InvalidState(this);
      saltAttachedState = new SaltAttachedState(this);
      readyForRegistrationState = new ReadyForRegistrationState(this);
   }

   public Molstructure getMolstructure(){
      return structure;
   }

   public void setMolstructure(Molstructure structure){
      this.structure = structure;
   }
```

```
public List getSaltFragments(){
   return Collections.unmodifiableList(salts);
}

public Iterator getSaltIterator(){
   return salts.iterator();
}

public void addSalt(StructureFragment fragment) throws IllegalArgumentException{
   if(!fragment.getRole().equals(StructureFragment.SALT_ROLE))
      throw new IllegalArgumentException("Fragment is not a salt.");
   salts.add(fragment);
}

public void removeSalt(StructureFragment fragment) throws IllegalArgument
Exception{
   if(!fragment.getRole().equals(StructureFragment.SALT_ROLE))
      throw new IllegalArgumentException("Fragment is not a salt.");
   salts.remove(fragment);
}

public List getParentFragments(){
   return Collections.unmodifiableList(parents);
}

public Iterator getParentIterator(){
   return parents.iterator();
}

public void addParent(StructureFragment fragment) throws IllegalArgument Exception{
   if(!fragment.getRole().equals(StructureFragment.PARENT_ROLE))
      throw new IllegalArgumentException("Fragment is not a parent.");
   parents.add(fragment);
}

public void removeParent(StructureFragment fragment) throws IllegalArgument Exception{
   if(!fragment.getRole().equals(StructureFragment.PARENT_ROLE))
      throw new IllegalArgumentException("Fragment is not a parent.");
   parents.remove(fragment);
}

void setState(CompoundState state){
   this.state = state;
}

void setStateDescription(String[] stateDescription){
   this.stateDescription = stateDescription;
}
```

```
    public String[] getStateDescription(){
        return stateDescription;
    }

    public void process(){
        state.process();
    }

}
```

At this point, the original single object, Compound in the conceptual model in Chapter 10, has evolved to a design model with eight classes using the GoF State Pattern. We used the conceptual model to inspire design ideas. However, object design is different from conceptual modeling in that it introduces additional layers of indirections to reduce coupling and increase cohesion.

The above design using The State Pattern addresses a simple one-way traffic scenario of the flow—the process can only go forward, not backward. In reality, the chemist may want to undo and roll back a step either because of human mistakes or the system did something that is against the chemist's intention. "Undo" is a common functionality in many productivity software systems, such as Microsoft Word. Fortunately, there is another GoF design pattern called Memento, which provides an elegant solution.

To perform undo, the system has to remember the states of the objects along the way in the process. Memento is the object that stores previous states on behalf of the subject object and allows the subject object to recover its previous states when necessary.

The Memento Pattern: Without violating encapsulation, capture and externalize an object's state so that the object can be restored to this state later.

Suppose the compound state transition only involves structure changes and undo only needs to recover the structure and the compound state. Figure 12.6 is the class diagram of the Memento Pattern for the Compound object.

The source code of CompoundMemento class is as follows:

```
package com.abcpharma.crs;

import com.abcpharma.crs.Molstructure.*;

class CompoundMemento {
    private Molstructure structure = null;
    private CompoundState state = null;

    void setState(Molstructure structure, CompoundState state ){
```

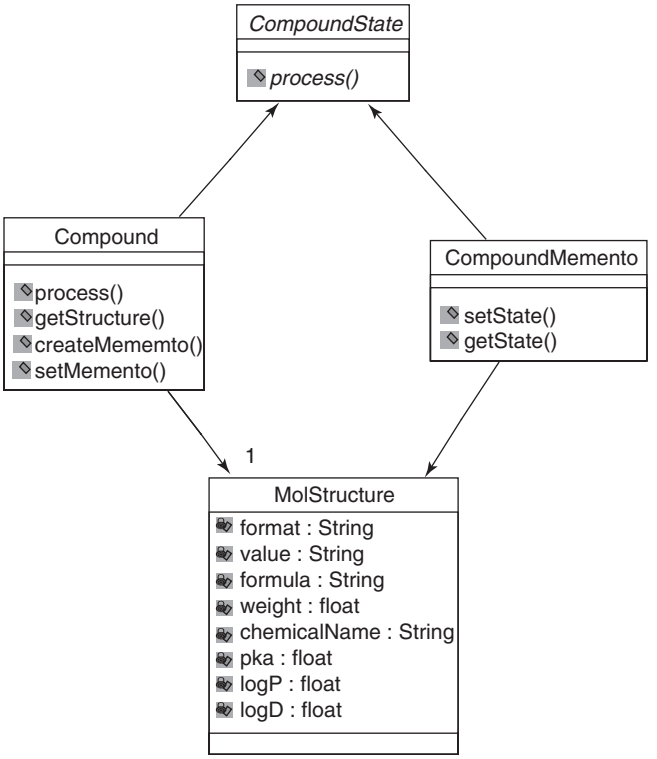


Figure 12.6 The class diagram of the Memento Pattern for the Compound object.

```
        this.structure = structure;
        this.state = state;
    }

    Molstructure getStructure(){
        return structure;
    }

    CompoundState getState(){
        return state;
  }
}
```

The modified Compound class has three additional methods: createMemento( ), setMemento( ), and undo( ). They are highlighted in the following source code:

```
package com.abcpharma.crs.entity;

import java.util.*;

import com.abcpharma.crs.Molstructure.*;
```

```
public class Compound {
   private Molstructure structure = null;
   private List salts = new ArrayList();
   private List parents = new ArrayList();

   private String[] stateDescription = null;

   CompoundState createdState = null;
   CompoundState fragmentedState = null;
   CompoundState correctedState = null;
   CompoundState validState = null;
   CompoundState invalidState = null;
   CompoundState saltAttachedState = null;
   CompoundState readyForRegistrationState = null;
   CompoundState state = createdState;

   public Compound(){
      createdState = new CreatedState(this);
      fragmentedState = new FragmentedState(this);
      correctedState = new CorrectedState(this);
      validState = new ValidState(this);
      invalidState = new InvalidState(this);
      saltAttachedState = new SaltAttachedState(this);
      readyForRegistrationState = new ReadyForRegistrationState(this);
   }

   public Molstructure getMolstructure(){
      return structure;
   }

   public void setMolstructure(Molstructure structure){
      this.structure = structure;
   }

   public List getSaltFragments(){
      return Collections.unmodifiableList(salts);
   }

   public Iterator getSaltIterator(){
      return salts.iterator();
   }

   public void addSalt(StructureFragment fragment) throws IllegalArgument
   Exception{
      if(!fragment.getRole().equals(StructureFragment.SALT_ROLE))
         throw new IllegalArgumentException("Fragment is not a salt.");
      salts.add(fragment);
}
```

```
public void removeSalt(StructureFragment fragment) throws IllegalArgument
Exception{
   if(!fragment.getRole().equals(StructureFragment.SALT_ROLE))
      throw new IllegalArgumentException("Fragment is not a salt.");
   salts.remove(fragment);
}

public List getParentFragments(){
   return Collections.unmodifiableList(parents);
}

public Iterator getParentIterator(){
   return parents.iterator();
}

public void addParent(StructureFragment fragment) throws IllegalArgument
Exception{
   if(!fragment.getRole().equals(StructureFragment.PARENT_ROLE))
      throw new IllegalArgumentException("Fragment is not a parent.");
   parents.add(fragment);
}

public void removeParent(StructureFragment fragment) throws IllegalArgument
Exception{
   if(!fragment.getRole().equals(StructureFragment.PARENT_ROLE))
      throw new IllegalArgumentException("Fragment is not a parent.");
   parents.remove(fragment);
}

void setState(CompoundState state){
   this.state = state;
}

void setStateDescription(String[] stateDescription){
   this.stateDescription = stateDescription;
}

public String[] getStateDescription(){
   return stateDescription;
}

public CompoundMemento createMemento(){
   CompoundMemento memento = new CompoundMemento();
   memento.setState(structure, state);
   return memento;
}

void setMemento(CompoundMemento memento){
   structure = memento.getStructure();
   state = memento.getState();
}
```

```
    public void undo(CompoundMemento memento){
        setMemento(memento);
    }

    public void process(){
        state.process();
    }
}
```

The createMemento() method creates and returns a CompoundMemento object that is a snapshot of the current state of the Compound object. The setMemento() method restores the state of the Compound object to the state that was saved in the CompoundMemento object that is passed in. The undo() method is called by the client of Compound object to restore the state.

Sometimes it is necessary to perform undo multiple steps. In this case, create a Caretaker class that holds a stack of Memento objects. If redo is desired, then have two stacks in the Caretaker, one for undo and one for redo, and the Memento object that is popped from the undo stack gets pushed into the redo stack and vice versa. The code example is as follows:

```
package com.abcpharma.crs.entity;

import java.util.*;

public class ComoundMementoCaretaker {
    private Stack undoStack = new Stack();
    private Stack redoStack = new Stack();
    private Compound compound = null;

    public ComoundMementoCaretaker(Compound compound){
        this.compound = compound;
    }

    public void saveState(){
        undoStack.push(compound.createMemento());
    }

    public void undoState(){
        if(undoStack.empty())
            return;
        CompoundMemento memento = (CompoundMemento) undoStack. pop();
        compound.undo(memento);
        redoStack.push(memento);
    }

    public void redoState(){
```

```
        if(redoStack.empty())
            return;
        CompoundMemento memento = (CompoundMemento) redoStack.pop();
        compound.redo(memento);
        undoStack.push(memento);
    }
}
```

Figure 12.7 is a sequence diagram that shows undo and redo using Memento Pattern.

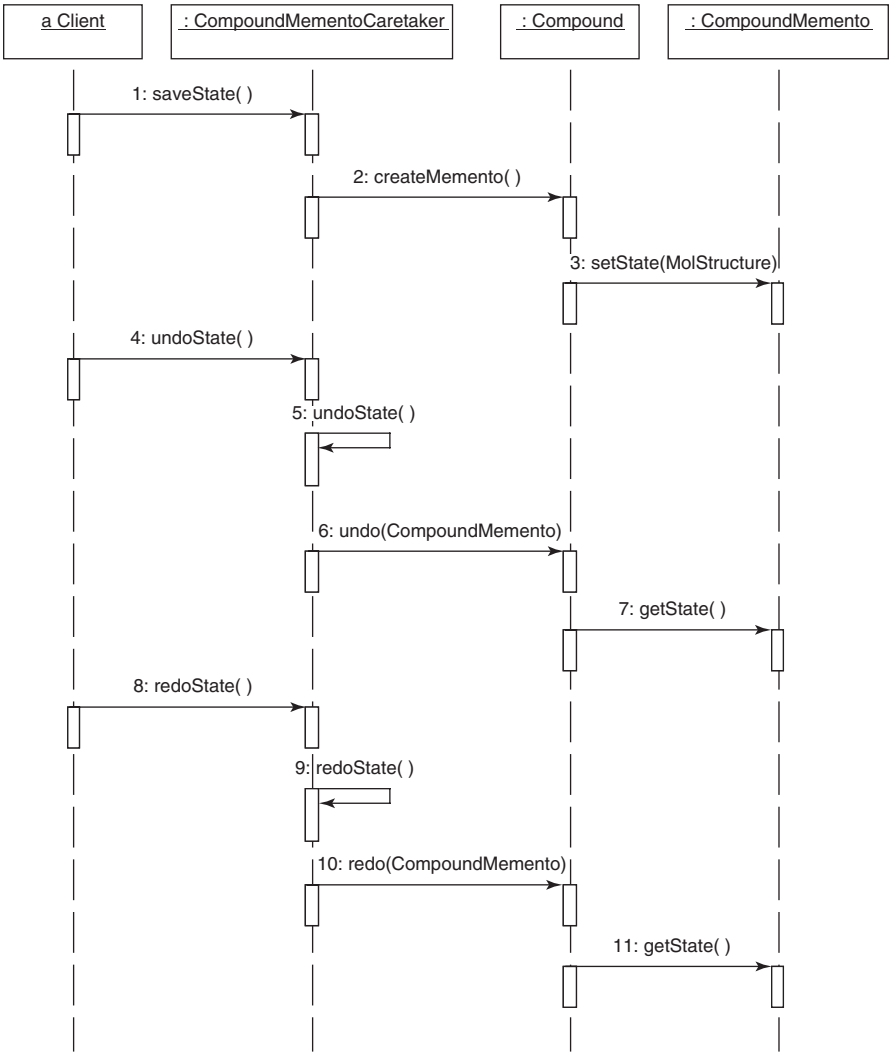


Figure 12.7 The sequence diagram of Memento Pattern.

12.2.2 Molecular Structure Object Model

Another key concept in CRS is molecular structure. A molecular structure has the following basic attributes:

- Format: The format in which the structure is represented—Molfile, Smiles, CXL, and so on.
- Value: The actual molecular structure represented in the specified format.
- Formula: Molecular formula—an attribute that can be derived from the structure.
- Weight: Molecular weight—an attribute that can be derived from the structure.
- Chemical Name: An attribute that can be derived from the structure.
- pKa: The negative logarithm of the acid dissociation constant—an attribute that can be derived from the structure.
- logP: Octanol–water partition coefficient—an attribute that can be derived from the structure.
- logD: Octanol–water distribution coefficient—an attribute that can be derived from the structure.

The source code of the Molstructure class is as follows:

```
package com.abcpharma.crs.Molstructure;

public class Molstructure {
   public static final String MOLFILE = “molfile”;
   public static final String SMILES = “smiles”;
   public static final String CDX = “cdx”;
   public static final String CDXML = “cdxml”;

   private String format;
   private String value;
   private String formula;
   private float weight;
   private String chemicalName;
   private float pka;
   private float logP;
   private float logD;

   public Molstructure(String format, String value) throws IllegalArgumentException{
      if(!format.equals(MOLFILE)
         && !format.equals(SMILES)
         && !format.equals(CDX)
         && !format.equals(CDXML)
```

```
            )
        throw new IllegalArgumentException("Structure format " + format + "
        not supported.");
        this.format = format;
        this.value = value;
    }

    public String getFormat(){
        return format;
    }

    public void setFormat(String format){
        this.format = format;
    }

    public String getValue(){
        return value;
    }

    public void setValue(String value){
        this.value = value;
    }

    public String getFormula(){
        return formula;
    }

    public float getWeight(){
        return weight;
    }

    public String getChemicalName(){
        return chemicalName;
    }

    public float getLogP(){
        return logP;
    }

    public float getLogD(){
        return logD;
    }

    public float getPka(){
        return pka;
    }

    private String calculateFormula(){
        if(format.equals(MOLFILE))
            return "formula calculated from molfile";
```

```
        if(format.equals(SMILES))
            return "formula calculated from smiles";
        return null;
    }
}
```

Molstructure is similar to the Quantity object described in Martin Fowler's *Analysis Patterns* book (Fowler, 1996) in that it has a format (unit) and a value.

Some compound registration systems require that not only the combined molecular structure gets registered, but its structure components or fragments get registered as well. By "combined" structure, I mean isomer mixture or a structure with salt(s). These fragments can be either parents or salts. The advantage of storing structure components is that some research activity may only care about parent structures. Here we introduce another concept—Structure Fragment. A Structure Fragment has all of the attributes that a molecular structure has plus two more attributes:

- Role: The role of the fragment in the combined molecular structure. It can take two possible values: parent or salt.
- Coefficient: The relative count of the fragment in the combined molecular structure.

We can make StructureFragment a subclass of Molstructure as in Figure 12.8.

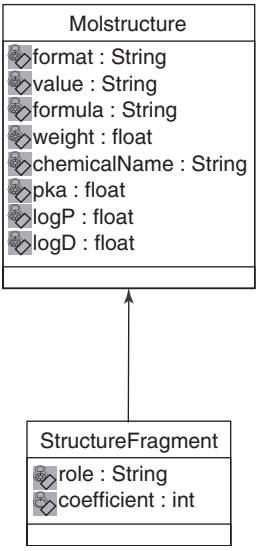


Figure 12.8 The class diagram of Molstructure and StructureFragment.

The source code of the StructureFragment class is as follows:

```
package com.abcpharma.crs.molstructure;

public class StructureFragment extends Molstructure {
   public static final String PARENT_ROLE = "parent";
   public static final String SALT_ROLE = "salt";

   private String role;
   private int coefficient;

   public StructureFragment(String format, String value, String role, int coefficient)throws
      IllegalArgumentException{
      super(format, value);
      setRole(role);
      setCoefficient(coefficient);
   }

   public String getRole(){
      return role;
   }

      public void setRole(String role) throws IllegalArgumentException{
         if(!role.equals(PARENT_ROLE) && !role.equals(SALT_ROLE)){
            throw new IllegalArgumentException("Illegal fragment role: "+ role);
      }
      this.role = role;
   }

   public int getCoefficient(){
      return coefficient;
   }

   public void setCoefficient(int coefficient){
      this.coefficient = coefficient;
   }
}
```

Some systems support various structure formats and need to convert the structures from one format to another from time to time. Although this may not be required by a registration system, it could be very useful in other types of chemical information systems such as molecular modeling. The good news is that some tools from the major chemical informatics software vendors can do the conversions. In this situation, commercial solutions make sense because you do not want to reinvent the wheel.

However, when you start developing your system, you may not know what tool you will end up using or you may pick one tool today and switch to

a different tool in the future for one reason or another. These tools may be developed in different technologies and may have different APIs. Still you want to build in the flexibility to insulate vendor specifics so that the system under development is not affected when the tools change. This result can be achieved by leveraging the power of abstraction—separation of interface from implementation. Designing the system to insulate itself from vendor specifics is an important subject of this book and will be touched on again in other chapters.

In Chapter 3, we talked about the Yin and Yang of an object. The Yang represents the external view of the object—behaviors that an outsider sees. It is encoded by the interface of the object. When designing an object's interface, think about what the clients of the object would expect: easy to understand and simple to use.

The interface of a Molfile–Smiles converter is as follows:

```
package com.abcpharma.crs.molstructure;

public interface MolfileSmilesConverter {
   public String molfileToSmiles(String molfile) throws MolstructureConversion
   Exception;
   public String smilesToMolfile(String smiles) throws MolstructureConversion
   Exception;
}
```

It defines two simple methods. The molfileToSmiles() method converts a Molfile string to a Smiles string. The smilesToMolfile() method does the opposite—converting a Smiles string to a Molfile string. They both throw MolstructureConversionException to flag conversion failures. The failure can be caused by exceptions from the vendor implementation or from the CRS code.

The vendor tools that you end up using may have completely different APIs. But you should not care when you design the interfaces of your system. Remember, the vendor API may not be the best design, or at least it may not be ideal for your system. All you need to care about is what makes the most sense to the clients that use the structure conversion API in your system. To do the conversion between Molfile and Smiles, the above API is the simplest.

Now assume you want to use the Vendor A solution. But the Vendor A API is different from MolfileSmilesConverter—the names of the methods may be different, parameter sets might be different, and the exceptions might be different too. How do you stick to your API on which the rest of your system is already dependent while leveraging a third-party solution? The *GoF Design Patterns* book offers an elegant solution: the Adapter (Gamma et al., 1995).

The Adapter Pattern converts the interface of a class into another interface that the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

A code example of the Adapter Pattern that leverages the Vendor A solution for Mofile–Smiles conversion and still adapts to the MolfileSmiles Converter interface is as follows:

```
package com.abcpharma.crs.molstructure;

public class MolfileSmilesVendorAAdapter implements MolfileSmilesConverter {
   private static final MolfileSmilesVendorAAdapter instance = new MolfileSmiles
   VendorAAdapter();

   private MolfileSmilesVendorAAdapter(){
   }

   public static MolfileSmilesVendorAAdapter getInstance(){
      return instance;
   }

   public String molfileToSmiles(String molfile) throws MolstructureConversion
   Exception{
      try{
         Call Vendor A API to convert molfile to smiles
      }catch(VendorAException ex){
         throw new MolstructureConversionException(ex.getMessage());
         }

         return "Vendor A smiles";
   }

   public String smilesToMolfile(String smiles) throws MolstructureConversion
   Exception{
      try{
         Call Vendor A API to convert smiles to molfile
      }catch(VendorAException ex){
         throw new MolstructureConversionException(ex.getMessage());
      }

      return "Vendor A molfile";
   }
}
```

The molfileToSmiles() and smilesToMofile() methods delegate the conversion calls to Vendor A APIs to do the work and therefore insulate the client of MolfileSmilesConverter from vendor implementations. The client of MolfileSmilesConverter API has no knowledge of who actually does the

work and is completely decoupled from vendor implementations. If someday in the future your company wants to switch to the Vendor B solution for some reason, all you have to do is write another Adapter that leverages the Vendor B solution and yet hides Vendor B APIs from the clients:

```
package com.abcpharma.crs.molstructure;

public class MolfileSmilesVendorBAdapter {
   private static final MolfileSmilesVendorBAdapter instance = new MolfileSmiles
   VendorBAdapter();

   private MolfileSmilesVendorBAdapter(){
   }

   public static MolfileSmilesVendorBAdapter getInstance(){
      return instance;
   }

   public String molfileToSmiles(String molfile) throws MolstructureConversion
   Exception{
      try{
         Call Vendor B API to convert molfile to smiles
   }catch(VendorBException ex){
         throw new MolstructureConversionException(ex.getMessage());
   }

      return "Vendor B smiles";
   }

   public String smilesToMolfile(String smiles) throws MolstructureConversion
   Exception{
      try{
         Call Vendor B API to convert smiles to molfile
      }catch(VendorBException ex){
         throw new MolstructureConversionException(ex.getMessage());
      }

      return "Vendor B molfile";
   }
}
```

To truly decouple client code from vendor solutions, the vendor-specific exception has to be handled by the adapters as well. If Vendor A API throws exceptions that are proprietary, there must be a try–catch block surrounding the Vender A API call that either handles it in the catch block or rethrows the MolstructureConversionException that contains information in the vendor exception for the client to handle. Otherwise the vendor proprietary exception

has to be exposed directly to the client, which increases the coupling and inhibits vendor insulation.

You may have noticed that both MolfileSmilesConverterAdapterA and MolfileSmilesConverterAdapterB have a private constructor and a static getInstance() method that returns an instance of the class. Yes, this is the GoF Singleton Pattern (Gamma et al., 1995).

The Singleton Pattern ensures that a class has only one instance, and it provides a global point of access to it.

The reason Singleton Pattern is used in these two classes is that only one instance of each is needed in the entire JVM, because there is no instance variable in these two classes—they are stateless. Limiting the number of objects that can be created is a way to increase runtime efficiency and achieve good performance because object creation and garbage collection are two of the most expensive operations in JVM. The Singleton Pattern is implemented by creating a private constructor—this makes sure no instance of the class can be created outside the class itself, a static final instance of the class itself, and a public static method that returns the instance. As the constructor is private, none of its clients can instantiate the class directly. Instead, they have to use the public static method getInstance() to retrieve the one and only instance of the class.

An example of a client class of the above Molfile–Smiles conversion API is as follows:

```
package com.abcpharma.crs.Molstructure;

public class AClientOfMolfileSmilesConverter {
   private MolfileSmilesConverter msConverter = null;

   public AClientOfMolfileSmilesConverter(MolfileSmilesConverter msConverter){
      this.msConverter = msConverter;
   }

   public void aMethod(String molfile){
      //do something

      try{
         String smiles = msConverter.molfileToSmiles(molfile);
      }catch(MolstructureConversionException ex){
         ex.printStackTrace();
      }

      //do something
   }
}
```

The constructor takes a MolfileSmilesConverter object as the argument and saves it into an instance variable. Its business method aMethod() calls the MolfileSmilesConverter object to convert a molfile to smiles. Because AClientOfMolfileSmilesConverter class uses the base type MolfileSmilesConverter, it is not coupled with any implementation of MolfileSmilesConverter interface. It is rather dependent on an abstraction—the MolfileSmilesConverter interface. Depending on abstraction instead of implementation is a good practice in object-oriented design that makes the system easy to maintain and change. This is called The Dependency Inversion Principle (DIP) (Martin, 2003). With DIP, when implementation of MolfileSmilesConverter changes, AClientOfMolfileSmilesConverter does not need to change at all.

The Dependency Inversion Principle:

1. **High-level modules should not depend on low-level modules. Both should depend on abstractions.**
2. **Abstractions should not depend on details. Details should depend on abstractions.**

Figure 12.9 illustrates the design of the MolfileSmilesConverter framework.

Figure 12.10 is the sequence diagram that illustrates how the objects work together in the above framework. You may wonder how the system determines

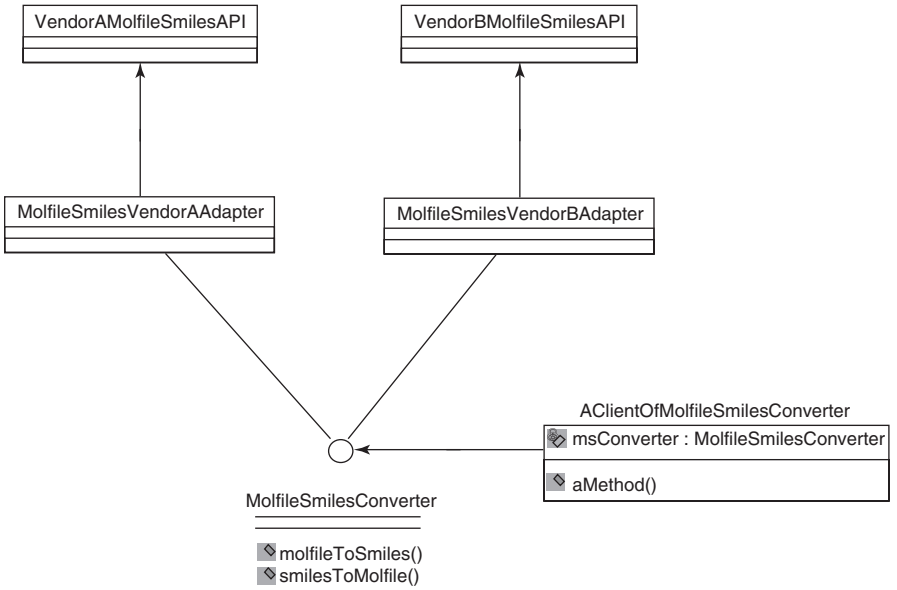


Figure 12.9 The design of the MolfileSmilesConverter framework using the Adapter Pattern.

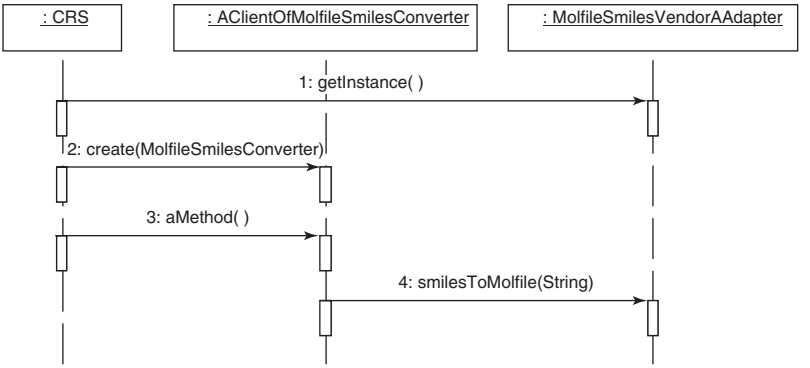


Figure 12.10 The sequence diagram that illustrates how objects work together in the Molfile–Smiles conversion framework.

which MolfileSmilesConverter implementation to use. The goal is to not change and recompile the code when switching from one implementation to another. There are several ways to achieve this goal. One way is to make this configurable in an application configuration file and use an application initialization object to read in the configurations from the file and set it as an application property. The configuration file can even include business rules: for example, if A is true, use MolfileSmilesVendorAAdapter; else if B is true, use Molfile SmilesVendorBAdapter. In Figure 12.10, the application CRS determines that the system should use Vendor A implementation. It gets the singleton instance from its static factory method—getInstance()—creates an instance of AClient OfMolfileSmilesConverter, and calls its aMethod(). Because the Molfile-SmilesConverter is set to MolfileSmilesVendorAAdapter in the constructor of AClientOfMolfileSmilesConverter, the aMethod() invokes the smilesToMolfile implementation of MolfileSmilesVendorAAdapter at runtime. AClient OfMolfileSmilesConverter has no knowledge of which implementation it uses at compile time. This process is called dynamic binding or late binding.

The above design reduces coupling in two ways. The Adapter Pattern decouples the system from vendor APIs. The Dependency Inversion Principle further decouples the client of MolfileSmilesConverter from its implementations—the adapters. Low coupling makes the system easy to change; adding adapters for new vendors does not need to change any existing code. Therefore, the system is Open for Extensions and Closed for Changes.

The same approach can be used for property calculations, because you may have multiple choices and each one has different APIs. It is possible that today you choose one vendor solution to do the work. Some time in the future you may switch to a different vendor for the purpose of cost savings, better algorithm, performance, or reliability. Or you may face a harder choice—the vendor you chose in the past is going out of business. The solution we described above for Molfile–Smiles conversion keeps your system closed for

these types of changes and yet makes your system adaptive to business needs. Figure 12.11 is the class diagram of a structure property calculator.

It must be pointed out that Figure 12.11 may not be the best design for property calculations. When we talked about High Cohesion Principle in Chapter 2, we said that a module should only have one reason to change. In other words, if a module has to change, all of its methods should change together. Otherwise the module should be broken down to smaller, more cohesive modules so that each module can evolve independently. In the StructurePropertyCalculator interface, there are two types of properties. One is chemical properties such as formula and weight. The other one is physical properties such as pKa, logP, and logD. It is very likely that you may use different vendor tools to calculate them. You may choose MDL tools to calculate chemical properties and ACDLabs tool to calculate physical properties. Using The Interface Segregation Principle (ISP) (Martin, 2003), we should separate them into two interfaces (Figures 12.12 and 12.13)

The Interface-Segregation Principle: Clients should not be forced to depend on methods that they do not use.

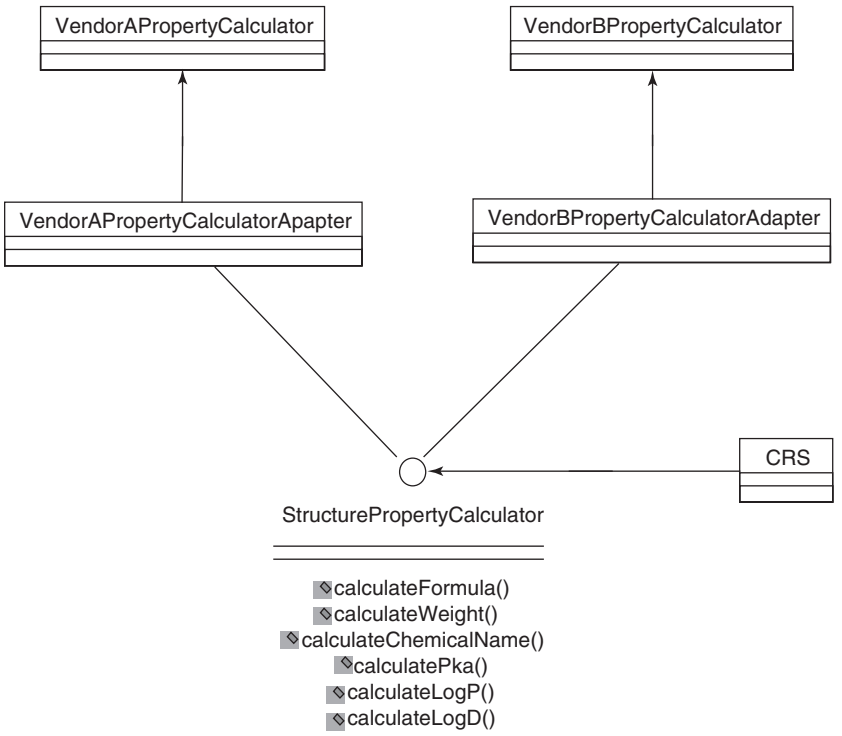


Figure 12.11 The class diagram of structure property calculator.

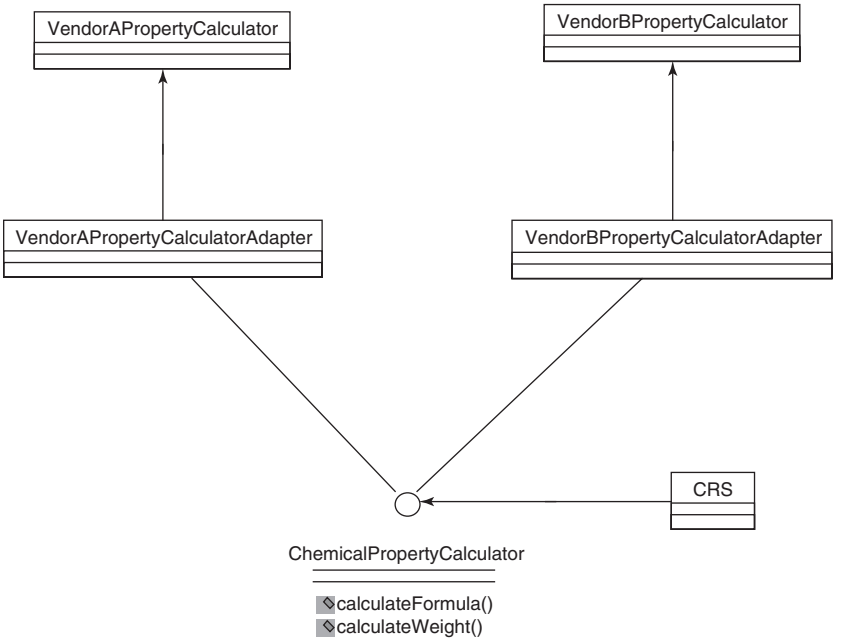


Figure 12.12 The class diagram of a ChemicalPropertyCalculator.

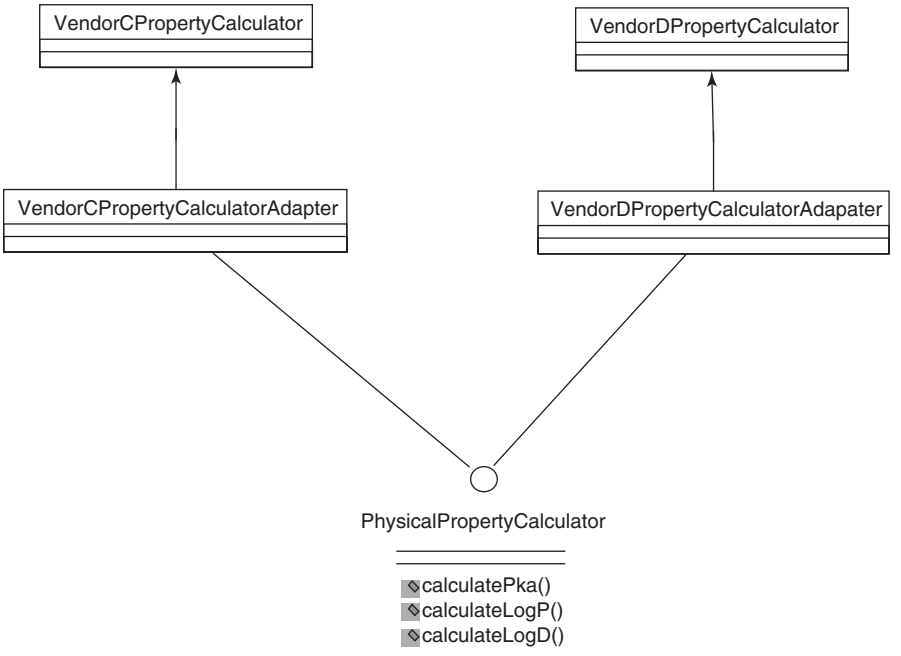


Figure 12.13 The class diagram of a PhysicalPropertyCalculator.

Figures 12.12 and 12.13 increase the cohesion of the interface by further limiting the responsibilities of each, and therefore, each one can evolve more "freely."

12.2.3 The Chemical Entity Object Design Model

To increase productivity, a registration system should be able to register a compound library as a single transaction. Usually a compound library consists of a group of compounds that are synthesized using parallel synthesis, combinatorial chemistry, or compounds that are acquired from commercial or academic sources. From the compound registration perspective, a library can also be a group of compounds that share some common attributes such as a research project they are synthesized for, the chemist who synthesized them, the creation date, and the notebook information.

A compound library contains one to many compound samples. In other words, library and compound sample form a "Has-A" relationship. At the same time, library and compound samples also share some common attributes such as project, chemist, and notebook. Therefore, they should have a common base class—we call it ChemicalEntity. In Figure 12.14, abstract class ChemicalEntity owns properties that are common to ChemicalLibrary and ChemicalSample, such as chemist, project, creation date, and notebook. Both ChemicalLibrary and ChemicalSample classes extend ChemicalEntity. The association line from ChemicalLibrary to ChemicalEntity indicates that a ChemicalLibrary is a composite of other chemical entities. This design is described as the Composite Pattern in the GoF book (Gamma et al., 1995). The benefit of the Composite Pattern is that component and composite share the same interfaces and therefore their clients can invoke them transparently without knowing whether they are dealing with a component or a composite at run time.

> **The Composite Pattern: It allows you to compose objects into tree structures to represent part–whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.**

The source code of ChemicalEntity class is as follows:

```
package com.abcpharma.crs.entity;

import java.util.*;

public abstract class ChemicalEntity {
   private int id;
   private Calendar creationDate = null;
```

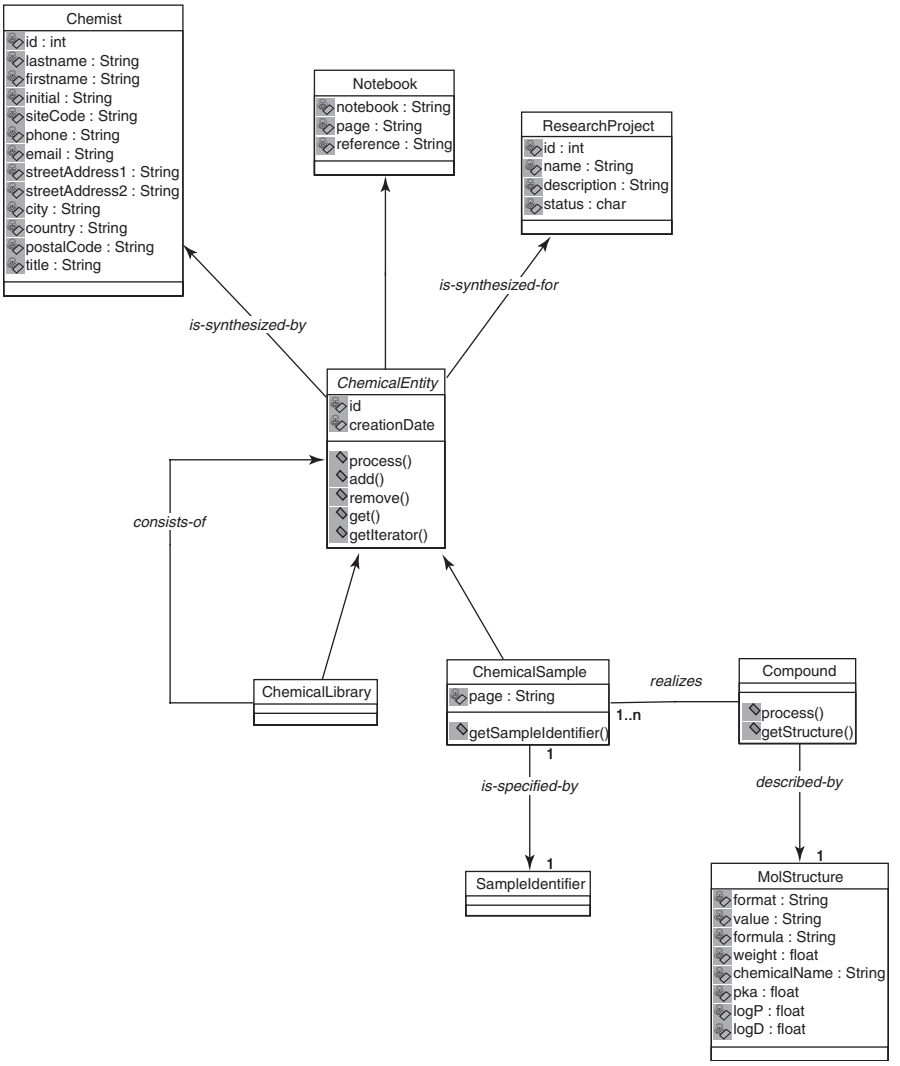


Figure 12.14 The class diagram of ChemicalLibrary–ChemicalSample composite hierarchy.

```
private Notebook notebook = null;

public ChemicalEntity(){

}

public ChemicalEntity(int id, Calendar creationDate){
   this.id = id;
   this.creationDate = creationDate;
}
```

```
    public int getID(){
        return id;
    }

    public void setID(int id){
        this.id = id;
    }

    public Calendar getCreationDate(){
        return creationDate;
    }

    public void setCreationDate(Calendar creationDate){
        this.creationDate = creationDate;
    }

    public Notebook getNotebookInfo(){
        return notebook;
    }

    public void setNotebook(Notebook notebook){
        this.notebook = notebook;
    }

    public void add(ChemicalEntity entity){
        throw new UnsupportedOperationException();
    }

    public void remove(ChemicalEntity entity){
        throw new UnsupportedOperationException();
    }

    public ChemicalEntity get(int index){
        throw new UnsupportedOperationException();
    }

    public Iterator getIterator(){
        throw new UnsupportedOperationException();
    }

    public abstract void process();
}
```

Notice that the add(), get(), remove(), and getIterator() methods do nothing but throw UnsupportedOperationException. This is because these operations are not available in the leaf objects in the composite hierarchy such as ChemicalSample. They only make sense in the composite objects such as ChemicalLibrary. Defining these methods in the component class makes the design compliant with The Liskov Substitution Principle (Martin, 2003).

The Liskov Substitution Principle: Subtype must be substitutable for the base types.

If we do not include these methods in the component class and leave them to the composite class only, the leaf and the composite will have different interfaces and the client has to determine which type it is dealing with at runtime and downcast the object to composite in order to call the aggregation operations such as add(), remove(), get(), and getIterator(). Also notice that the process() method is defined as abstract and is left to the composite and leaf classes to implement.

The ChemicalSample class is implemented as the leaf in the Composite Pattern:

```
package com.abcpharma.crs.entity;

import java.util.*;

public class ChemicalSample extends ChemicalEntity {

   private Compound compound = null;

   public ChemicalSample() {

   }

   public ChemicalSample(int id, Calendar creationDate, Compound compound) {
      super (id, creationDate);
      this.compound = compound;
   }

   public void process() {
      compound.process();
   }
}
```

Notice how the process() method is implemented—it calls the process() method on the Compound object, which is a member variable of the ChemicalSample object. As described at the beginning of this chapter, the Compound object is a finite state machine. Its process() method is implemented based on what state the Compound object is in and is responsible for the state transition of the Compound object.

The ChemicalLibrary class is implemented as the composite in the Composite Pattern:

```
package com.abcpharma.crs.entity;

import java.util.*;
```

```
public class ChemicalLibrary extends ChemicalEntity {
   private List elements = new ArrayList();

   public ChemicalLibrary(){

   }

   public ChemicalLibrary(int id, Calendar creationDate){
      super(id, creationDate);
   }

   public void add(ChemicalEntity entity){
      elements.add(entity);
   }

   public void remove(ChemicalEntity entity){
      elements.remove(entity);
   }

   public ChemicalEntity get(int index){
      return (ChemicalEntity) elements.get(index);
   }

   public Iterator getIterator(){
      return elements.iterator();
   }

   public void process() {
      for(int i = 0; i < elements.size(); i++){
         ((ChemicalEntity) elements.get(i)).process();
      }
   }
}
```

As ChemicalLibrary is a composite class, it implements all aggregation methods—add(), get(), remove(), and getIterator(). Its process() method iterates through all of its components, which in this case are the ChemicalSample objects, and calls their process() method. When you register a library with multiple compounds, each compound goes through its lifecycle depending on the quality of its structure. If the structure is perfect according to the chemistry conventions, it becomes ready to be registered immediately. Otherwise, it has to go through other states such as Invalid State, Corrected State, and Valid State before it becomes ready to be registered. The GUI display strategy should be based on the compound lifecycle as well. For corrected and invalid compounds, they may require the chemist's intervention for manual corrections. Therefore, they should

be reported to the chemist before they become valid and ready to be registered.

12.2.4 The Chemical Lab Sample Identifier

The lab sample identifier (LSI) is an important piece of information for medicinal chemists. They use LSI as a compound identifier to communicate with their peer chemists and biologists who perform assay screening. LSI is also used to track the compound when it goes to preclinical development.

Although LSI is a unique identifier of compound samples, it is worth noting that LSI should not be used as the primary key of the chemical sample in the compound database because it has business meanings and its value may change. If a compound structure was registered wrong and has to be corrected, the uniqueness checking may return a different result and the LSI may have to change to maintain data integrity. For the same reason, LSI should not be used as a reference (foreign key) to the compound data in other databases such as the assay screening database. The primary key, on the other hand, is usually a sequence number that has no business meanings and never changes. Using LSI as a primary key or as a foreign key increases data maintenance costs and should be avoided if possible.

LSI usually is composed of five parts:

- Prefix—a letter that specifies the source or type of the chemical sample.
- Base—specifies the parent structure in the compound.
- Form—for a given parent, specifies different salt forms, different radio isomers, or different formulations.
- Check—a checksum that is derived from the Prefix, Base, and Form combination.
- Batch—for a given Prefix, Base, and Form combination, specifies the actual physical sample of the compound that is synthesized in a specific chemical reaction.

Different organizations may have slightly different LSI representations, but the basic structure should be similar.

In this chapter, we assume the LSI takes the form of

P-XXXXXXXXX-XXXCXXX

where P is the Prefix, XXXXXXXXX is the Base, XXX is the Form, C is the Check, and XXX is the Batch.

Because each chemical sample has a molecule structure that represents its chemical characteristics, and each compound has a parent that represents its core structure, there are also three types of LSI—Parent ID, Compound ID, and Sample ID.

- Parent ID—combination of Prefix and Base. Uniquely identifies a parent structure.
- Compound ID—combination of Parent ID, Form, and Check. Uniquely identifies the complete structure, including salt, radio isomer, or formulation.
- Sample ID—combination of Compound ID and Batch. Uniquely identifies the physical sample.

Based on the above analysis, we come to the conclusion that Sample ID and Compound ID form a "Has-A" relationship. The same is true between Compound ID and Parent ID. At the same time, Compound ID and Parent ID also form an "Is-A" relationship in that both of them have a Prefix and a Base as their attributes. The same is true between Sample ID and Compound ID in that they both have Parent ID, Form, and Check. This type of relationship is similar to the ChemicalSample–ChemicalLibrary relationship and can be represented by the GoF Composite Pattern (Gamma et al., 1995) as shown in Figure 12.15.

It is a good practice to represent LSI as objects rather than as strings because LSI is a data structure that has attributes and behaviors. However, creating objects is an expensive operation. There will be times when you want to simply treat the LSI as a string and parse its constituents. This is the reason behind creating a utility class called LsiUtil. All of its methods are static. All LSI objects use LsiUtil to parse LSI strings.

The source code of LsiUtil is as follows, and you will see later how ParentID, CompoundID, and SampleID objects use it to parse LSI string to get its constituents in their reset() methods:

```
package com.abcpharma.crs.lsi;

public class LsiUtil {

  public static String getPrefix(String input)
      throws IllegalArgumentException {
    if (input != null && input.length() > Lsi.PREFIX_LENGTH) {
      return input.substring(0, Lsi.PREFIX_LENGTH);
    } else
      throw new IllegalArgumentException(
        "The input parameter does not have a valid prefix.");
  }
```

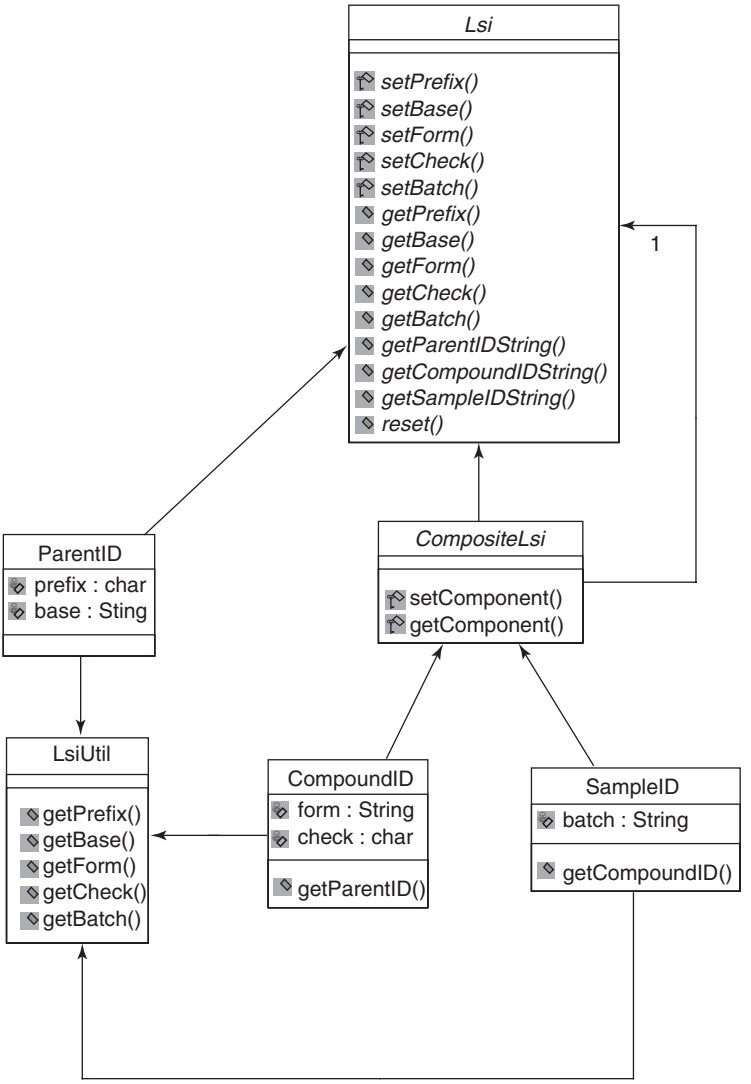


Figure 12.15 The LSI class diagram.

```
public static String getBase(String input) throws IllegalArgumentException {
   if (input != null && input.length() >= Lsi.PARENTID_LENGTH) {
      return input.substring(Lsi.PREFIX_LENGTH + 1,
            Lsi.PARENTID_LENGTH);
   } else
      throw new IllegalArgumentException(
            "The input parameter does not have a valid
            base.");
}
```

```
public static String getForm(String input) throws IllegalArgumentException {
   if (input != null && input.length() >= Lsi.COMPOUNDID_LENGTH) {
      return input.substring(Lsi.PARENTID_LENGTH + 1,
            Lsi.COMPOUNDID_LENGTH - 1);
   } else
      throw new IllegalArgumentException(
            "The input parameter does not have a valid
            form.");
}

public static char getCheck(String input) throws IllegalArgumentException {
   if (input != null && input.length() >= Lsi.COMPOUNDID_LENGTH) {
      return input.charAt(Lsi.PARENTID_LENGTH + 1 +Lsi. FORM_
      LENGTH);
   } else
      throw new IllegalArgumentException(
            "The input parameter does not have a valid
            checksum.");
}

public static String getBatch(String input) throws IllegalArgumentException {
   if (input != null && input.length() == Lsi.SAMPLEID_LENGTH) {
      return input.substring(Lsi.COMPOUNDID_LENGTH,
            Lsi.SAMPLEID_LENGTH);
   } else
      throw new IllegalArgumentException(
            "The input parameter does not have a valid
            batch.");
}

public static String getParentIDString(String input)
      throws IllegalArgumentException {
   if (input != null && input.length() >= Lsi.PARENTID_LENGTH) {
      return input.substring(0, Lsi.PARENTID_LENGTH);
   } else
      throw new IllegalArgumentException(
            "The input parameter does not have a valid
            parent id.");
}

public static String getCompoundIDString(String input)
      throws IllegalArgumentException {
   if (input != null && input.length() >= Lsi.COMPOUNDID_LENGTH) {
      return input.substring(0, Lsi.COMPOUNDID_LENGTH);
   } else
      throw new IllegalArgumentException(
            "The input parameter does not have a valid
            compound id.");
```

```
    }

    public static String getSampleIDString(String input)
            throws IllegalArgumentException {
        if (input != null && input.length() == Lsi.SAMPLEID_LENGTH) {
            return input;
        } else
            throw new IllegalArgumentException(
                    "The input parameter does not have a valid
                    sample id.");
    }
}
```

At the very top of Figure 12.15 is an abstract class Lsi. Lsi is the base class of all concrete Lsi classes—Parent ID, Compound ID and Sample ID. All of its methods are declared abstract. It defines common interfaces of the component and composite in the Lsi class hierarchy. Parent ID is the smallest possible unit and therefore is a component. Both Compound ID and Sample ID can have another type of Lsi as a component and therefore are composites.

The source code of the abstract base class Lsi is as follows:

```
package com.abcpharma.crs.lsi;

public abstract class Lsi implements Cloneable {

    public static final int PREFIX_LENGTH = 1;

    public static final int BASE_LENGTH = 9;

    public static final int FORM_LENGTH = 3;

    public static final int CHECK_LENGTH = 1;

    public static final int BATCH_LENGTH = 3;

    public static final int PARENTID_LENGTH = PREFIX_LENGTH + 1 +
    BASE_LENGTH;

    public static final int COMPOUNDID_LENGTH = PARENTID_LENGTH + 1
            + FORM_LENGTH + CHECK_LENGTH;

    public static final int SAMPLEID_LENGTH = COMPOUNDID_LENGTH +
    BATCH_LENGTH;

    public abstract Object clone();

    public String toString() {
```

```
    StringBuffer buffer = new StringBuffer(SAMPLEID_LENGTH);
    buffer.append(getPrefix()).append('-').append(getBase());
    try {
        String form = getForm();
        char check = getCheck();
        buffer.append('-').append(form).append(check);
        buffer.append(getBatch());
    } catch (UnsupportedOperationException ex) {
        ex.printStackTrace();
    }
    return buffer.toString();
}

public int hashCode() {
    return toString().hashCode();
}

public boolean equals(Object obj) {
    if (obj == null) {
        return false;
}

if (obj == this) {
    return true;
}

if (obj.getClass() != this.getClass()) {
    return false;
}

return obj.toString().equals(this.toString());
}

abstract void reset(String input);

protected void setComponent(Lsi lsi) throws UnsupportedOperationException {
    throw new UnsupportedOperationException(
            "AbstractLSI object does not have component");
}

protected Lsi getComponent() throws UnsupportedOperationException {
    throw new UnsupportedOperationException(
            "AbstractLSI object does not have component");
}

public abstract String getPrefix();

public abstract String getBase();

public abstract String getForm() throws IllegalArgumentException;
```

```
    public abstract char getCheck() throws UnsupportedOperationException;

    public abstract String getBatch() throws UnsupportedOperationException;

    public abstract String getParentIDString();

    public abstract String getCompoundIDString()
      throws UnsupportedOperationException;

    public abstract String getSampleIDString()
      throws UnsupportedOperationException;

    protected abstract void setPrefix(String prefix);

    protected abstract void setBase(String base);

    protected abstract void setForm(String form)
      throws UnsupportedOperationException;

    protected abstract void setCheck(char check)
      throws UnsupportedOperationException;

    protected abstract void setBatch(String batch)
      throws UnsupportedOperationException;
}
```

The Lsi class first defines some constants (static final variables), the lengths of each component of the LSI objects. We use these "global"constants to parse the input of the various methods in the Lsi class library instead of hard coding them in each and every parsing logic. A better approach is to ready these constants from a configuration file. The Lsi class provides implementations of Java canonical methods such has equals(), toString(), and hashCode() and declares getters and setters of LSI elements, but they are left to the concrete classes to implement.

ParentID is the smallest unit of LSI. The source code of the ParentID class is as follows:

```
package com.abcpharma.crs.lsi;

public class ParentID extends Lsi {
    private String prefix = null;

    private String base = null;

    ParentID(String input) {
      reset(input);
}
```

```
public Object clone() {
   return new ParentID(toString());
}

public String getPrefix() {
   return prefix;
}

public String getBase() {
   return base;
}

public String getForm() throws UnsupportedOperationException {
   throw new UnsupportedOperationException("A ParentID object has no Form");
}

public char getCheck() throws UnsupportedOperationException {
   throw new UnsupportedOperationException(
            "A ParentID object has no Check");
}

public String getBatch() throws UnsupportedOperationException {
   throw new UnsupportedOperationException(
            "A ParentID object has no Batch");
}

public String getParentIDString() {
   return toString();
}

public String getCompoundIDString() throws UnsupportedOperationException {
   throw new UnsupportedOperationException(
            "ParentID.getCompoundIDString() not supported");
}

public String getSampleIDString() throws UnsupportedOperationException {
   throw new UnsupportedOperationException(
            "ParentID.getSampleIDString() not supported");
}

void reset(String input) {
   setPrefix(LsiUtil.getPrefix(input));
   setBase(LsiUtil.getBase(input));
}

protected void setPrefix(String prefix) {
   this.prefix = prefix;
}
```

```
    protected void setBase(String base) {
        this.base = base;
    }

    protected void setForm(String form) throws UnsupportedOperationException {
        throw new UnsupportedOperationException("A ParentID object has no Form");
    }

    protected void setCheck(char check) throws UnsupportedOperationException {
        throw new UnsupportedOperationException(
                "A ParentID object has no Check");
    }

    protected void setBatch(String batch) throws UnsupportedOperationException {
        throw new UnsupportedOperationException(
                "A ParentID object has no Batch");
    }
}
```

The ParentID class has two instance variables: prefix and base. It also defines their getters and setters. For attributes that are not part of ParentID, their getters and setters simply throw UnsupportedOperationException.

CompositeLsi is an abstract superclass of CompoundID and SampleID classes. It is the abstract composite in the LSI class hierarchy:

```
package com.abcpharma.crs.lsi;

public abstract class CompositeLsi extends Lsi {
    private Lsi component = null;

    protected void setComponent(Lsi lsi) {
        component = lsi;
    }

    protected Lsi getComponent() {
        return component;
    }
}
```

CompositeLsi has a component as instance variable and its setter and getter. In the case of CompoundID, the component is its ParentID. In the case of SampleID, the component is its CompoundID.

The source code of CompoundID class is as follows:

```
package com.abcpharma.crs.lsi;

public class CompoundID extends CompositeLsi {
```

```
    private char check = (char) 0;

    private String form = null;

    CompoundID(String input) {
      reset(input);
    }

    public Object clone() {
      return new CompoundID(toString());
    }

    public ParentID getParentID() {
      return (ParentID) getComponent();
    }

    public String getPrefix() {
      return getParentID().getPrefix();
    }

    public String getBase() {
      return getParentID().getBase();
    }

    public String getForm() throws UnsupportedOperationException {
      if (form == null) {
        throw new UnsupportedOperationException("This LSI has no
        form");
      }
      return form;
    }

    public char getCheck() throws UnsupportedOperationException {
      if (check == (char) 0) {
        throw new UnsupportedOperationException("This LSI has no
        check");
      }
      return check;
    }

    public String getBatch() throws UnsupportedOperationException {
      throw new UnsupportedOperationException(
            "A CompoundID object has no Batch");
    }

    public String getParentIDString() {
      return getParentID().getParentIDString();
    }

public String getCompoundIDString() throws UnsupportedOperationException {
```

```
    if (form == null) {
  throw new UnsupportedOperationException(
            "This LSI has no compound ID");
    }
    return toString();
  }

  public String getSampleIDString() throws UnsupportedOperationException {
    throw new UnsupportedOperationException(
            "CompoundID.getSampleIDString() not supported");
  }

  void reset(String input) {
    if (getComponent() == null) {
      setComponent(new ParentID(input));
    } else {
      getComponent().reset(input);
  }

    if (input.length() >= COMPOUNDID_LENGTH) {
      setForm(LsiUtil.getForm(input));
      setCheck(LsiUtil.getCheck(input));
    }
  }

  protected void setPrefix(String prefix) {
    getParentID().setPrefix(prefix);
  }

  protected void setBase(String base) {
    getParentID().setBase(base);
  }

  protected void setForm(String form) {
    this.form = form;
  }

  protected void setCheck(char check) {
    this.check = check;
  }

  protected void setBatch(String batch) throws UnsupportedOperationException {
    throw new UnsupportedOperationException(
            "A CompoundID object has no Batch");
    }
  }
```

The CompoundID class has two more instance variables: form and check, and their getters and setters. For prefix and base operations, it delegates to its

component—the ParentID object. Because batch does not belong to CompoundID, its getter and setter throw UnsupportedOperationException.

The last member in the LSI class hierarchy is the SampleID class:

```
package com.abcpharma.crs.lsi;

public class SampleID extends CompositeLsi {
   private String batch = null;

   SampleID(String input) {
      reset(input);
   }

   public CompoundID getCompoundID() {
      return (CompoundID) getComponent();
   }

   public Object clone() {
      return new SampleID(toString());
   }

   public ParentID getParentID() {
      return getCompoundID().getParentID();
   }

   public String getPrefix() {
      return getParentID().getPrefix();
   }

   public String getBase() {
      return getParentID().getBase();
   }

   public String getForm() throws UnsupportedOperationException {
      return getCompoundID().getForm();
   }

   public char getCheck() throws UnsupportedOperationException {
      return getCompoundID().getCheck();
   }

   public String getBatch() throws UnsupportedOperationException {
      if (batch == null) {
         throw new UnsupportedOperationException("This LSI has no batch");
      }
      return batch;
   }
```

```
public String getParentIDString() {
   return getParentID().getParentIDString();
}

public String getCompoundIDString() throws UnsupportedOperationException {
   return getCompoundID().getCompoundIDString();
}

public String getSampleIDString() throws UnsupportedOperationException {
   if (batch == null) {
      throw new UnsupportedOperationException("This LSI has no
      sample ID");
   }
   return toString();
}

void reset(String input) {
   if (getComponent() == null) {
      setComponent(new CompoundID(input));
   } else {
      getComponent().reset(input);
}

   if (input.length() == SAMPLEID_LENGTH) {
      setBatch(LsiUtil.getBatch(input));
   }
}

   protected void setPrefix(String prefix) {
      getParentID().setPrefix(prefix);
}

   protected void setBase(String base) {
      getParentID().setBase(base);
}

   protected void setForm(String form) throws UnsupportedOperationException {
      getCompoundID().setForm(form);
}

   protected void setCheck(char check) throws UnsupportedOperationException {
      getCompoundID().setCheck(check);
}

   protected void setBatch(String batch) {
      this.batch = batch;
   }
}
```

You might have noticed that the constructors of the above LSI classes are not public, which means they cannot be instantiated from outside the LSI

package. This restriction is because the string argument these constructors take has to be validated to make sure it is in the right format before a valid LSI object can be created. We assign the responsibility of instantiating LSI objects to an LSIFactory object on behalf of LSI clients. The LSIFactory object uses LSIValidator objects to validate the input before the LSI objects are created on behalf of the clients.

Figure 12.16 is the class diagram that shows LSIFactory and the LSIValidator classes:

Figure 12.16 also shows the ICheckLetterGenerator interface. The CompoundIDValidator object uses it to make sure the compound id's checksum matches what is derived from its prefix, base, and form using the checksum algorithm.

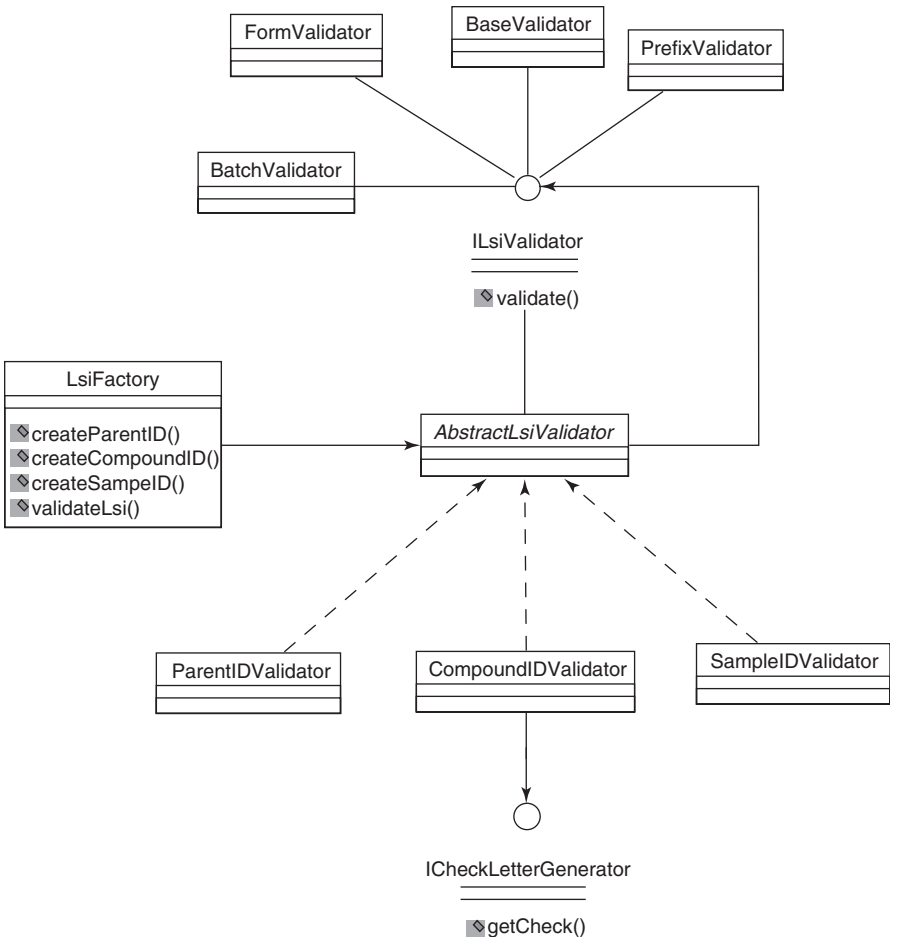


Figure 12.16 The class diagram of LSI Factory and the LSI Validator.

The source code of the ILsiFactory interface is as follows:

```
package com.abcpharma.crs.lsi;

public interface ILsiFactory {
   public Lsi createLsi(String input)
      throws IllegalArgumentException;

   public Lsi createParentID(String input)
      throws IllegalArgumentException;

   public Lsi createCompoundID(String input)
      throws IllegalArgumentException;

   public Lsi createSampleID(String input)
      throws IllegalArgumentException;
}
```

The ILsiFactory interface declares four methods for creating ParentID, CompoundID, and SampleID objects and the Lsi object in general.

The source code of the abstract LsiFactory class is as follows:

```
package com.abcpharma.crs.lsi;

public abstract class LsiFactory implements ILsiFactory {
   public Lsi createLsi(String input)
         throws IllegalArgumentException {
      switch (input.length()) {
      case Lsi.PARENTID_LENGTH:
         return createParentID(input);
      case Lsi.COMPOUNDID_LENGTH:
         return createCompoundID(input);
      case Lsi.SAMPLEID_LENGTH:
         return createSampleID(input);
      default:
         throw new IllegalArgumentException("Invalid length of LSI "
         + input);
      }
   }
}
```

The abstract factory class implements the createLsi() method. It simply calls the createParentID(), createCompoundID(), or createSampleID() method depending on the length of the input.

The LsiFactoryImpl class is as follows. It implements the rest of the methods in the ILsiFactory interface:

```
package com.abcpharma.crs.lsi;
```

```
import com.abcpharma.crs.lsi.validator.*;

public class LsiFactoryImpl extends LsiFactory {
   private static final LsiFactoryImpl instance = new LsiFactoryImpl();

   private LsiFactoryImpl() {
   }

   public static LsiFactoryImpl getInstance() {
      return instance;
   }

   private void validateLsi(ILsiValidator validator, String input)
         throws IllegalArgumentException {
      if (input == null) {
         throw new IllegalArgumentException("LSI is null");
      }

      input = input.toUpperCase();
      validator.validate(input);
   }

   public Lsi createParentID(String input)
         throws IllegalArgumentException {
      validateLsi(ParentIDValidator.getInstance(), input);
      return new ParentID(input);
   }

   public Lsi createCompoundID(String input)
         throws IllegalArgumentException {
      this.validateLsi(CompoundIDValidator.getInstance(), input);
      return new CompoundID(input);
   }

   public Lsi createSampleID(String input)
         throws IllegalArgumentException{
      validateLsi(SampleIDValidator.getInstance(), input);
      return new SampleID(input);
   }
}
```

Notice that each factory method uses the corresponding validator object to validate the input before the LSI object is created. This process makes sure that all LSI objects are in the correct format.

The ILsiValidator in Figure 12.16 defines the LsiValidator interface:

```
package com.abcpharma.crs.lsi.validator;

public interface ILsiValidator {
```

```
    public void validate(String input) throws IllegalArgumentException;
}
```

This simple interface has only one method: validate(). It makes perfect sense because this is all its clients need it to do.

The source code of the LsiValidator abstract class is as follows:

```
package com.abcpharma.crs.lsi.validator;

import java.util.*;

public abstract class LsiValidator implements ILsiValidator {
    protected List validators = new ArrayList();

    protected int length;

    public void validate(String input) throws IllegalArgumentException {
        if (input == null) {
            throw new IllegalArgumentException("Could not validate null LSI");
        }

        if (input.length() < length) {
            throw new IllegalArgumentException("Invalid length of LSI " + input);
        }

        for (int i = 0; i < validators.size(); i++) {
            ILsiValidator validator = (ILsiValidator) validators.get(i);
            validator.validate(input);
        }
    }
}
```

The abstract class LsiValidator has two attributes: the correct length of the LSI that is being validated and a list of validators. Each element in the list is the validator of one particular component in the LSI. LsiValidator implements the common validation logic: Check null value and the length of the input and call the validator objects to validate the LSI components.

The source code of the ParentIDValidator is as follows:

```
package com.abcpharma.crs.lsi.validator;

import com.abcpharma.crs.lsi.*;

public class ParentIDValidator extends LsiValidator {
    private static final ParentIDValidator instance = new ParentIDValidator();
```

```
    private ParentIDValidator() {
      length = Lsi.PARENTID_LENGTH;
      validators.add(PrefixValidator.getInstance());
      validators.add(BaseValidator.getInstance());
    }

    public static ParentIDValidator getInstance() {
      return instance;
    }

    public void validate(String input) throws IllegalArgumentException {
      super.validate(input); // validate prefix and base

      // check hyphen between prefix and base
      if (input.charAt(Lsi.PREFIX_LENGTH) != '-') {
        throw new IllegalArgumentException(
            "No hyphen between the prefix and base in
            LSI " + input);
      }
    }
}
```

ParentID validator is a Singleton. Its constructor sets the length of parent id and the list of validator objects that are responsible for validating the components of the parent id—PrefixValidator and BaseValidator. Its validate() method invokes the validate() method in the base abstract class—LsiValidator, which iterates through the validator list, invokes their validate() method, and makes sure there is a hyphen between prefix and base in the input.

The source code of the CompoundIDValidator is as follows:

```
package com.abcpharma.crs.lsi.validator;

import com.abcpharma.crs.lsi.*;

public class CompoundIDValidator extends LsiValidator {
    private static final CompoundIDValidator instance = new CompoundIDValidator();

    private CompoundIDValidator() {
      length = Lsi.COMPOUNDID_LENGTH;
      validators.add(ParentIDValidator.getInstance());
      validators.add(FormValidator.getInstance());
    }

    public static CompoundIDValidator getInstance() {
      return instance;
    }

    public void validate(String input) throws IllegalArgumentException {
```

```
        super.validate(input); // validate prefix and base

        // validate check
        int checkPos = Lsi.COMPOUNDID_LENGTH - 1;
        ICheckLetterGenerator checkLetterGenerator = CheckLetterGenerator
            .getCheckLetterGenerator();
        char validCheckLetter = checkLetterGenerator.getCheck(input);
        if (input.charAt(checkPos) != validCheckLetter) {
            throw new IllegalArgumentException(
                "Check letter is not valid in LSI " + input);
        }

        // check for hyphen between base and form
        if (input.charAt(Lsi.PARENTID_LENGTH) != '-') {
            throw new IllegalArgumentException(
                "No hyphen between the base and form exists in LSI "
                + input);
        }
    }
}
```

Similar to the ParentIDValidator, its constructor sets the length and the list of validator objects—ParentIDValidator and FormValidator. The validate() method does two more things: making sure the check letter matches what it derived from the CheckLetterGenerator and that there is a hyphen between the parent id and the form.

The source code of the SampleIDValidator is as follows:

```
package com.abcpharma.crs.lsi.validator;

import com.abcpharma.crs.lsi.*;

public class SampleIDValidator extends LsiValidator {
    private static final SampleIDValidator instance = new SampleIDValidator();

    private SampleIDValidator() {
        length = Lsi.SAMPLEID_LENGTH;
        validators.add(CompoundIDValidator.getInstance());
        validators.add(BatchValidator.getInstance());
    }

    public static SampleIDValidator getInstance() {
        return instance;
    }
}
```

Similar to the ParentIDValidator and CompoundIDValidator, the constructor sets the length and the list of validator objects—CompoundIDValidator and BatchValidator. Once the validator list is set correctly, there is no additional

validation logic required in SampleIDValidator, and therefore, there is no need to override the validate() method.

The above discussion is the Lab Sample Identifier framework. I skipped PrefixValidator, BaseValidator, FormValidator, and BatchValidator because their rules may be different in different organizations.

12.2.5 Data Binder Object Model

In a compound registration system, compound data can be imported from data files such as SD File, XML File, or Molfile. Alternatively, data can be entered from the presentation layer using a structure drawing package such as ISISDraw or ChemDraw. These data, once imported to the system, need to be bound to the domain objects in order for the system to process them efficiently. To support a variety of data sources, a Data Binder API is needed to decouple the system from specific format of input data and make it easily extensible to support other data input formats down the road. Figure 12.17 is the class diagram of the Data Binder API.

The interface DataBinder defines two methods:

- bind(), which takes a String as input and returns a List object—a list of ChemicalSample objects that are represented in the input. The input String can be an SD File, an XML File, and so on.

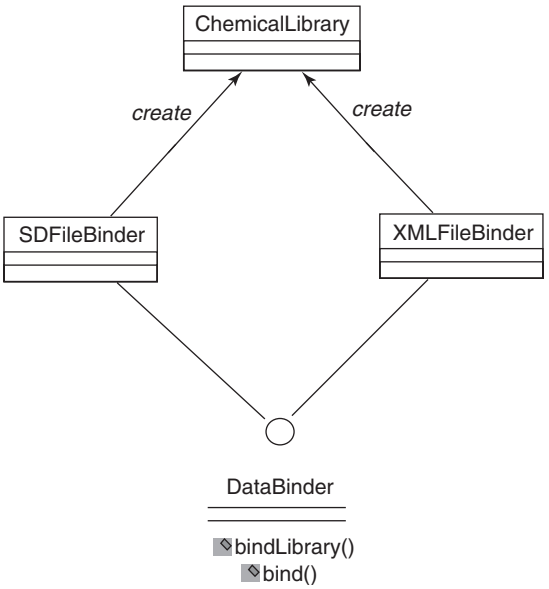


Figure 12.17 The class diagram of the Data Binder API.

- bindLibrary(), which takes a String as input and returns a ChemiclLibrary object—a library of ChemicalSample that is represented by the input.

The source code of the DataBinder interface is as follows:

```
package com.abcpharma.crs.databinder;

import java.util.*;

import com.abcpharma.crs.*;

public interface DataBinder {

   public ChemicalLibrary bindLibrary(String input);

   public List bind(String input);

}
```

The SDFileBinder implementation takes an SD File as input, whereas the XMLFileBinder implementation takes an XML File as input.

Parsing the SD File is a tricky job because the SD File is not a well-structured format, in contrast to XML. An SD File example with two records is as follows:

```
   −ISIS− 07240513032D

   13  13  0  0  0  0  0  0  0  0999  V2000
   −1.1556   −0.1291    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   −1.1568   −0.9565    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   −0.4419   −1.3694    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.2745   −0.9560    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.2716   −0.1255    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   −0.4437    0.2836    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   −0.4462    1.1086    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   −1.1667    1.5250    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.9846    0.2897    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    1.7006   −0.1201    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    2.4135    0.2951    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.7037   −0.9451    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.2677    1.5221    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
   1  2  2  0  0  0  0
   6  7  1  0  0  0  0
   3  4  2  0  0  0  0
   7  8  2  0  0  0  0
   5  9  1  0  0  0  0
   4  5  1  0  0  0  0
   9  10  1  0  0  0  0
   2  3  1  0  0  0  0
```

```
 10  11  1  0  0  0  0
 5  6  2  0  0  0  0
 10  12  2  0  0  0  0
 6  1  1  0  0  0  0
 7  13  1  0  0  0  0
M END
> <ID>
2

> <Salt_type>
TFA

> <Salt_Coefficient>
2

> <page>
86

$$$$

−ISIS− 07240513072D

12 13 0 0 0 0 0 0 0 0999 V2000
−0.3723   −0.3166   0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
−0.3734   −1.1440   0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  0.3414   −1.5569   0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1.0578   −1.1435   0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1.0550   −0.3130   0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  0.3396    0.0961   0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1.7648    0.1009   0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  2.4811   −0.3106   0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  3.1936    0.1039   0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  3.1909    0.9297   0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  2.4699    1.3394   0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1.7604    0.9225   0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
6  1  1  0  0  0  0
1  2  2  0  0  0  0
3  4  2  0  0  0  0
7  8  2  0  0  0  0
8  9  1  0  0  0  0
4  5  1  0  0  0  0
9  10  2  0  0  0  0
2  3  1  0  0  0  0
10  11  1  0  0  0  0
5  6  2  0  0  0  0
11  12  2  0  0  0  0
12  7  1  0  0  0  0
5  7  1  0  0  0  0
M END
```

```
> <ID>
1

> <Parent_Coefficient>
1

> <Salt_type1>
TFA

> <Salt_Coefficient1>
1

> <page>
86

> <reference>
12345

$$$$
```

The source code of the SDFileBinder class binds the above SD File to a ChemicalLibrary object or a list of ChemicalSample objects, as follows:

```
package com.abcpharma.crs.databinder;

import java.util.*;
import java.io.*;

import com.abcpharma.crs.*;
import com.abcpharma.crs.Molstructure.*;

public class SDFileBinder implements DataBinder {

   private static final SDFileBinder instance = new SDFileBinder();
   private static final String REC_DELIMITER = "$$$$";
   private static final String MOL_DELIMITER = "M END";

   private SDFileBinder(){
   }

   public static SDFileBinder getInstance(){
      return instance;
   }

   public ChemicalLibrary bindLibrary(String input){
      ChemicalLibrary library = new ChemicalLibrary();
      int index = -1;
      while(input.length() > 1){
         int endOfRecord = input.indexOf(REC_DELIMITER) + 4;
         String record = input.substring(0, endOfRecord);
```

```
        library.add(buildRecord(index++, record));
        input = '\n' + input.substring(endOfRecord).trim();
    }
    return library;
}
public List bind(String input){
    List samples = new ArrayList();
    int index = -1;
    while(input.length() > 1){
        int endOfRecord = input.indexOf(REC_DELIMITER) + 4;
        String record = input.substring(0, endOfRecord);
        samples.add(buildRecord(index++, record));
        input = '\n' + input.substring(endOfRecord).trim();
    }
    return samples;
}
protected ChemicalSample buildRecord(int id, String record){
    Compound compound = new Compound();
    ChemicalSample sample = new ChemicalSample(id, new Gregorian Calendar(), com-
    pound);
    StringBuffer attributeValue = new StringBuffer();
    String attributeName = null;

    while (!record.equals(REC_DELIMITER)) {
        int endOfLine = record.indexOf('\n') + 1;
        String line = record.substring(0, endOfLine);
        record = record.substring(endOfLine);
        if (line.startsWith(MOL_DELIMITER)) { //the end of mol string
            attributeValue.append(line);
            Molstructure structure = new Molstructure
            (Molstructure.MOLFILE,
            attributeValue.toString());
            compound.setMolstructure(structure);
        }
        else if (line.startsWith("> <") || (line.startsWith("> <"))) { //get
        attribute name
            attributeName = (line.substring(line.indexOf("<") + 1,
            line.lastIndexOf(">"))).toUpperCase(); //take field to
            uppercase
            attributeName.trim();
            attributeValue.setLength(0);
        }
        else if ((line.length() == 0 || line.equals("\n")) &&
        attributeName != null) {//get attribute
            value other than mofile
            String value = attributeValue.substring(0,
            attributeValue.length()).toUpperCase().
            trim();
            attributeName = null;
        /*
```

```
            insert code here to set compound or sample attribute
            */
            //attributeValue.setLength(0);
            }
            else {
               attributeValue.append(line);
            }
         }
         return sample;
      }
}
```

Binding an XML File to CRS objects is a much simpler task because open source tools are available that can help us to do the work. Castor XML Mapping and Apache Xerces are the two examples. Using the Data Binder API and implementing an XMLBinder that uses any open source XML parser should do the work.

12.2.6 Application Controller

Before a chemical sample or a chemical library can be registered into a database, a series of business logic and flow of screens have to be involved to make sure the compounds are in "good shape" before being registered into the database. Some of these screens are displayed only if the compounds are transitioned into certain states. A standard way of implementing a system like this is to use the Model−View−Controller framework. As Martin Fowler (2003a) pointed out, "to some degree the various Model–View–Controller input controllers can make some of these (flow) decisions, but as an application gets more complex this can lead to duplicated code as several controllers for different screens need to know what to do in a certain situation." The Application Controller is used to address the following problem:

> **Application Controller: You can remove this duplication by placing all flow logic in an Application Controller. Input controllers then ask the Application Controller for the appropriate commands for execution against a model and the correct view to use depending on the application context.**

The Application Controller decides what business logic to run and which view to forward to depending on the outcome of the business logic. In CRS, there are three core business logics: load input data, process chemistry intelligence on the compounds, and submit the compounds for registration. The Application Controller holds domain command objects as well as names and locations of views (e.g., URL) and returns them to the Front Controller such as

a Java Servlet. Figure 12.18 is a class diagram that illustrates the Application Controller and its collaborators.

This design uses another GoF Design Pattern—the Command Pattern.

The Command Pattern: It encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

The idea behind Command Pattern is that the requester (Front Controller) invokes the execute() method on the command object. The command object invokes an action on the receiver (e.g., DataBinder) object that the requester has no knowledge of and therefore decouples the requester from the receiver. We will see The Command Pattern in action soon.

The source code of the CRSCommand interface is as follows:

```
package com.abcpharma.crs;

import java.util.*;

public interface CRSCommand {

   public void execute(Map parameters) throws IllegalArgumentException;
}
```

The Command Pattern is simple. It only declares one method execute(), which delegates the request to a receiver object.

The source code of CRSApplicationController class is as follows:

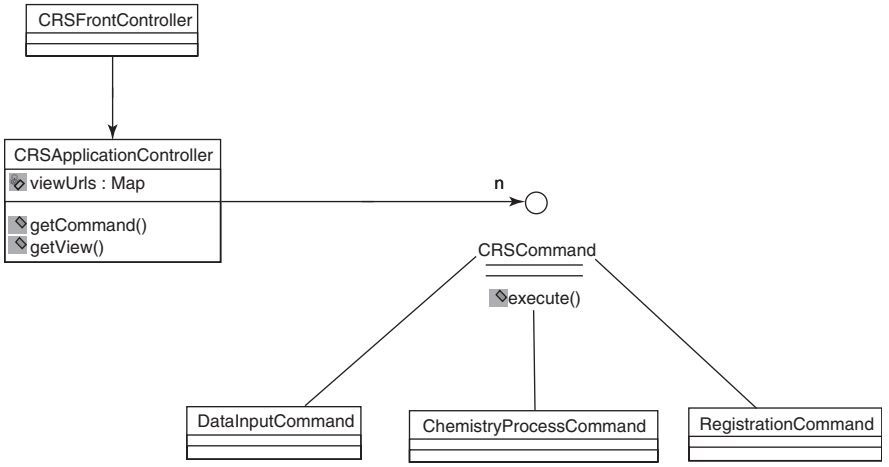


Figure 12.18 The class diagram of Application Controller and its collaborators.

```
package com.abcpharma.crs;
import java.util.*;
import com.abcpharma.crs.databinder.*;
public class CRSApplicationController {
   private Map commands = new HashMap();
   private Map views = new HashMap();
   private static final CRSApplicationController instance = new CRSApplication
   Controller();
   private CRSApplicationController(){
      commands.put("datainput", DataInputCommand.getInstance());
      commands.put("xmlfile", XMLBinder.getInstance());
      commands.put("chemistry", LibraryChemistryProcess Command. get
      Instance());
   }

   public static CRSApplicationController getInstance(){
      return instance;
   }

   public CRSCommand getCommand(String commandName){
      return (CRSCommand) commands.get(commandName);
   }

   public String getView(String stateName){
      return (String) views.get(stateName);
   }
}
```

The CRSApplicationController object has a Map of Command object that maps the key—the name of the request to the Command object that handles the request. It also holds another Map that maps the key—the alias of the view to the URL of the view. The getCommand() method takes the request name and returns the Command object that is responsible for handling the request. The getView() method takes the alias of the view and returns the URL of the view.

Figure 12.19 is a sequence diagram that shows how the Front Controller, the Application Controller, and the Command objects interact with each other in the Load SD File transaction.

Load SD File Transaction

After the user imports an SD File and submits the form, the Front Controller receives a request to upload the SD File. The Front Controller asks the Application Controller for the appropriate Command object that can handle the SD File upload request. The Application Controller returns a Data Input Command. The Front Controller invokes the execute() method on the DataInputCommand object. The DataInputCommand object invokes the SD File Binder's bind() method, which returns a List of ChemicalSample

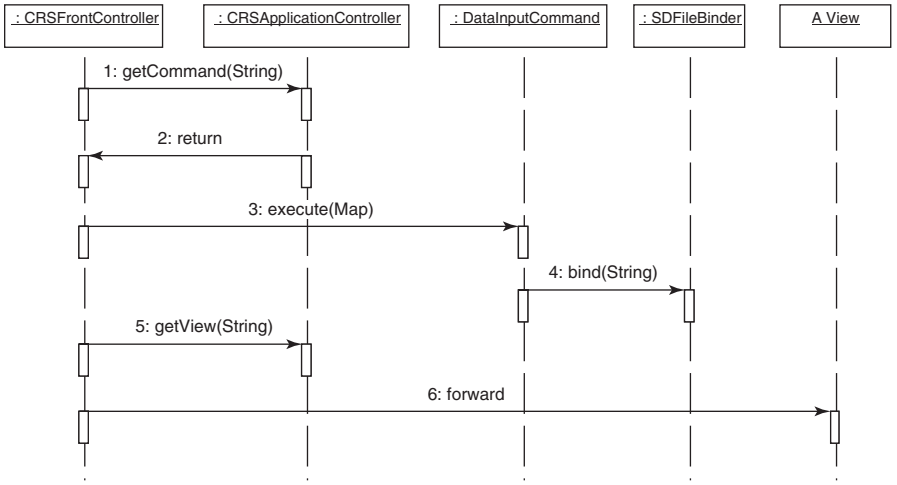


Figure 12.19 The sequence diagram of Application Controller and DataInputCommand in the Load SD File transaction.

objects. The Front Controller then makes another call to the Application Controller to get the next view the system should display and forward the flow to that view that finishes the Load SD File Transaction. The Front Controller should also save the ChemicalSampleList into the HTTPSession object so that the next transaction can retrieve it and continue the work.

The source code of DataInputCommand is as follows:

```
package com.abcpharma.crs;
import java.util.*;
import com.abcpharma.crs.databinder.*;
public class DataInputCommand implements CRSCommand{

   private static final DataInputCommand instance = new DataInputCommand();
      private DataInputCommand(){
   }

   public static DataInputCommand getInstance(){
      return instance;
   }

   public void execute(Map input){
      ChemicalLibrary library = null;
      if(input.get("sdfile") != null){
         library = SDFileBinder.getInstance().bindLibrary((String) input.
         get("SDFile"));
      }
      else if(input.get("xmlfile") != null){
         library = XMLBinder.getInstance().bindLibrary((String) input.
         get("SDFile"));
```

```
    }

    else{
      throw new IllegalArgumentException("Neither an SD file nor an XML file can be
      found
        in the input");
    }

    ((UserSession) input.get("usersession")).setLibrary(library);
  }
}
```

The execute() method uses the DataBinder object as the request receiver to bind the input SD File or XML File to the ChemicalLibrary object. Because registering a compound or a library requires several interactions between the user and the system, the state has to be maintained between those interactions. As a result, the DataInputCommand object saves the library object into a UserSession object for future access.

Figure 12.20 is the sequence diagram of the chemistry intelligence transaction.

Chemistry Intelligence Transaction

Every time the chemist edits the data and submits the form, the Front Controller receives a request and updates the ChemicalSample objects in the session. Depending on the type of request and the states of the Compound objects, it retrieves a Command object from the ApplicationController. If this is in the middle of structure QC or salt handling process, the Application Controller returns a ChemistryProcessCommand object. The FrontController

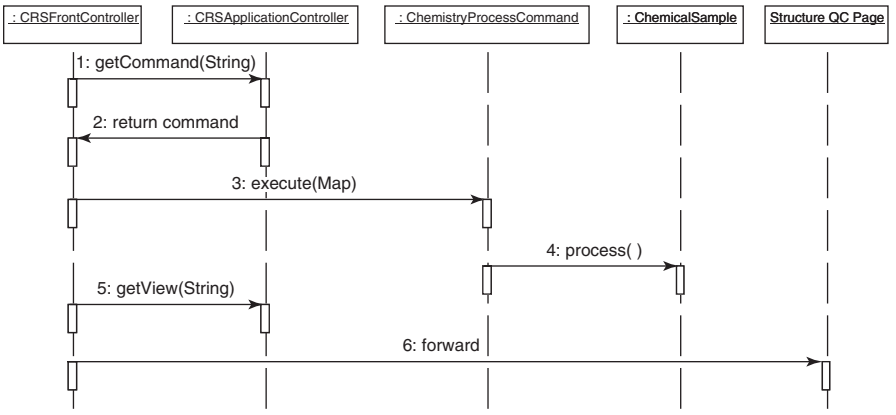


Figure 12.20 The sequence diagram of the chemistry intelligence transaction.

invokes the ChemistryProcessCommand object's execute() method, which invokes the process() method on the ChemicalSample objects. As described earlier in this chapter, the process() method of the ChemicalSample object relies on the Compound object's State object to do the work.

The source code of the LibraryChemistryProcessCommand, which processes chemistry logic on a chemical library, is as follows. (You can have another Command object to process discrete compounds. The idea is the same.)

```
package com.abcpharma.crs;

import java.util.Map;

public class LibraryChemistryProcessCommand implements CRSCommand{
   private static final LibraryChemistryProcessCommand instance = new Library
   ChemistryProcessCommand();

   private LibraryChemistryProcessCommand(){

   }

   public static LibraryChemistryProcessCommand getInstance(){
      return instance;
   }

   public void execute(Map input){
      if(input.get("library") == null){
         throw new IllegalArgumentException("Library not found ");
      }
      ((ChemicalLibrary) input.get("library")).process();
   }
}
```

The execute() method simply calls the ChemicalLibrary object's process() method. If you remember how the process() method in ChemicalLibrary is implemented, all it does is iterate through all its ChemicalSample objects and invoke their process() methods, which depends on what state the Compound object is in, and either decompose the compound's structure into fragments, QC the structure, or attach salt to the parent structure.

Figure 12.21 is the sequence diagram of the submit registration transaction.

Submit Registration Transaction

When all ChemicalSample objects are in the Ready To Be Registered State and all required ancillary data are entered, the user can submit the Chemical Samples for registration. The ApplicationController returns a Registration Command object to the FrontController. The FrontController invokes the

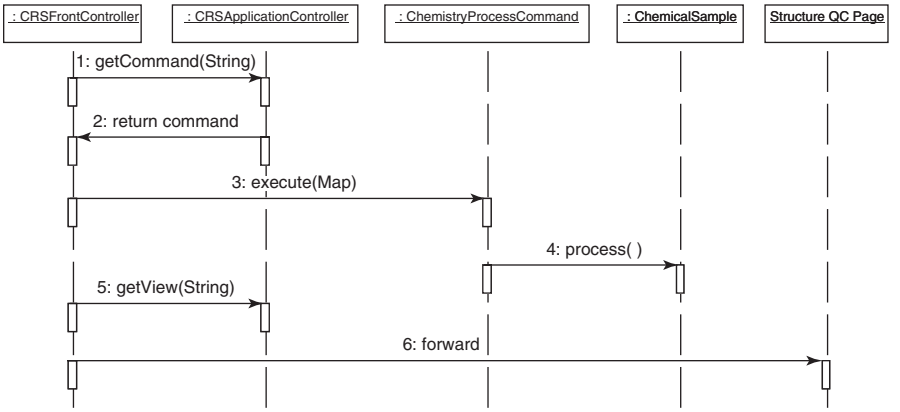


Figure 12.21 The sequence diagram of the submit registration transaction.

RegistrationCommand object's execute() method, which invokes the register() method on the RegistrationService object. The RegistrationService object does the job of registering the compounds, including generating sample identifiers and persisting compounds into the database. The Application Controller returns the RegistrationReportView to the FrontController to display the registration report.

The source code of the RegistrationCommand class is as follows:

```
package com.abcpharma.crs;

import java.util.Map;

public class RegistrationCommand implements CRSCommand{
   private static final RegistrationCommand instance = new RegistrationCommand();

   private RegistrationCommand(){

   }

   public static RegistrationCommand getInstance(){
      return instance;
   }

   public void execute(Map input){
      RegistrationService.getInstance().register((ChemicalLibrary)input.get
      ("library"));
   }

}
```

12.2.7 Registration Service

The last step of the compound registration workflow persists compound data into the database. This step is accomplished by a RegistrationService object along with a data persistence layer, which will be discussed in the next chapter.

A robust compound registration system should support two registration modes—synchronous and asynchronous. The synchronous mode locks up a user's application screen until the registration transaction finishes. This is fine if the number of compounds being registered is small and the processing time is short. When registering a large library of hundreds or even thousands of compounds, synchronous registration is not optimal. Asynchronous registration, on the other hand, submits the registration request to a message queue and frees the user session immediately. In this case, the RegistrationService becomes the message consumer that takes the message from the queue and processes the registration request in the background.

Figure 12.22 is the class diagram of the Registration Service.

The IRegistrationServiceDelegate interface defines three register() methods. The first method takes a single ChemicalSample object as input. The second method takes a List of ChemicalSample objects as input. The third method takes a ChemicalLibrary as input.

IRegistrationServiceDelegate has two implementations: SynchRegistrationDelegate and AsynchRegistrationDelegate. SynchRegistrationDelegate simply delegates the registration request to the RegistrationService object that does the actual work. AsynchRegistrationDelegate submits the registration request to a message queue—RegistrationMessageQueue. The consumer of the registration message—RegistrationMessageConsumer—subscribes to the message queue and, upon receiving the message, forwards it to the RegistrationService object. The DAO is the data persistence layer that will be discussed in Chapter 13. Notice that both SynchRegistrationDelegate and AsynchRegistrationDelegate simply delegate the registration request to RegistrationService. AsynchRegistrationDelegate does this through the message queue and the message consumer. They are not responsible for registering the compounds. RegistrationService is the object that does the actual work. The above design supports both synchronous and asynchronous registration modes with maximized code reuse.

RegistrationService has two private methods: uniquenessSearch(), which determines whether the structure of the compound to be registered is unique in the database; and generateSampleIdentifier(), which according to the result of uniquenessSearch() generates the sample identifier.

Figure 12.23 is the sequence diagram of the synchronous registration process.

Figure 12.24 is the sequence diagram of the asynchronous registration process.

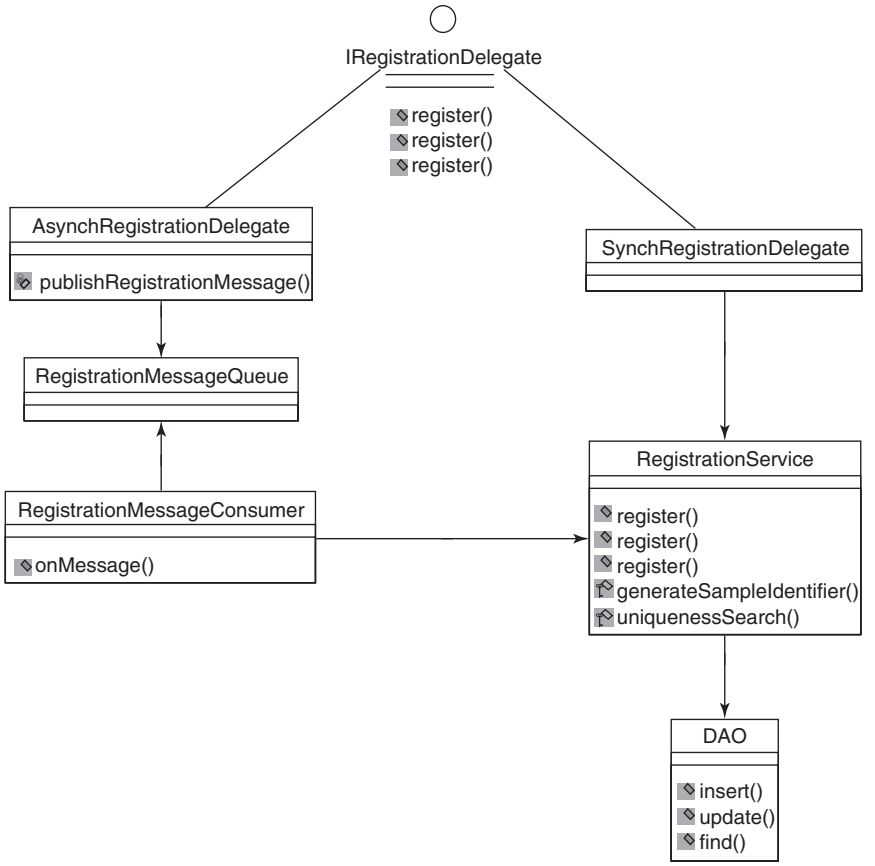


Figure 12.22 The class diagram of the Registration Service.

The Source code of the IRegistrationDelegate interface is as follows:

```
package com.abcpharma.crs.registrationservice;

import java.util.*;

import com.abcpharma.crs.entity.*;

public interface IRegistrationDelegate {

    public void register(ChemicalSample sample);

    public void register(ChemicalLibrary library);

    public void register(List samples);
}
```

The clients of the interface are given three options—register a single sample, register a list of samples, and register a sample library. These registrations can

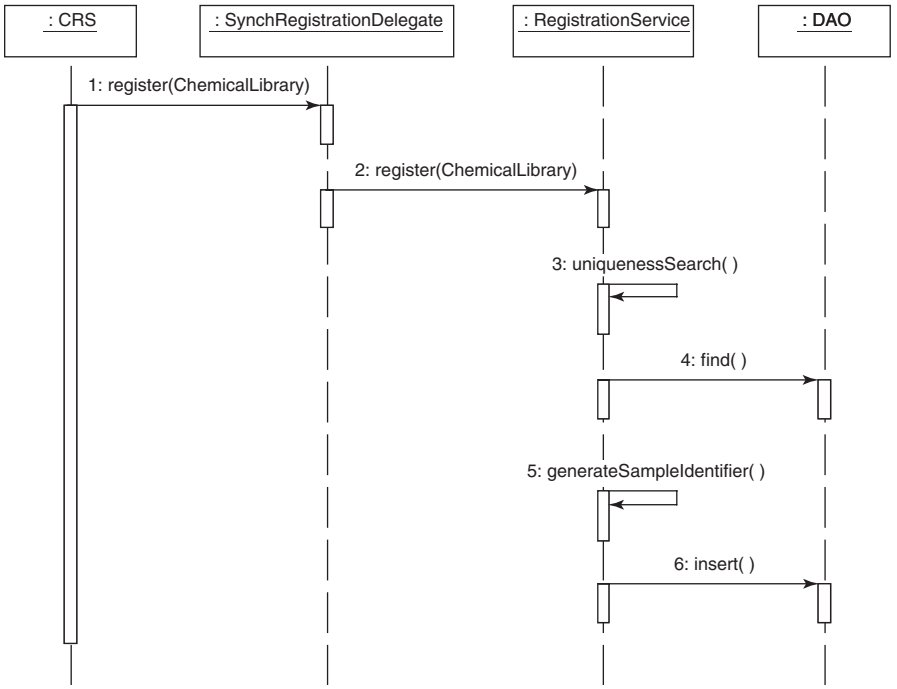


Figure 12.23 The sequence diagram of the synchronous registration process.

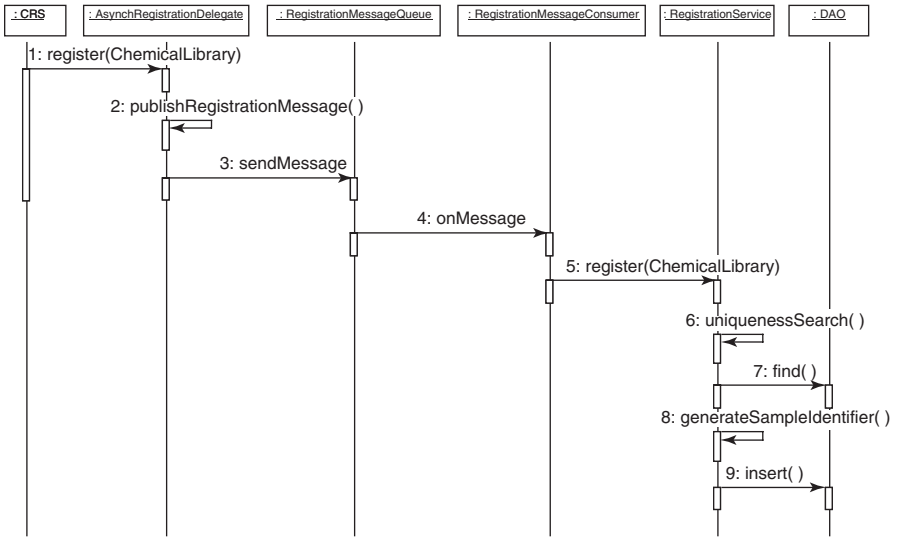


Figure 12.24 The sequence diagram of asynchronous registration process.

be processed in two different modes—synchronous or asynchronous. They are provided by the two implementations of the IRegistrationDelegate interface—SynchronousRegistrationDelegate and AsynchronousRegistrationDelegate.

The source code of the SynchronousRegistrationDelegate is as follows:

```
package com.abcpharma.crs.registrationservice;

import java.util.List;

import com.abcpharma.crs.entity.*;

public class SynchRegistrationDelegate implements IRegistrationDelegate{

   private static final SynchRegistrationDelegate instance = new Synch Registration
   Delegate();

   private SynchRegistrationDelegate(){

   }

   public static SynchRegistrationDelegate getInstance(){
      return instance;

   }

   public void register(ChemicalSample sample){
      RegistrationsService.getInstance().register(sample);
   }

   public void register(ChemicalLibrary library){
      RegistrationsService.getInstance().register(library);
   }

   public void register(List samples){
      RegistrationsService.getInstance().register(samples);
   }
}
```

The SynchronousRegistrationDelegate delegates registration requests directly to the RegistrationService object to do the work. As explained, synchronous registration is good when the number of compounds in the registration is small. When the number becomes large and the processing time is long, asynchronous registration is more desirable:

```
package com.abcpharma.crs.registrationservice;

import java.util.*;
import javax.jms.*;
import javax.naming.*;
import java.io.*;

import com.abcpharma.crs.entity.*;
```

```
public class AsynchRegistrationDelegate implements IRegistrationDelegate{

   private static final AsynchRegistrationDelegate instance = new Asynch Registration
   Delegate();

   private AsynchRegistrationDelegate(){

   }

   private void sendMessage(Serializable obj) throws NamingException, JMSException{
      InitialContext jndiContext = new InitialContext();
      QueueConnectionFactory factory = (QueueConnectionFactory)
            jndiContext.lookup("AsynchRegisterCF");
      Queue registrationQueue = (Queue) jndiContext. lookup("Asynch Register
      JMSQueue");
      QueueConnection connection = null;
      try{
            connection = factory.createQueueConnection();
            QueueSession session = connection.createQueueSession(false,
               Session.AUTO_ACKNOWLEDGE);
            QueueSender sender = session.createSender(registrationQueue);
            ObjectMessage message = session.createObjectMessage();
            message.setObject(obj);
            sender.send(message);
      }finally{
            connection.close();
      }
   }

   public static AsynchRegistrationDelegate getInstance(){
      return instance;
   }

   public void register(ChemicalSample sample){
      sendMessage((Serializable) sample);
   }

   public void register(ChemicalLibrary library){
      sendMessage((Serializable) library);
   }

   public void register(List samples){
      sendMessage((Serializable) samples);
   }
}
```

The above code assumes JMS is used as the messaging server. The register() methods call the sendMessage method() to submit the registration to a JMS message queue. Here we use a message queue rather than a message topic

because there is only one consumer of the message—the RegistrationService object. The sendMessage() method first uses JNDI to look up the Message Queue and the MessageQueueFactory. The MessageQueueFactory then creates a QueueConnection on which a QueueSession is created. The Queue Session object creates a QueueSender and an ObjectMessage. Finally, the ObjectMessage, with the registration request being set as its content, is sent to the MessageQueue by the QueueSender, which completes the transaction.

Once the message queue receives the registration request message, an event is fired off and the onMessage() method on the RegistrationMessageConsumer object is invoked that retrieves the message from the queue and processes the registration request. The source code of the RegistrationMessageConsumer is as follows:

```
package com.abcpharma.crs.registrationservice;

import javax.jms.*;

public class RegistrationMessageConsumer implements MessageListener{

   public void onMessage(Message msg) {
      ObjectMessage message = (ObjectMessage) msg;
      RegistrationService regService = RegistrationService.getInstance();
      try {
         Object obj = (Object) message.getObject();
         Class[] argTypes = {obj.getClass()};
         Object[] args = {obj};
         RegistrationService.class.getMethod("register", argTypes)
            .invoke(regService, args);
      } catch (Exception ex) {
         ex.printStackTrace();
      }
   }
}
```

The RegistrationMessageConsumer class implements the JMS MessageListener interface. It therefore has to implement the onMessage() method. The onMessage() method uses reflections to invoke the right version of the register() method in the RegistrationService class.

Finally, the source code of the RegistrationService class is as follows:

```
package com.abcpharma.crs.registrationservice;

import java.util.List;

import com.abcpharma.crs.entity.ChemicalLibrary;
import com.abcpharma.crs.entity.ChemicalSample;
```

```
public class RegistrationService {

   private static final RegistrationService instance = new RegistrationService();

   private RegistrationService(){

   }

   public static RegistrationService getInstance(){
      return instance;
   }

   public void register(ChemicalSample sample){
      //register sample
   }

   public void register(ChemicalLibrary library){
      //register sample libary
   }

   public void register(List samples){
      //register sample list
   }

}
```

As each organization has slightly different rules of registration, the implementation details of the register() methods are left blank in the above example.

12.2.8 Enterprise Java Beans

You might have wondered about Enterprise Java Beans (EJBs). The advantage of using EJBs is that the container does a lot of plumings, such as object lifecycle management, object pooling, and access control. However, EJBs are more difficult to port and test than Plain Old Java Objects (POJOs). For that reason, my recommendation is to not leave any business logic in the EJBs (especially session beans). Use EJBs as wrappers that simply delegate the method calls to POJOs. Also use Business Delegate and Service Lookup Patterns (Alur et al., 2003) to hide EJB-specific protocols from clients. The code example of the SynchRegistrationBean, which wraps the SynchRegistrationDelegate object is as follows. Both ChemicalSampleDTO and ChemicalLibraryDTO are data transfer objects (Alur et al., 2003).

```
package com.abcpharma.crs.registration.service;
import javax.ejb.*;
import java.util.*;
import com.abcpharma.crs.entity.*;
```

```
public class SynchRegistrationBean implements SessionBean {
SessionContext sessionContext;
SynchRegistrationDelegateregService;

public void ejbCreat() throws CreateException {
regService = SynchRegistrationDelegate.getInstance();
}

public void setSessionContext(SessionContextsessionContext) {
this.sessionContext = sessionContext;
}

public ChemicalSample register(ChemicalSampleDTO sample) throws Compound
RegistrationExpception{
try{
      regService.register(sample);
   }

   catch (Exception ex) {
      throw new CompoundRegistrationException(ex.getMessage());
   }
   return sample;
}

public List register(List sample) throws CompoundRegistrationException{
   try{
      regService.register(samples);
}

   catch(Exception ex) {
      throw new CompoundRegistrationException(ex.getMessage());
   }
   return samples;
}

public ChemicalLibrary register(Chemical LibraryDTO library) throws Compound
RegistrationException{
   try{
      regService.register(library);
}
   catch(Exception ex) {
      throw new CompoundRegistrationException(ex.getMessage());
      }
      return library;
   }
}
```

CHAPTER 13

Entity Dictionary

The compound registration process uses some reference data that do not change very often. Examples of these data are Research Projects, Assays, People, and Salts. These ancillary data are usually accessed by CRS using some lookup mechanism, such as by name, site, and id.

These data do not change very often, should not be entered as free text, and should introduce very little overhead into the overall registration process. It is recommended that they be cached in the application and refreshed periodically. This way, they can be accessed quickly from the memory rather than queried every time from the underline database. In memory data access is much faster than any input/output (I/O) operations, especially I/O that involves network traffic. As the dictionary data do not change very often, the performance benefit of accessing it from the cache overweighs the benefit of real-time up-to-date data.

In this chapter, we present a way of caching the Entity Dictionary—the lookup data. Figure 13.1 is the class diagram of the Entity Dictionary framework.

The EntityDictionaryDao is a **Data Access Object** that is responsible for retrieving entity dictionary data from the underlined data source. The EntityDictionaryManager is responsible for caching and refreshing entity dictionary data and for providing the API to access the entity dictionary for the clients. The diagram does not show all getter methods in the EntityDictionaryManager class, but it gives you an idea of how the entity dictionary is accessed.

The source code of the EntityDictionaryDao is as follows:

```
package com.abcpharma.crs.entitydictionary;
import java.sql.*;
import java.util.*;

import com.abcpharma.crs.entity.*;
import com.abcpharma.crs.molstructure.*;
```

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

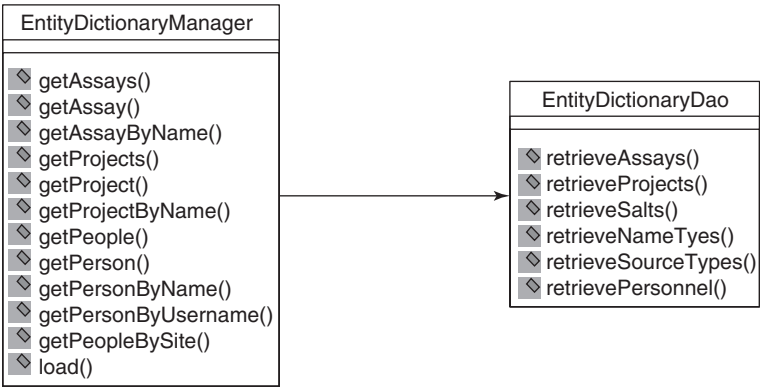


Figure 13.1 The class diagram of the Entity Dictionary.

```
public class EntityDictionaryDao {
   private static final EntityDictionaryDao instance = new EntityDictionaryDao();

   public static final int SALT_SIZE = 256;
   public static final int PERSONNEL_SIZE = 8192;
   public static final int SITE_SIZE = 32;
   public static final int PROJECT_SIZE = 512;
   public static final int ASSAY_SIZE = 1024;
   public static final int NAME_TYPE_SIZE = 8;
   public static final int SOURCE_SIZE = 8;

   public static final String ASSAY_LIST = "Assay List";
   public static final String ASSAYID_MAP = "Assay Id Map";
   public static final String ASSAYNAME_MAP = "Assay Name Map";

   public static final String PROJECT_LIST = "Project List";
   public static final String PROJECTID_MAP = "Project Id Map";
   public static final String PROJECTNAME_MAP = "Project Name Map";

   public static final String PERSON_LIST = "Person List";
   public static final String PERSONSITE_MAP = "Person Site Map";
   public static final String PERSONID_MAP = "Person Id Map";
   public static final String PERSONUSERNAME_MAP = "Person ISID Map";
   public static final String PERSONNAME_MAP = "Person Name Map";

   public static final String SITE_LIST = "Site List";

   public static final String SALTTYPE_MAP = "Salt Type Map";
   public static final String SALTID_MAP = "Salt Id Map";
   public static final String SALTFORMULA_MAP = "Salt Formula Map";

   public static final String SOURCE_LIST = "Source List";

   public static final String NAMETYPE_LIST = "Name Type List";
```

```
private Connection conn = null;

private static String assayQuery = null;
private static String projectQuery = null;
private static String personnelQuery = null;
private static String nameTypeQuery = null;
private static String sourceQuery = null;
private static String saltTypeQuery = null;

static {
  StringBuffer temp = new StringBuffer(512);

  temp.append("select assay_id, assay_name")
    .append(" from assay_view")
    .append(" order by assay_name");
  assayQuery = temp.toString();

  temp.setLength(0);
  temp.append("select name_type")
    .append(" from name_type_view");
  nameTypeQuery = temp.toString();

  temp.setLength(0);
  temp.append("select project_id, project_name")
    .append(" from research_project_view")
    .append(" order by project_name ");
  projectQuery = temp.toString();

  temp.setLength(0);
  temp.append("select person_id, username, first, middle, last")
    .append(", site, mail_drop, email_address")
    .append(" from personnel_view")
    .append(" where division = 'RESEARCH'")
    .append(" and status = 'ACTIVE'")
    .append(" order by last, first");
  personnelQuery = temp.toString();

  temp.setLength(0);
  temp.append("select source_name")
    .append(" from source_table")
    .append(" order by source_name");
  sourceQuery = temp.toString();

  temp.setLength(0);
  temp.append("select salt_id, salt_type, molfile, molweight, molformula")
    .append(" FROM salt_dictionary")
    .append(" order by salt_type");
  saltTypeQuery = temp.toString();
}
```

```
private EntityDictionaryDao() {
}

public static EntityDictionaryDao getInstance() {
  return instance;
}

public Map retrieveAssays() throws SQLException {
  List assays = new ArrayList(ASSAY_SIZE);
  Map assayByID = new HashMap();
  Map assayByName = new HashMap();

  PreparedStatement prepStmt = null;
  ResultSet rs = null;
  try {
    prepStmt = conn.prepareStatement(assayQuery);
    prepStmt.setFetchSize(ASSAY_SIZE);
    rs = prepStmt.executeQuery();
    while (rs.next()) {
      Assay assay = new Assay();
      int assayid = rs.getInt("assay_id");
      String assayname = rs.getString("assay_name");

      assay.setName(assayname);
      assay.setId(assayid);

      assays.add(assay);
      assayByID.put(new Integer(assayid), assay);
      assayByName.put(assayname, assay);
    }
  } finally {
    Util.cleanUp(conn, prepStmt, rs);
  }

  Map result = new HashMap();
  result.put(ASSAY_LIST, assays);
  result.put(ASSAYID_MAP, assayByID);
  result.put(ASSAYNAME_MAP, assayByName);
  return result;
}

public List retrieveNameTypes() throws SQLException {
  List nameTypes = new ArrayList(NAME_TYPE_SIZE);

  PreparedStatement prepStmt = null;
  ResultSet rs = null;
  try {
    prepStmt = conn.prepareStatement(nameTypeQuery);
    prepStmt.setFetchSize(NAME_TYPE_SIZE);
```

```
        rs = prepStmt.executeQuery();
        while (rs.next()) {
            nameTypes.add(rs.getString("name_type"));
        }
    } finally {
        Util.cleanUp(conn, prepStmt, rs);
    }

    return nameTypes;
}

public Map retrieveProject() throws SQLException {
    List projects = new ArrayList(PROJECT_SIZE);
    Map projectByID = new HashMap();
    Map projectByName = new HashMap();

    PreparedStatement prepStmt = null;
    ResultSet rs = null;
    try {
        prepStmt = conn.prepareStatement(projectQuery);
        prepStmt.setFetchSize(PROJECT_SIZE);
        rs = prepStmt.executeQuery();

        while (rs.next()) {
            int projectID = rs.getInt("project_id");
            String projectName = rs.getString("project_name");
            Integer id = new Integer(projectID);
            ResearchProject project = (ResearchProject) project
            ByID
                    .get(id);
            if (project == null) {
                project = new ResearchProject();
                project.setId(projectID);
                project.setName(projectName);
                projectByID.put(id, project);
                projectByName.put(projectName, project);
                projects.add(project);
            }
        }
    } finally {
        Util.cleanUp(conn, prepStmt, rs);
    }

    Map result = new HashMap();
    result.put(PROJECT_LIST, projects);
    result.put(PROJECTID_MAP, projectByID);
    result.put(PROJECTNAME_MAP, projectByName);

    return result;
}
```

```
public Map retrievePersonnel() throws SQLException {
   List persons = new ArrayList(PERSONNEL_SIZE);
   Map personBySite = new TreeMap();
   Map personByID = new HashMap();
   Map personByUsername = new HashMap();
   Map personByName = new HashMap();

   Set sites = new HashSet(SITE_SIZE);

   PreparedStatement prepStmt = null;
   ResultSet rs = null;

   try {
      prepStmt = conn.prepareStatement(personnelQuery);
      prepStmt.setFetchSize(PERSONNEL_SIZE);
      rs = prepStmt.executeQuery();
      while (rs.next()) {
         String site = rs.getString("site");
         int id = rs.getInt("person_id");
         String username = rs.getString("username");
         sites.add(site);
         Person person = new Person();
         person.setSite(site);
         person.setId(id);
         person.setUsername(username);
         person.setLast(rs.getString("last"));
         person.setFirst(rs.getString("first"));
         person.setMiddle(rs.getString("middle"));
         person.setMailDrop(rs.getString("mail_drop"));
         person.setEmail(rs.getString("email_address"));

         //for a new site, create a new entry in the personBySite
         map
         List personSite = (List) personBySite.get(site);
         if (personSite == null) {
            personSite = new ArrayList();
            personBySite.put(site, personSite);
         }
         personSite.add(person);

         //mapping individual person by personID
         personByID.put(new Integer(id), person);

         //mapping individual person by username
         personByUsername.put(username, person);

         //mapping individual person by person's fullName
         personByName.put(person.getFullName(), person);

         //constructing all of the person objects
         persons.add(person);
      }
```

```
    } finally {
      Util.cleanUp(conn, prepStmt, rs);
    }
    Map result = new HashMap();
    result.put(PERSON_LIST, persons);
    result.put(PERSONSITE_MAP, personBySite);
    result.put(PERSONID_MAP, personByID);
    result.put(PERSONUSERNAME_MAP, personByUsername);
    result.put(PERSONNAME_MAP, personByName);
    result.put(SITE_LIST, sites);
    return result;
  }

  public List retrieveSource() throws SQLException {
    List sources = new ArrayList(SOURCE_SIZE);

    PreparedStatement prepStmt = null;
    ResultSet rs = null;
    try {
      prepStmt = conn.prepareStatement(sourceQuery);
      prepStmt.setFetchSize(SOURCE_SIZE);
      rs = prepStmt.executeQuery();
      while (rs.next()) {
        sources.add(rs.getString("SOURCE_NAME"));
      }
    } finally {
      Util.cleanUp(conn, prepStmt, rs);
    }
    return sources;
  }

  public Map retrieveSalts() throws SQLException {
    Map saltByType = new HashMap(SALT_SIZE);
    Map saltById = new HashMap(SALT_SIZE);
    Map saltByFormula = new HashMap(SALT_SIZE);

    PreparedStatement prepStmt = null;
    ResultSet rs = null;
    try {
      prepStmt = conn.prepareStatement(saltTypeQuery);
      prepStmt.setFetchSize(SALT_SIZE);
      rs = prepStmt.executeQuery();
      while (rs.next()) {
        StructureFragment salt = new StructureFragment();
        String saltType = rs.getString("salt_type");

        Clob molfile = rs.getClob("molfile");
        salt.setValue(molfile.getSubString(1, (int) molfile.
        length()));
        salt.setFormat(Molstructure.MOLFILE);
        Clob formula = rs.getClob("molformula");
```

```
            salt.setFormula(formula.getSubString(1, (int) formula.
            length()));
            salt.setRole(StructureFragment.SALT_ROLE);
            salt.setId(rs.getInt("structure_id"));
            salt.setName(rs.getString("salt_type"));
            salt.setWeight(rs.getFloat("molweight"));
            saltByType.put(salt.getName(), salt);
            saltById.put(new Integer(salt.getId()), salt);
            saltByFormula.put(salt.getFormula(), salt);
          }
      } finally {
          Util.cleanUp(conn, prepStmt, rs);
      }

      Map result = new HashMap();
      result.put(SALTTYPE_MAP, saltByType);
      result.put(SALTID_MAP, saltById);
      result.put(SALTFORMULA_MAP, saltByFormula);
      return result;
   }
}
```

At the beginning of the EntityDictionaryDao, it defines a series of constants: the size and the name of each entity dictionary. Using these constants, the code allocates the Java Collection objects so that the sizes of these Collection objects do not need to expand at runtime, which causes extra CPU cycles to be wasted. Following the sizes are String constants: names that are used to look up each entity dictionary. The class also defines SQL statements that are used to query the entity dictionary as static variables and initialize them in a static block:

```
   static {
      StringBuffer temp = new StringBuffer(512);

      temp.append("select assay_id, assay_name")
          .append(" from assay_view")
          .append(" order by assay_name");
      assayQuery = temp.toString();

      temp.setLength(0);
      temp.append("select name_type")
          .append(" from name_type_view");
      nameTypeQuery = temp.toString();

      temp.setLength(0);
      temp.append("select project_id, project_name")
          .append(" from research_project_view")
          .append(" order by project_name");
      projectQuery = temp.toString();
```

```
    temp.setLength(0);
    temp.append("select person_id, username, first, middle, last")
        .append(", site, mail_drop, email_address")
        .append(" from personnel_view")
        .append(" where division = 'RESEARCH'")
        .append(" and status = 'ACTIVE'")
        .append(" order by last, first");
    personnelQuery = temp.toString();

    temp.setLength(0);
    temp.append("select source_name")
        .append(" from source_table")
        .append(" order by source_name");
    sourceQuery = temp.toString();

    temp.setLength(0);
    temp.append("select salt_id, salt_type, molfile, molweight, molformula")
        .append(" FROM salt_dictionary")
        .append(" order by salt_type");
    saltTypeQuery = temp.toString();
}
```

An alternative is to externalize these SQL statements into a configuration file and load them in from the static block.

The EntityDictionaryDao is implemented as a **Singleton** by introducing a private constructor and a static getInstance() method:

```
private EntityDictionaryDao() {
}

public static EntityDictionaryDao getInstance() {
    return instance;
}
```

The core of the EntityDictionaryDao is in the retrieve…() methods. Here we assume the entity dictionaries are stored in a relational database. They can also be accessed from other types of data sources, such as web service, XML, and flat files. The point is to transform them into something that can be accessed easily and quickly by CRS. Take a closer look at the retrievePersonnel() method. Like most other retrieve…() methods, retrievePersonnel() returns a Map. What is in the Map depends on what kind of lookups the clients want to use to access the personnel dictionary. In the context of CRS, the personnel data can be accessed by its entirety, the research site where the person is located, person id, person's full name, or person's username. Therefore, the Map that retrievePersonnel() returns has four Collections—an entire personnel list, a site–people map, a person id–person map, a person's full name–person map, and a username–person map.

The retrievePersonnel() method first executes the personnelQuery that retrieves the personnel data from the data source. It then iterates through the ResultSet and creates a Person object per each row in the ResultSet. The following code snippet shows how the personnel dictionary is built:

```
while (rs.next()) {
   String site = rs.getString("site");
   int id = rs.getInt("person_id");
   String username = rs.getString("username");
   sites.add(site);
   Person person = new Person();
   person.setSite(site);
   person.setId(id);
   person.setUsername(username);
   person.setLast(rs.getString("last"));
   person.setFirst(rs.getString("first"));
   person.setMiddle(rs.getString("middle"));
   person.setMailDrop(rs.getString("mail_drop"));
   person.setEmail(rs.getString("email_address"));

   //for a new site, create a new entry in the personBySite map
   List personSite = (List) personBySite.get(site);
   if (personSite == null) {
      personSite = new ArrayList();
      personBySite.put(site, personSite);
   }
   personSite.add(person);

   //mapping individual person by personID
   personByID.put(new Integer(id), person);

   //mapping individual person by username
   personByUsername.put(username, person);

   //mapping individual person by person's fullName
   personByName.put(person.getFullName(), person);

   //constructing all of the person objects
   persons.add(person);
}
```

The code does not rerun the query for each lookup. It accesses the data only once from the data source and adds the Person object from each row to all lookups in each iteration of the while loop. This approach offers good performance for building the entity dictionary.

In each iteration in the while loop, the code determines whether a new research site is introduced:

```
//for a new site, create a new entry in the personBySite map
List personSite = (List) personBySite.get(site);
if (personSite == null) {
   personSite = new ArrayList();
   personBySite.put(site, personSite);
}
```

If yes, then a new list is created that stores people in that site and the new site is added to the site–people map.

The whole database access code is surrounded by a try-finally block. There is no catch because the exceptions are rethrown by the code. However, the finally block is necessary because the ResultSet, Statement, and Connection objects must be closed properly whether an exception is thrown, or otherwise there will be database resource leaks. The database cleanup is implemented in a Util class. The initialization of the Connection object is purposely left out from the code. It can be either from a connection pool or from a Thread Local variable that is set earlier in the method call stack. Thread Local will be discussed further in Chapter 15.

The other retrieve…() methods are all implemented more or less in the same way as retrievePersonnel() and become self-explanatory once retrievePersonnel() is understood.

EntityDictionaryManager has two responsibilities: provide entity dictionaries to clients and refresh the entity dictionaries from the data source periodically. The first responsibility is accomplished by its public API—the getters. The second one is accomplished by its private refresh() and load…() methods, and a background thread that wakes up every 30 minutes.

The source code of the EntityDictionaryManager is as follows:

```
package com.abcpharma.crs.entitydictionary;

import java.util.*;
import java.sql.*;

import com.abcpharma.crs.entity.*;
import com.abcpharma.crs.molstructure.*;

public class EntityDictionaryManager {
   private static List assays = null;
   private static Map assayByID = null;
   private static Map assayByName = null;

   private static List nameTypes = null;

   private static List projects = null;
   private static Map projectByID = null;
   private static Map projectByName = null;

   private static List personnel = null;
```

```
private static Map personnelBySite = null;
private static Map personnelByID = null;
private static Map personnelByUsername = null;
private static Map personnelByName = null;

private static List sites = null;

private static List sources = null;

private static Map saltByType = null;
private static Map saltByID = null;
private static Map saltByFormula = null;

private static final EntityDictionaryManager instance = new
  EntityDictionaryManager();

public static int INTERVAL_TIME = 30 * 60; // 30 min
private static Thread refreshThread;

private EntityDictionaryManager() {
  loadEntityDictionary();
}

public static EntityDictionaryManager getInstance() {
  return instance;
}

static {
  refreshThread = new Thread() {
    public void run() {
      while (!isInterrupted()) {
        try {
          Thread.sleep(INTERVAL_TIME
          * 1000);
        }
        catch (InterruptedException ex) {
          System.out.println("Entity dic-
          tionary refresh thread
            interrupted..." +ex.get
            Message());
          break;
        }
        instance.refresh();
      }
    }
  };
  refreshThread.start();
}

public synchronized void interruptThread() {
```

```
    if (refreshThread != null) {
        refreshThread.interrupt();
    }
}

public void finalize() {
    interruptThread();
}

private void refresh() {
    loadEntityDictionary();
}

private void loadAssays() {
    EntityDictionaryDao dao = EntityDictionaryDao.getInstance();
    Map assayMap = new HashMap();

    try {
        assayMap = dao.retrieveAssays();
    }
    catch (SQLException e) {
        e.printStackTrace();
    }

    assays = (List) assayMap.get(EntityDictionaryDao.ASSAY_LIST);
    assayByID = (Map) assayMap.get(EntityDictionaryDao.ASSAYID
    _ MAP);
    assayByName = (Map) assayMap.get(EntityDictionaryDao.ASSAY-
    NAME_MAP);
}

private void loadNameTypes() {
    EntityDictionaryDao dao = EntityDictionaryDao.getInstance();
    try {
        nameTypes = dao.retrieveNameTypes();
    }
    catch (SQLException e) {
        e.printStackTrace();
    }
}

private void loadProjects() {
    EntityDictionaryDao dao = EntityDictionaryDao.getInstance();
    Map projectTarget = null;
    try {
        projectTarget = dao.retrieveProject();
    }
    catch (SQLException e) {
        e.printStackTrace();
    }
```

```
    projects = (List) projectTarget.get(EntityDictionaryDao.PROJECT_ LIST);
    projectByID = (Map) projectTarget.get(EntityDictionaryDao.PROJEC
    TID_MAP);
    projectByName = (Map) projectTarget.get(EntityDictionaryDao.PRO-
    JECTNAME_MAP);
  }

  private void loadPersonnel() {
    EntityDictionaryDao dao = EntityDictionaryDao.getInstance();
    Map people = null;
    try {
      people = dao.retrievePersonnel();
    }
    catch (SQLException e) {
      e.printStackTrace();
    }

    personnel = (List) people.get(EntityDictionaryDao.PERSON_LIST);
    personnelBySite = (Map) people.get(EntityDictionaryDao.PERSON-
    SITE_MAP);
    personnelByID = (Map) people.get(EntityDictionaryDao.PERSONID_
    MAP);
    personnelByUsername = (Map) people.get(EntityDictionaryDao.PER-
    SONUSERNAME_MAP);
    personnelByName = (Map) people.get(EntityDictionaryDao.PERSON-
    NAME_MAP);

    sites = (List) people.get(EntityDictionaryDao.SITE_LIST);
  }

  private void loadSource() {
    EntityDictionaryDao dao = EntityDictionaryDao.getInstance();
    try {
      sources = dao.retrieveSource();
    }

    catch (SQLException e) {
      e.printStackTrace();
    }
  }

  private void loadSalts() {
    EntityDictionaryDao dao = EntityDictionaryDao.getInstance();
    Map salt_map = null;

    try {
      salt_map = dao.retrieveSalts();
    }
    catch (SQLException e) {
```

```
            e.printStackTrace();
        }
        saltByType = (Map) salt_map.get(EntityDictionaryDao. SALTTYPE _
        MAP);
        saltByFormula = (Map) salt_map.get(EntityDictionaryDao.
            SALTFORMULA_MAP);
        saltByID = (Map) salt_map.get(EntityDictionaryDao.SALTID_MAP);
    }

    public synchronized void loadEntityDictionary() {
        loadPersonnel();
        loadAssays();
        loadNameTypes();
        loadProjects();
        loadSalts();
        loadSource();
    }

    public List getAssays() {
        synchronized (this) {
            if (assays == null) {
                loadAssays();
            }
        }
        return assays;
    }

    public Assay getAssay(Integer id) {
        synchronized (this) {
            if (assays == null) {
                loadAssays();
            }
        }
        return (Assay) assayByID.get(id);
    }

    public Assay getAssayByName(String name) {
        synchronized (this) {
            if (assays == null) {
                loadAssays();
            }
        }
        return (Assay) assayByName.get(name);
    }

    public List getNameTypes() {
        synchronized (this) {
            if (nameTypes == null) {
                loadNameTypes();
            }
```

```
    }
    return nameTypes;
  }

  public List getSites() {
    synchronized (this) {
      if (sites == null) {

        loadPersonnel();
      }
    }
    return sites;
  }

  public List getProjects() {
    synchronized (this) {
      if (projects == null) {
        loadProjects();
      }
    }
    return projects;
  }

  public ResearchProject getProjectByID(String projectID) {
    synchronized (this) {
      if (projectByID == null) {
        loadProjects();
      }
    }
    return (ResearchProject) projectByID.get(projectID);
  }

  public ResearchProject getProjectByName(String projectName) {
    synchronized (this) {
      if (projectByName == null) {
        loadProjects();
      }
    }
    return (ResearchProject) projectByName.get(projectName);
  }

  public List getProjectNames() {
    List projectNames = null;

    synchronized (this) {
      if (projects == null) {
        loadProjects();
      }
    }
    Iterator it = projects.iterator();
```

```
    while (it.hasNext()) {
        ResearchProject project = (ResearchProject) projects;
        String name = project.getName();
        projectNames.add(name);
    }
    return projectNames;
}

public List getPeople() {
    synchronized (this) {
        if (personnel == null) {
            loadPersonnel();
        }
    }
    return personnel;
}

public List getPeopleBySite(String site) {
    synchronized (this) {
        if (personnelBySite == null) {
            loadPersonnel();
        }
    }
    return (List) personnelBySite.get(site);
}

public Person getPersonByID(String personid) {
    synchronized (this) {
        if (personnelByID == null) {
            loadPersonnel();
        }
    }
    return (Person) personnelByID.get(personid);
}

public Person getPersonByUsername(String username) {
    synchronized (this) {
        if (personnelByUsername == null) {
            loadPersonnel();
        }
    }
    return (Person) personnelByUsername.get(username);
}

public Person getPersonByName(String first, String middle, String last) {
    synchronized (this) {
        if (personnelByName == null) {
            loadPersonnel();
        }
    }
```

```
    EntityDictionaryDao dao = EntityDictionaryDao.getInstance();
    String strUniqueName = Person.getFullName(first, middle, last);
    return (Person) personnelByName.get(strUniqueName);
  }

  public List getPersonNames() {
    List persons = personnel;
    List personNames = null;

    synchronized (this) {
      if (personnel == null) {
        loadPersonnel();
      }
    }

    Iterator it = persons.iterator();
    while (it.hasNext()) {
      Person person = (Person) persons;
      String first = person.getFirst();
      String middle = person.getMiddle();
      String last = person.getLast();
      personNames.add(last + ", " + first + " " + middle);
    }
    return personNames;
  }

  public List getPersonNames(String site) {
    List persons = (List) personnelBySite.get(site);
    List personNames = null;

    synchronized (this) {
      if (personnelBySite == null) {
        loadPersonnel();
      }
    }

    Iterator it = persons.iterator();
    while (it.hasNext()) {
      Person person = (Person) persons;
      String first = person.getFirst();
      String middle = person.getMiddle();
      String last = person.getLast();
      personNames.add(last + ", " + first + " " + middle); //full names
    }
    return personNames;
  }

  public List getSource() {
    synchronized (this) {
      if (sources == null) {
```

```
            loadSource();
        }
    }
    return sources;
}

public StructureFragment getSaltByType(String saltType) {
    synchronized (this) {
        if (saltByType == null) {
            loadSalts();
        }
    }
    return (StructureFragment) saltByType.get(saltType.toUpperCase());
}

public StructureFragment getSaltByID(String saltID) {
    synchronized (this) {
        if (saltByID == null) {
            loadSalts();
        }
    }
    return (StructureFragment) saltByID.get(saltID);
}

public StructureFragment getSaltByFormula(String formula) {
    synchronized (this) {
        if (saltByFormula == null) {
            loadSalts();
        }
    }
    return (StructureFragment) saltByFormula.get(formula);
}
}
```

The class first declares a set of static variables for holding the entity dictionaries. Its private constructor and the static getInstance() method make it a Singleton. Its load…() methods call EntityDictinaryDao to load the entity dictionaries from the data source. Let us again use personnel dictionary as an example. The code snippet of the loadPersonnel() method is as follows:

```
private void loadPersonnel() {
    EntityDictionaryDao dao = EntityDictionaryDao.getInstance();
    Map people = null;
    try {
        people = dao.retrievePersonnel();
    }
    catch (SQLException e) {
        e.printStackTrace();
    }
```

```
    personnel = (List) people.get(EntityDictionaryDao.PERSON_LIST);
    personnelBySite = (Map) people.get(EntityDictionaryDao.PERSONSITE_MAP);
    personnelByID = (Map) people.get(EntityDictionaryDao.PERSONID_MAP);
    personnelByUsername = (Map) people.get(EntityDictionaryDao.PERSONUSER-
    NAME_MAP);
    personnelByName = (Map) people.get(EntityDictionaryDao.PERSONNAME
    _MAP);
    sites = (List) people.get(EntityDictionaryDao.SITE_LIST);
}
```

It calls EntityDictionaryDao's retrievePersonnel() method and saves the returned personnel dictionaries into its own dictionary variables: personnel(the entire personnel list), personnelBySite(site to people map), personnelByID(person id to person map), personnelByUsername(username to person map), personnelByName(full name to person map), and sites(the research site list). Because EntityDictionaryDao's retrievePersonnel() method builds these personnel dictionaries with one single query, the retrieval process is highly efficient.

Now let us take a look at one of personnel dictionary's getter methods—getPersonnelBySite():

```
public List getPeopleBySite(String site) {
    synchronized (this) {
        if (personnelBySite == null) {
            loadPersonnel();
        }
    }
    return (List) personnelBySite.get(site);
}
```

The method takes a site name as input. It first checks whether personnelBySite is null. If yes, it calls the loadPersonnel() method to load the personnel dictionaries first. It returns the list of people at the site.

Please note that all getters in the EntityDictionaryManager have a synchronized block. This block makes sure that when a load method is being executed, all its getter counterparts are on hold. You may wonder why none of the load methods are synchronized. This is because the load methods are all private and therefore cannot be invoked other than the EntityDictionaryManager, and all invocations inside the EntityDictionaryManager are enclosed in a synchronized block. This is why the loadEntityDictionary method is synchronized as follows:

```
public synchronized void loadEntityDictionary() {
    loadPersonnel();
    loadAssays();
    loadNameTypes();
    loadProjects();
```

```
    loadSalts();
    loadSource();
}
```

The above are all good until the entity dictionaries change in the data source. What if new employees are added and new projects and assays are registered? What if some employees are terminated? These situations require the entity dictionary cache in the EntityDictionaryManager get refreshed periodically. The following static block accomplishes this task:

```
static {
    refreshThread = new Thread() {
        public void run() {
            while (!isInterrupted()) {
                try {
                    Thread.sleep(INTERVAL_TIME * 1000);
                }
                catch (InterruptedException ex) {
                    System.out.println("Entity dictionary refresh thread
                    interrupted..." +
                        ex.getMessage());
                    break;
                }
                instance.refresh();
            }
        }
    };
    refreshThread.start();
}
```

The static block is executed when the class is loaded into the JVM. Look at what the static block does. It creates an anonymous inner class of type Thread—the refreshThread. Its run() method puts the thread into sleep for a period of time (30 minutes in our code example). When it wakes up, it calls the refresh() method of the EntityDictionaryManager. The static block starts the refresh thread, which means that the entity dictionary refresh is handled by an independent thread. The thread is in an indefinite while-loop that puts the thread into sleep after every entity dictionary refresh. The thread wakes up every 30 minutes and does the next entity dictionary refresh.

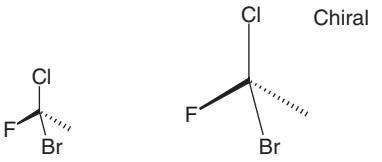
The above entity dictionary module provides fast, in-memory access of most frequently accessed data for the compound registration process while keeping the in-memory cache up-to-date. In a multitiered system, controlling network traffic is critical to its speed and user experience. The above design eliminates the need of querying the database every time a piece of data is asked for, and since the system can use the cached dictionaries to display only the valid entities to the user, it also reduces human errors.

CHAPTER 14

Chemistry Intelligence API

One critical task of any compound registration system is to make sure molecular structures are compliant with chemistry conventions. This ensures consistent representations of molecular structures in the database so that structure searches can find, and only find, the right compounds. Although different organizations may have slightly different conventions, the following ones are some of the most common that the Chemistry Intelligence API takes care of.

Add chiral flag: If the compound has a single stereo center, a chiral flag should be present:

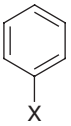


Invalid stereo bonds: Replace invalid stereo bonds with flat bonds:

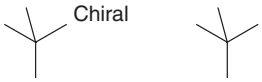
Explicit hydrogen: Hydrogen atoms should be implicit rather than explicit:

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

Invalid atom symbol should be detected:



Remove chiral flag: If a structure does not have a stereo center, the chiral flag should not be present:



Valence: Make sure each atom in the molecule has the right number of valences. The following is an example. A carbon cannot have four bonds:

Wave bond: The wave bond should be replaced by the straight bond:

Commercial software vendors provide tools that can codify these chemistry rules. MDL Cheshire, CambridgeSoft ChemOffice, and Accerlys Accord Structure Checker are three examples. The recommendation is to leverage these tools. However, you do not want to be locked into these tools (although that is what the vendors would prefer). What you want is a design that allows you to easily unplug one tool and plug in another when necessary.

In Chapter 12, we talked about Design by Interface. To do this, forget about the vendor solution for a moment and think about what a client might expect from the Chemistry Intelligence API. The client has a molecular structure and wants to inspect the structure using a chemistry convention to make sure the structure is in good quality and can be accepted by the compound database. The client wants to receive the corrected molecular structure and be informed of what has been changed as a result of the inspection. Therefore, these inspection methods take a String molfile as input and return a String array, one element of which is the corrected molfile, and the other is the reason for the correction. The source code of the MolstructureInspector interface is as follows:

```
package com.abcpharma.crs.chemintell;

public interface MolstructureInspector {

  public String[] fragment(String molfile) throws ChemistryRulesException;
```

```
    public String addFragment(String targetMol, String fragMol) throws Chemistry
    RulesException;

    public String[] executeAllRules(String molfile) throws ChemistryRulesException;

    public String[] inspectValence(String molfile) throws ChemistryRulesException;

    public String[] inspectExplicitHydrogen(String molfile) throws ChemistryRules
    Exception;

    public String[] inspectWavyBond(String molfile) throws Chemistry Rules
    Exception;

    public String[] inspectInvalidStereoBond(String molfile) throws ChemistryRules
    Exception;

    public int getFragmentCount(String molfile) throws ChemistryRulesException;

    public String neutralize(String molfile) throws ChemistryRulesException;

    public int getCharge(String molfile) throws ChemistryRulesException;

    public int getQuaternionNitrogenCount(String molfile) throws ChemistryRules
    Exception;

    public boolean isNoStructure(String molfile) throws ChemistryRulesException;

    public boolean hasInvalidAtom(String molfile) throws ChemistryRulesException;

    public String[] addChiralFlag(String molfile) throws ChemistryRulesException;

    public String removeChiralFlag(String molfile) throws ChemistryRulesException;
}
```

In addition to the above chemistry rules, it also declares other structure manipulation methods such as neutralize(), getFragmentCount(), and getCharge().

Once the interface is defined, the vendor-specific adapters can be implemented. Assume we use MDL Cheshire as the chemistry intelligence engine; we need a CheshireMolstructureInspectorImpl class. By the same token, you can also create an AccordMolstructureInspectorImpl class.

MDL Cheshire is a proprietary technology. It is composed of Cheshire Runtime, Cheshire scripts, a scripting language for writing chemistry logic, and a Cheshire Studio for developing and debugging Cheshire scripts. In addition, Cheshire has a Java JNI interface that can be called from Java programs. Figure 14.1 is a component diagram that illustrates Cheshire architecture.

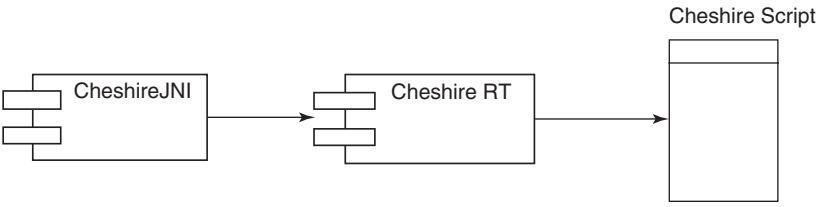


Figure 14.1 The component diagram of MDL Cheshire.

Although Cheshire does a good job applying chemistry logic on molecule structures, it has two serious drawbacks:

1. Its JNI API is unnecessarily complex.
2. It is not thread safe. Only one thread can execute Cheshire Runtime at a time. To make it worse, you can only create one Cheshire Runtime instance in a JVM!

The second drawback makes Cheshire completely single threaded. In an enterprise system, this is not acceptable. Both of the above suggest that the Cheshire environment must be carefully managed, which inspires us to create a CheshireManager class and a CheshireManagerFactory class. The responsibility of the two classes is to hide Cheshire's complexities and make sure the Cheshire environment is run by one thread only at all times. Figure 14.2 is the class diagram of the CRS Cheshire Chemistry Intelligence Framework.

Both CheshireJNI and CheshireUtil are classes in MDL's Cheshire JNI library. One goal of the CheshireManager is to hide this complexity so that the clients only interact with one **Façade** object—CheshireManager.

Figure 14.3 is a sequence diagram that illustrates how the CRS Cheshire Chemistry Intelligence Framework works.

Façade: Provide a unified interface to a set of interfaces to a subsystem. Façade defines a higher level interface that makes the subsystem easier to use.

When the CheshireMolstructureInspectorImpl object provides service to its client, the first thing it does is to get a CheshireManager object from the CheshireManagerFactory object. After the work is done by the execute() method of CheshireManager, the CheshireMolstructureInspectorImpl returns the CheshireManager object immediately back to the CheshireManagerFactory object so that the CheshireManager object is available to serve other clients.

There are two ways that CheshireManagerFactory manages the CheshireManager object. One is to store only one instance of the CheshireManager so that when it is being used by one client, other clients have to wait in a queue until it is returned to the CheshireManagerFactory. The other is to store a list

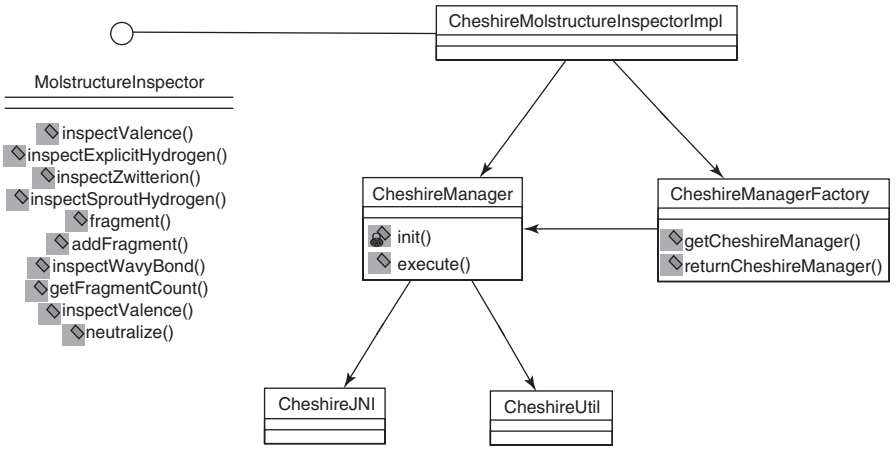


Figure 14.2 The class diagram of the CRS Cheshire Chemistry Intelligence Framework.

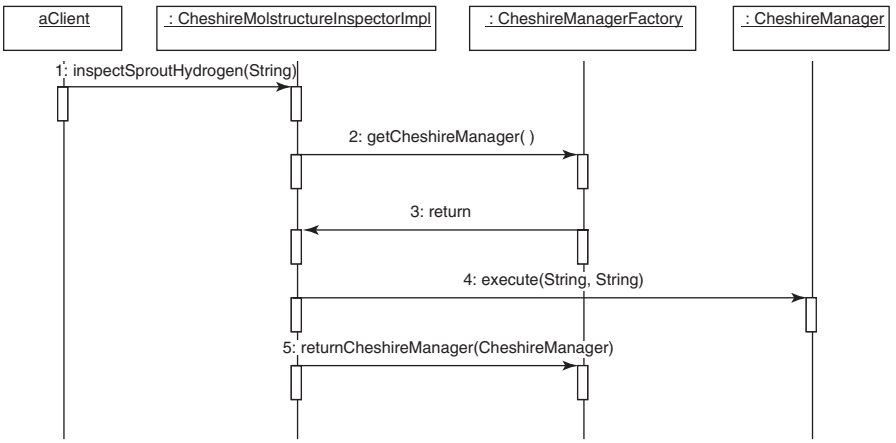


Figure 14.3 The sequence diagram of the CRS Cheshire Chemistry Intelligence Framework.

of CheshireManager objects. To do this, the CheshireManager has to be an RMI remote object because only one Cheshire JNI object can be created in each JVM. Although there is no simple answer to which way is better, I can provide some advice here. Using RMI means there will be inter-JVM remote method calls. Even if all the JVMs run on the same physical computer, there is still a significant overhead of remote method calls between the JVMs. Plus each JVM requires some computing resources. Therefore, using RMI is not efficient if the number of simultaneous transactions is low. On the other hand, a single Cheshire shared by all users and transactions does not scale. If the number of concurrent users and transactions is high, using a

pool of RMI objects is recommended. Although it takes longer to serve a single user, it certainly scales better if the number of simultaneous transactions is high.

The source code of the CheshireManager class is as follows:

```
package com.abcpharma.crs.cheshire;

import java.io.*;
import java.util.*;

import com.mdli.cheshire.jni.*;
import com.mdli.cheshire.util.*;

import com.abcpharma.crs.chemintell.*;

   public class CheshireManager{

   private CheshireJNI cheshireJni = null;

   private CheshireUtil cheshireUtil = null;

   private CheshReturnObject cheshireReturn = null;

   private long cCatId = -1;

   private static String qcScript = null;

   private static String saltScript = null;

   private static final int resetLimit = 1000;

   private static int resetCounter = 0;

   private static final String STRUCTUERE_LOAD_ERROR = " structure can not be
   loaded to Cheshire environment.";

   CheshireManager() throws ChemistryRulesException {

   init();
   }

   private void init() throws ChemistryRulesException {

      // initialize Cheshire environment
      cheshireJni = new CheshireJNI();
      if (cheshireJni == null) {
         throw new ChemistryRulesException("Can not create Cheshire
         JNI");
      }
```

```
    cheshireReturn = cheshireJni.createEnvironment();
    if (cheshireReturn == null) {
        throw new ChemistryRulesException(
            "Can not create Cheshire Return Object");
    }

    cheshireUtil = new CheshireUtil();
    if (cheshireUtil == null) {
        throw new ChemistryRulesException(("Can not create Cheshire
        Util Object"));
    }
    cCatId = cheshireJni.currentEnvironment().getId();

    if (cCatId != 1) {

        throw new ChemistryRulesException(
            "Can not create a valid Cheshire ID");
    }

    // read in Cheshire script as String

    String path = System.getProperty("user.dir");
    String sctQC = path + "/../cheshirescripts/cheshire_qc.cct";
    String sctSalt = path + "/../cheshirescripts/cheshire_salt.cct";
    try {

        qcScript = cheshireUtil.readFile(sctQC);
        saltScript = cheshireUtil.readFile(sctSalt);
    } catch (IOException e) {
        throw new ChemistryRulesException(
            "Could not read Cheshire Script file. " +
            e.getMessage());
    }
    loadScript(qcScript);
    loadScript(saltScript);
}

//load Cheshire Script
private void loadScript(String cheshireScript) throws ChemistryRulesException {
    if (!cheshireJni.runScript(cCatId, cheshireScript).isSuccess()) {
        if (!cheshireJni.repeatLastScript(cCatId).isSuccess()) {
            throw new ChemistryRulesException(
                "Could not use previously loaded
                Cheshire Script");
        }
    }
}

private void reset() throws ChemistryRulesException {
    resetCounter++;
```

```
    if (resetCounter >= resetLimit) {
      doReset();
    }

}

private void doReset() throws ChemistryRulesException {
    cheshireReturn = cheshireJni.resetEnvironment(cCatId);
    if (!cheshireReturn.isSuccess()) {
      throw new ChemistryRulesException(
            "Reset Cheshire Environment failed.");
    }
    resetCounter = 0;
    loadScript(qcScript);
    loadScript(saltScript);
}

public String[] executeRule(String molIn, String rule)
      throws ChemistryRulesException {
    String[] cheshireOutput = new String[2];
    try {
      if (!cheshireJni.loadTargetMol(cCatId, molIn).isSuccess()) {
        throw new ChemistryRulesException("executeRule: "
              + STRUCTURE_LOAD_
              ERROR);
      }
      cheshireReturn = cheshireJni.runScript(cCatId, rule);
      cheshireOutput[0] = cheshireReturn.getOutputString();
      cheshireOutput[1] = cheshireJni.unLoadTargetMol(cCatId).get
      OutputString();
    } finally {
      reset();
    }
    return cheshireOutput;
}

private String[] executeFragment(String molIn, String targetName,

      String molFrag, String ruleName, String fragName)
      throws ChemistryRulesException {
    String[] output = new String[2];
    String sctOut = null;

    try {
      if (!cheshireJni.loadMol(cCatId, molFrag, fragName).is
      Success()) {
        throw new ChemistryRulesException(fragName
              + STRUCTURE_LOAD_
              ERROR);
      }

      if (!cheshireJni.loadMol(cCatId, molIn, targetName).
      isSuccess()) {
```

```
            throw new ChemistryRulesException("Target "
                + STRUCTURE_LOAD_
                ERROR);
        }
        cheshireReturn = cheshireJni.runScript(cCatId, ruleName);
        output[0] = cheshireReturn.getOutputString();

        if (output[0].startsWith("Function") || output[0].equals("-1")) {
            throw new ChemistryRulesException("Add " + frag
            Name
                                    + " error.");
        }
        sctOut = cheshireUtil.catCheshReturn(cheshireReturn);
        output[1] = cheshireJni.unLoadMol(cCatId, targetName). get
        OutputString();
    } finally {
        reset();
    }
    return output;
}

public String[] executeAddFragment(String molIn, String fragIn)
        throws ChemistryRulesException {

    String rule = "AddFragment()";
    return executeFragment(molIn, "MIXTURE_TARGET_MOL", fragIn, rule,
            "PARENT_FRAG");
}

public String[] executeGetChargedMol(String combinedMol, String fragMol)
        throws ChemistryRulesException {

    String rule = "GetChargedMol()";
    return executeFragment(combinedMol, "COMBINED_TARGET_ MOL",
    fragMol,
                rule, "FRAG_MOL");
}

public String[] executeAddSalt(String molIn, String saltIn,
        int parentCoeff, int vhc, int nsalt)
        throws ChemistryRulesException {

    String sctOut = null;
    String rule = "ADDSALT4(" + parentCoeff + ", " + vhc + ", " + nsalt + ");";
    return executeFragment(molIn, "TARGET_MOL", saltIn, rule, "SALT_
    FRAG");
}

public List readSDFile(String sdfile) throws ChemistryRulesException {
    List molList = new ArrayList();
```

```
        if (!cheshireUtil.openSDFile(sdfile)) {
            throw new ChemistryRulesException(
                "Read SDFile ERROR: Cannot open input
                SD File\n");
        }
        while (cheshireUtil.isSDFileOpen()) {
            molList.add(cheshireUtil.readMolFromSDFile());
        }
        cheshireUtil.closeSDFile();
        return molList;
    }
}
```

The CheshireManager's constructor has no access modifier; it cannot be instantiated outside its package. Its client can acquire it only through its factory object, which makes sure its instantiation is controlled. The constructor calls the private init() method. The init() method initializes a CheshireJNI object(cheshireJni), a CheshireUtil object(cheshireUtil), a CheshireReturn object(cheshireReturn), and a Cheshire id(cCatId). The init() method makes sure the creation of these objects is successful. Otherwise it throws exceptions. Lastly, the init() method loads the Cheshire scripts that it needs to do its works from the file system. At this point, the CheshireManager is ready to receive service calls.

The executeRule() method executes the Cheshire scripts that encode chemistry rules. The two arguments are the molfile representing the structure that it needs to act upon and the name of the chemistry rule it is asked to execute. It first loads the molfile to the Cheshire Runtime through its JNI object. It then executes the chemistry rule, retrieves the output, and then unloads the molfile. All rules are already loaded from the Cheshire script files in the init() method when the CheshireManager object is created. Cheshire Runtime maintains a map between the signature of the Cheshire script procedure and the body of the procedure. This map requires that the second parameter in the executeRule(String molIn, String rule) be exactly the same as the signature of the Cheshire script procedure. For example, if the RemoveWavyBond() procedure in your Cheshire script is to be executed, the second argument of the executeRule() method in the CheshireManager must be "RemoveWavyBond()". The executeRule() method returns a String array. The first element of the array is an annotation of what gets changed as a result of the rule execution, and the second element is the updated molfile. If the structure has no issues according to the rule, the first element is an empty String and the second is the original molfile. If the structure does have problems but it requires manual corrections, the first element explains what is wrong, and the second element is the original molfile. The reset() and doReset() methods are self-explanatory—reset the Cheshire environment every so often to release resources it holds to prevent memory leaks. The readSDFile() is a utility

method. It takes an SD File as input and returns a list of molfiles contained by the SD File.

Now CheshireManager becomes the Façade of Cheshire Runtime. As described earlier in this chapter, the Cheshire environment is not thread-safe; therefore, managing the CheshireManager object becomes a challenge. There are two options: one is to create only one instance of the CheshireManager within the JVM and let all requests share it. The other is to create a pool of CheshireManager that implements the RMI Remote interface—each runs in its own JVM. We have discussed the pros and cons of each approach. Here we take the first approach. However, we still implement it using an object pool for readers who are interested in the second approach. We use Apache Org's Object Pool Library to illustrate how it works.

Apache's Object Pool Library requires an object factory for each pooled object. The factory only needs to implement one method—makeObject(). The library provides default implementations to all other methods. In our example, the CheshireManager is the pooled object, so we need a PoolableCheshire-ManagerFactory:

```
package com.abcpharma.crs.cheshire;

import org.apache.commons.pool.*;

class PoolableCheshireManagerFactory extends BasePoolableObjectFactory{
   public Object makeObject() throws Exception{
      return new CheshireManager();
   }
}
```

It is very simple: the makeObject() method simply instantiates a CheshireManager object and returns it. Notice the class is not declared public; it can be accessed only by other classes in the package. This PoolableCheshire-ManagerFactory is not to be accessed by clients of CheshireManager objects. It is used only to create the CheshireManager pool.

Next, we need a factory object that the CheshireManager's clients use to get a CheshireManager instance. We name it the CheshireManagerFactory:

```
package com.abcpharma.crs.cheshire;

import com.abcpharma.crs.chemintell.*;

import org.apache.commons.pool.*;
import org.apache.commons.pool.impl.*;

public class CheshireManagerFactory {
   private static final int POOL_SIZE = 1;
```

```
    private static final ObjectPool pool = new GenericObjectPool(new Poolable
    CheshireManagerFactory());

    private static final CheshireManagerFactory instance = new CheshireManager
    Factory();

    private CheshireManagerFactory() {
        try {
            init();
        } catch (ChemistryRulesException ex) {
            ex.printStackTrace();
        }
    }

    public static CheshireManagerFactory getInstance() {
        return instance;
    }

    private static void init() throws ChemistryRulesException {
        for (int i = 0; i < POOL_SIZE; i++) {
            try{
                pool.addObject();
            }catch(Exception ex){
                ex.printStackTrace();
            }
        }
    }

    public CheshireManager getCheshireManager() throws Exception{
        CheshireManager cm = (CheshireManager) pool.borrowObject();
        return cm;
    }
    public void returnCheshireManager(CheshireManager cm) throws Exception{
        pool.returnObject(cm);
    }
}
```

The CheshireManagerFactory is a singleton—only one instance exists. We use Apache's GenericObjectPool to maintain the CheshireManager objects. The constructor of GenericObjectPool takes an instance of PoolableCheshire ManagerFactory as input so that when its addObject() method is called, the PoolableCheshireManagerFactory object's makeObject() is called to create an instance of CheshireManager class that is saved in the GenericObjectPool. The init() method in CheshireManagerFactory does exactly that. Everything else in the CheshireManagerFactory is self-explanatory—the getCheshireManager() method checks out a CheshireManager object from the pool and returns it to

the client, and the returnCheshireManager() method checks it back into the pool.

Now we have a managed Cheshire environment that hides Cheshire complexity and allows us to implement a molstructure inspector that uses Cheshire through the CheshireManager. The source code of the CheshireMolstructureInspectorImpl is as follows:

```
package com.abcpharma.crs.cheshire;

import com.abcpharma.crs.chemintell.*;

import java.lang.reflect.*;

public class CheshireMolstructureInspectorImpl implements
    MolstructureInspector {

  private static final CheshireManagerFactory cmFactory =
    CheshireManagerFactory.getInstance();

  private static final CheshireMolstructureInspectorImpl instance = new Cheshire
  MolstructureInspectorImpl();

  private CheshireMolstructureInspectorImpl() {
  }

  public static CheshireMolstructureInspectorImpl getInstance() {
    return instance;
  }

  public String[] fragment(String molfile) throws ChemistryRulesException {
    return executeRule(molfile, "GetFragments()");
  }

  public String[] executeAllRules(String molfile) throws ChemistryRulesException {

    String retMsg = null;

    String[] feedback = new String[2];

    String[] output = new String[2];

    if (hasInvalidAtom(molfile)) {
      output[0] = "Found unknown atom.";
      output[1] = molfile;
      return output;
    }

    Method[] methods = this.getClass().getDeclaredMethods();
```

```
        Object[] args = {molfile};

        for (int i = 0; i < methods.length; i++){
            if(methods[i].getName().startsWith("inspect")){
                try{
                    feedback = (String[]) methods[i].invoke
                    (this, args);
                }catch(Exception ex){
                    ex.printStackTrace();
                }
                if (!feedback[0].equals("none")) {
                    retMsg = retMsg + feedback[0] + ", ";
            }
        }
    }

    output[0] = (retMsg == null) ? "none" : retMsg;
    output[1] = feedback[1];
    return output;
}

public boolean isNoStructure(String molfile) throws ChemistryRulesException {
    if (executeRule(molfile, "IsNoStructure()")[0].equalsIgnoreCase("yes")) {
        return true;
    } else {
        return false;
    }
}

public boolean hasInvalidAtom(String molfile) throws ChemistryRulesException {
    if (executeRule(molfile, "HasUnknownAtom()")[0].equalsIgnore
    Case("yes")) {
        return true;
    } else {
        return false;
    }
}

public String[] addChiralFlag(String molfile) throws ChemistryRulesException {
    return executeRule(molfile, "AddChiralFlag()");
}

public String removeChiralFlag(String molfile) throws ChemistryRulesException {
    return executeRule(molfile, "RemoveChiralFlag()")[1];
}

public String[] inspectStereo(String molfile) throws ChemistryRulesException {
    return executeRule(molfile, "DrawStereo()");
}
```

```
public String[] inspectValence(String molfile) throws ChemistryRulesException {
   return executeRule(molfile, "ValenceCheck()");
}

public String[] inspectExplicitHydrogen(String molfile) throws ChemistryRules
Exception {
   return executeRule(molfile, "ExplicitHydrogen()");
}

public String[] inspectInvalidStereoBond(String molfile) throws Chemistry
RulesException {
   return executeRule(molfile, "InvalidStereoBond()");
}

public String[] inspectZwitterion(String molfile) throws ChemistryRulesException {
   return executeRule(molfile, "Zwitterion()");
}

public String[] inspectWavyBond(String molfile) throws Chemistry RulesException {
   return executeRule(molfile, "RemoveWavyBond()");
}

public String[] inspectOverlappingGroups(String molfile) throws ChemistryRules
Exception {
   return executeRule(molfile, "OverlappingGroups()");
}

public int getQuaternionNitrogenCount(String molfile) throws ChemistryRules
Exception {
   int numOfQuatN = -1;
   try {
      numOfQuatN = Integer.parseInt((executeRule(molfile,
      "GetQuatN()"))[0]);
   } catch (NumberFormatException ex) {
      throw new ChemistryRulesException("Get quaternion nitrogen
      failed: "
               + ex.getMessage());
   }
   return numOfQuatN;
}

public int getCharge(String molfile) throws ChemistryRulesException {
   return Integer.parseInt(executeRule(molfile, "GetCharge()")[0]);
}

public int getFragmentCount(String molfile) throws ChemistryRulesException {
   return Integer.parseInt(executeRule(molfile, "CountFragments()")[0]);
}
```

```
    public String neutralize(String molfile) throws ChemistryRulesException {
        return executeRule(molfile, "NeutralizeParent()")[1];
    }

    private String[] executeRule(String molIn, String rule)    throws                Chemistry
    RulesException {
        String[] feedback = null;
        CheshireManager cheshireManager = null;
        try {
            cheshireManager = cmFactory.getCheshireManager();
            feedback = cheshireManager.executeRule(molIn, rule);
        }catch(Exception ex){
            ex.printStackTrace();
        } finally {
            try{
                cmFactory.returnCheshireManager(cheshireManager);
            }catch(Exception ex){
                ex.printStackTrace();
            }
        }
        return feedback;
    }

    public String addFragment(String targetMol, String fragMol) throws Chemistry
    RulesException {
        CheshireManager cheshireManager = null;
        String[] feedback = null;
        try {
            cheshireManager = cmFactory.getCheshireManager();
            feedback = cheshireManager.executeAddFragment(targetMol,
            fragMol);
        }catch(Exception ex){
            ex.printStackTrace();
        } finally {
            try{
                cmFactory.returnCheshireManager(cheshireManager);
            }catch(Exception ex){
                ex.printStackTrace();
            }
        }
        return feedback[1];
    }
}
```

This implementation of MolstructureInspector uses CheshireManager to apply chemistry rules to inspect molstructures. Take a close look at the executeRule() method, which is the core of the CheshireMolstructure-InspectorImpl class. This private method is called by almost all other

methods in the class. Three consecutive lines of code make use of the CheshireManager object:

```
cheshireManager = cmFactory.getCheshireManager();
feedback = cheshireManager.executeRule(molIn, rule);
cmFactory.returnCheshireManager(cheshireManager);
```

The first line grabs the CheshireManager object from the object pool through its factory. The second line uses the CheshireManager object to execute a rule. And the third line returns the CheshireManager object to the pool through its factory. These three lines of code make sure the CheshireManager object is obtained right before it is needed and returned right after its job is done. Also notice that these three lines are surrounded by a try—finally block to make sure the CheshireManager object is returned to the pool even if an exception is thrown.

Every chemistry rule method simply calls the executeRule() method by passing in the molfile and the name of the Cheshire script function that implements the rule. Take the wavy bond rule as an example. The following method removes wavy bonds in the molfile:

```
public String[] inspectWavyBond(String molfile) throwsChemistryRulesException {
  return executeRule(molfile, "RemoveWavyBond()");
}
```

The Cheshire script that does the actual work is as follows:

```
function RemoveWavyBond(){
  colWavy = Find(B_MARK, B_MARK_EITHER);
  colWavy.Set(B_MARK, B_MARK_NONE);

  if(colWavy.Count() > 0){
    var ret = "Removed Wavy Bond";
    return (ret);
  }
  else {
    return "none";
  }
}
```

Let us review how we accomplished the above design. We first created an interface—MolstructureInspector— which defines a set of chemistry intelligence methods— their signatures only. In doing so, we put aside any implementation details and vendor specifics. We only considered what made sense to clients. This is the essence of Design by Interface. Based on the decision of using MDL Cheshire and our knowledge of its complexity, we then implemented a CheshireManager and its factory. This approach insulates Cheshire complexities

and offloads the burden of every client having to manage the complexity. Finally, we implemented CheshireMolstructureInspectorImpl, which is an implementation of MolstructureInspector and uses Cheshire to execute chemistry logics through CheshireManager. The same process can be repeated to create an implementation of the above framework for any vendor solutions while keeping the clients of the chemistry intelligence API intact. To achieve this, there is one more piece that is missing in the puzzle. How could the client get an instance of the right implementation? The trick is a configuration file and a MolstructureInspectorFactory object. The factory object reads a configuration or a property file that instructs it on which implementation to use and returns the implementation as its interface type—MolstructureInspector. This way the client does not care which implementation is returned at run time, and therefore, the design achieves the goal of Closed for Changes. To switch from one vendor to another, simply code another MolstuctureInspector implementation and change the configuration file. It is the factory's responsibility to return the right implementation. A code template based on the above idea is as follows:

```
public class MolstrcutureInspectorFactory {
   public MolstructureInspector getStructureInspector(String key){
      //return the right implementation based on configuration.
      }
   }
```

There is always a better way. One thing can be improved in the above design. Since executeAllRules() should be the same no matter what vendor tool is used, it could be put into an abstract base class that all implementation classes extend to enhance code reuse.

CHAPTER 15

Data Persistence Layer

The Data Persistence Layer (also known as the Data Access Layer) is responsible for bridging business objects with the Data Storage Layer, which is usually a relational database. There are some obvious inconsistencies between an object model and a relational model. Relational databases are composed of tables with primary keys as the identifier of each row and foreign keys that cross-reference between tables. This can easily be mapped to the "Has-A" relationship in the object model. However, inheritance in the object model cannot easily be mapped to the relational database. A bridge between the two, therefore, is desired to decouple the object model from the relational storage. Another motivation of having the Data Persistence Layer is that the application code and data storage schema may evolve independently. Having the relational access logic embedded in the Business Layer may cause the Business Layer to change every time the relational database changes, which is not desirable.

The Data Persistence Layer can be a thick layer that resides in the application middleware or a thin layer in the application middleware combined with some stored procedures in the database. The former makes the Data Persistence Layer independent of the underline RDBMS that is being used. The latter provides better performance because the number of network calls can be reduced. Either way, a well-defined interface between the Business Layer and the Data Storage Layer is the key.

There are a few design options for the Data Persistence Layer. Here we use the Data Mapper Pattern (Fowler, 2003b). The reason is that we want to separate the domain layer and the database schema and allow them to evolve independently.

Data Mapper Pattern: A layer of Mappers that moves data between objects and database while keeping them independent of each other and the mapper itself.

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

The Data Persistence Layer performs four types of operations in the database on behalf of the Business Layer: query, insert, update, and delete. These operations are invoked by the service objects in the Business Layer. In Chapter 12, we presented one type of service—the Registration Service. There are other types of services such as Query Service and Update Service, each of which leverages the Data Persistence Layer to perform database operations.

In the case of the chemistry database, the situation is more complex. Compound data cannot be queried or updated using standard SQL. The read and write operations are encapsulated in vendor-specific Oracle data cartridges. Therefore, the Data Persistence Layer has another responsibility—hide vendor specifics with a standard interface so that the Business Layer does not care about vendor variations.

The first interface of the Data Persistence Layer is the finder interface—the interface for querying the database. Two types of queries must be supported by a chemical information database: one is numeric or text query, and the other is structure query. The first type is the same as any other type of database. The second type is where chemistry databases differ—search the database based on substructure, structure similarity, and structure flexmatch. As such, the finder interface has to support these two types of finder methods. The CompoundFinder interface is as follows:

```
package com.abcpharma.crs.registrationservice;

import com.abcpharma.crs.entity.*;

import java.util.*;

public interface CompoundFinder {
   public Compound find(int id);
   public Compound find(Integer id);
   public Compound findByCompoundId(String compoundId);
   public List findByParentId(String parentId);
   public List findBySubstructure(String structure);
   public List findBySimilarity(String structure);
   public List findByFlexmatch(String structure);
   public List findByFormulalike(String formula);
   public List findByWeightRange(float low, float high);
}
```

The first two find() methods search the compound by its primary key. The third and fourth methods search the compound by its Lab Sample Identifier—Compound ID or Parent ID (Section 12.2.6). The fifth to seventh methods search the compound by its structure. The eighth and ninth methods search the compound by its formula and weight. These are the canonical finder methods of a chemical database.

A chemical database usually has three core tables:

1. Compound table, which stores compound level information such as structure, molweight, molformula, compound id, and the like.
2. Sample table, which stores sample level information such as notebook info, project info, and sample id.
3. Compound component table, which stores the parent and salt components with which the compound is made up.

Each one of these tables will have a data mapper object. There might be other tables such as name table and distribution table, and they can be handled in the same way. These data mapper objects are used together to build the entire object tree that was described in Chapter 12 and to insert, update, and delete a record in the database.

We now introduce another interface that defines write operations to the database: the PersistenceManager class. The three write operations are insert, update, and delete:

```
package com.abcpharma.crs.registrationservice;

import java.util.*;

import com.abcpharma.crs.datamapper.DataSourceAccessException;
import com.abcpharma.crs.entity.*;

public interface PersistenceManager {
   public int insert(AbstractEntity entity) throws DataSourceAccessException;
   public void update(AbstractEntity entity) throws DataSourceAccessException;
   public void delete(AbstractEntity entity) throws DataSourceAccessException;
}
```

Now we are ready to develop the data mapper for the Compound object—the CompoundMapper class. The following class diagram describes Compound Mapper's relationship with the two interfaces (Figure 15.1).

The AbstractMapper class is introduced to abstract out the common code that can be used by all data mapper objects of entity objects (Fowler, 2003b):

```
package com.abcpharma.crs.datamapper;

import java.util.*;
import java.sql.*;

import com.abcpharma.crs.entity.*;
import com.abcpharma.crs.registrationservice.*;
import com.abcpharma.db.*;
```

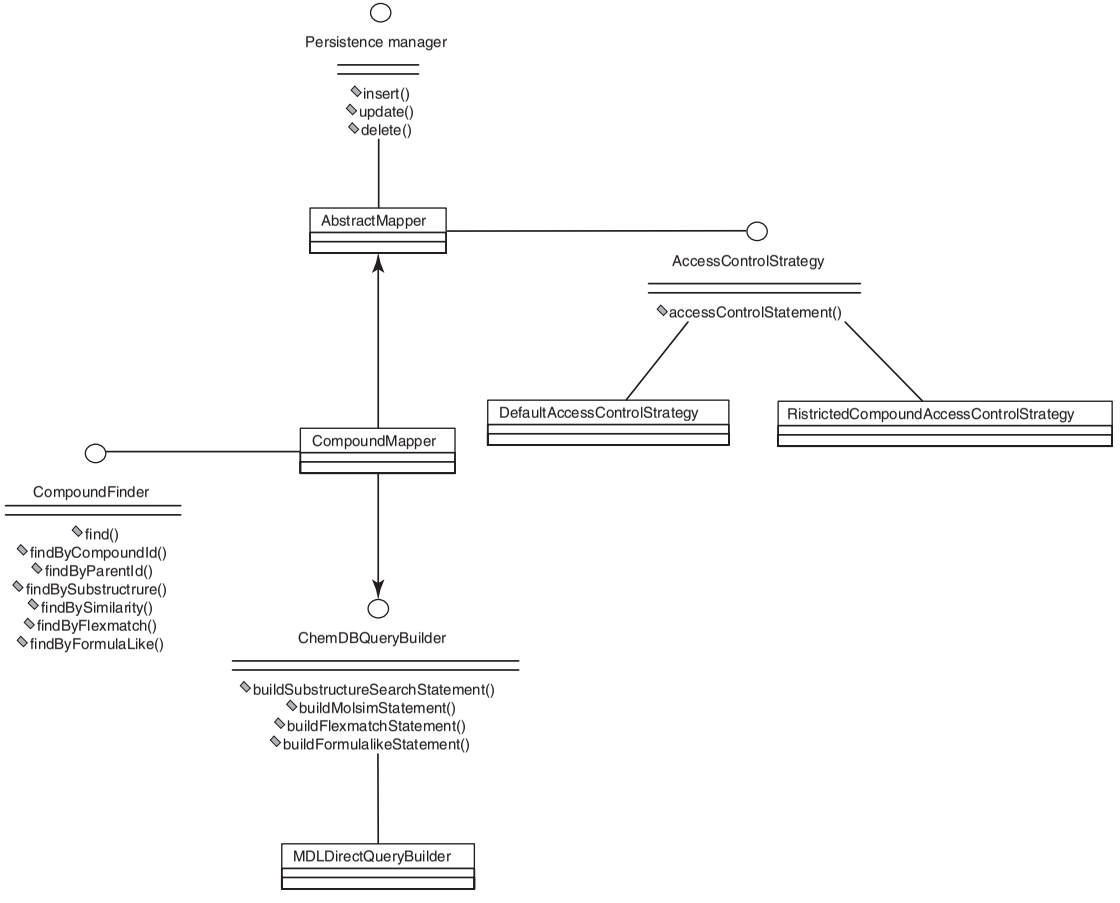


Figure 15.1 Class diagram of CompoundMapper.

```
public abstract class AbstractPersistenceManager implements PersistenceManager{
   protected Map registry = new HashMap();

   private AccessControlStrategy accessController = null;

   public AbstractPersistenceManager(AccessControlStrategy accessController){
      this.accessController = accessController;
   }

   protected String findStatement(){
      return findStatementPrimary() + accessController.access Control
      Statement();
   }

   protected String findByCorpIdStatement(){
      return findByCorpIdStatementPrimary() + accessController.access
      ControlStatement();
   }

   abstract protected String findStatementPrimary();

   abstract protected String insertStatement();

   abstract protected String findByCorpIdStatementPrimary();

   abstract protected AbstractEntity doLoad(Integer id, ResultSet rs) throws SQL
   Exception;

   abstract protected void doInsert(AbstractEntity entity, PreparedStatement insert
   Statement) throws
      SQLException;

   protected AbstractEntity abstractFind(Integer id) throws DataSource Access
   Exception{
      AbstractEntity entity = (AbstractEntity) registry.get(id);
      if(entity != null)
         return entity;


      PreparedStatement stmt = null;
      ResultSet rs = null;
      try{
         stmt = DBUtil.prepare(findStatement() + access Controller.
         accessControlStatement());
         stmt.setInt(1, id.intValue());
         rs = stmt.executeQuery();
         rs.next();
         entity = load(rs);
      }catch(Exception ex){
         throw new DataSourceAccessException(ex);
      }finally{
         DBUtil.cleanUp(rs, stmt);
      }
```

```
    return entity;
}

protected AbstractEntity abstractFindByCorpId(String corpId) throws DataSource
AccessException{
    AbstractEntity entity = null;
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try{
        stmt = DBUtil.prepare(findByCorpIdStatement() + access
        Controller.accessControl Statement());
        stmt.setString(1, corpId);
        rs = stmt.executeQuery();
        rs.next();
        entity = load(rs);
    }catch(Exception ex){
        throw new DataSourceAccessException(ex);
    }finally{
        DBUtil.cleanUp(rs, stmt);
    }
    return entity;
}

protected AbstractEntity load(ResultSet rs) throws SQLException{
    Integer id = new Integer(rs.getInt(1));
    if(registry.containsKey(id))
        return (AbstractEntity) registry.get(id);
    AbstractEntity entity = doLoad(id, rs);
    registry.put(id, entity);
    return entity;
}

protected List loadAll(ResultSet rs) throws SQLException{
    List entities = new ArrayList();
    while(rs.next())
        entities.add(load(rs));
    return entities;
}

public List findMany(StatementSource source) throws DataSourceAccess
Exception{
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try{
        stmt = DBUtil.prepare(source.getSql() + accessController.
        accessControlStatement());
        for(int i = 0; i < source.getParameters().length; i++){
            stmt.setObject(i+1, source.getParameters()[i]);
        }
```

```
            rs = stmt.executeQuery();
            return loadAll(rs);
        }catch(Exception ex){
            throw new DataSourceAccessException(ex);
        }finally{
            DBUtil.cleanUp(rs, stmt);
        }
    }

    public int insert(AbstractEntity entity) throws DataSourceAccessException{
        PreparedStatement stmt = null;
        try{
            stmt = DBUtil.prepare(insertStatement());
            stmt.setInt(1, entity.getId());
            doInsert(entity, stmt);
            registry.put(new Integer(entity.getId()), entity);
            return entity.getId();
        }catch(SQLException ex){
            throw new DataSourceAccessException(ex);
        }finally{
            DBUtil.cleanUp(null, stmt);
        }
    }
}
```

Notice that AbstractDataMapper has one collaborator—AccessControlStrategy, which is the object that provides access control logic—whether a client should have access to certain records in the database. This object has a single method—accessControlStatement()—which returns a "where" clause that gets appended to the SQL statement in findStatement(), findByCorpId Statement(), abstractFind(), and findMany() methods. The reason is that not all compounds in the corporate database are accessible by everyone. Some compounds developed through partnership are protected by legal obligations. It is the AccessControlStrategy object's responsibility to make sure these compounds are protected from unauthorized access.

Usually the level of access is controlled by user role, which is managed in a centralized repository (LDAP or a database). Two approaches can be implemented by access control. One is to give AccessControlStrategy the user's identity so that it can look up the user role from the central repository. The other is to provide AccessControlStrategy with the user role directly so that it can simply use it to build the access control logic. Either way, the user identity information should be saved in a session object in the front layer when the user logs into the system and needs to be passed into the AccessControlStrategy object. Using arguments in the method call is tedious in that the argument has to be passed through a chain of method calls before it is actually used. An alternative is to use Thread Local variables. Thread Locals are variables that

are local to their own thread. Somewhere in the thread these variables are created and saved in a registry and only the thread that creates them can access them. The following DataMapperThreadLocalRegistry class defines three thread local variables—a JDBC Connection object, a username and a list of user roles, along with their setters and getters:

```
package com.abcpharma.crs.datamapper;

import java.sql.*;

public class DataMapperThreadLocalRegistry {
   private static final DataMapperThreadLocalRegistry instance = new DataMapper
   ThreadLocalRegistry();

   private DataMapperThreadLocalRegistry() {
   }

   public static DataMapperThreadLocalRegistry getInstance(){
      return instance;
   }


   private static class ThreadLocalConnection extends ThreadLocal {
   }

   private static class ThreadLocalUsername extends ThreadLocal {
   }

   private static class ThreadLocalUserRoles extends ThreadLocal {
   }

   private ThreadLocalConnection connection = new ThreadLocalConnection();
   private ThreadLocalUsername username = new ThreadLocalUsername();
   private ThreadLocalUserRoles userRoles = new ThreadLocalUserRoles();

   public void setConnection(Connection conn) {
      connection.set(conn);
   }

   public Connection getConnection() {
      return (Connection) connection.get();
   }

   public void setUsername(String un) {
      username.set(un);
   }

   public String getUsername() {
      return (String) username.get();
   }
```

```
    public void setUserRoles(String[] ur) {
        userRoles.set(ur);
    }

    public String[] getUserRoles() {
        return (String[]) userRoles.get();
    }
}
```

The way it works is that these variables are set by an application controller and can then be accessed by the AccessControlStrategy object to create the access control "where" clause. If a transaction involves multiple updates to the database, for example, the controller should start and finish the transaction and make sure all updates use the same Connection object to ensure they are either all committed or all rolled back.

The above approach is one way of handling row level access control. If Oracle Fine Grained Access Control is used, the AccessControlStrategy object should set the username to the connection context to allow access control to be handled by Oracle transparently.

The findStatement() and findByCorpIdStatement() are Template Methods. They rely on the subclass to provide their primary behaviors (findStatement Primary() and findByCorpIdStatementPrimary()) while incorporating a common behavior through the AccessControlStrategy object.

> **Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. The Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.**

All entity objects should be able to be queried by their primary keys. The way it works has little difference between the entity objects except in the SQL statement. The method abstractFind() abstracts out common behaviors of find by primary key operation of all entity objects:

```
protected AbstractEntity abstractFind(Integer id) throws DataSourceAccess
Exception{
    AbstractEntity entity = (AbstractEntity) registry.get(id);
    if(entity != null)
        return entity;

    PreparedStatement stmt = null;
    ResultSet rs = null;
    try{
        stmt = DBUtil.prepare(findStatement() + accessController.
        accessControlStatement());
        stmt.setInt(1, id.intValue());
```

```
        rs = stmt.executeQuery();
        rs.next();
        entity = load(rs);
      }catch(Exception ex){
        throw new DataSourceAccessException(ex);
      }finally{
        DBUtil.cleanUp(rs, stmt);
      }
      return entity;
  }
```

The registry object, defined as an instance variable of the AbstractMapper class, caches the entity objects in a map, the primary key of the object as the key and the object as the value. The abstractFind() method performs a lookup in the registry and returns the object being searched if it is already in the cache. Otherwise it performs a database query, calls the load() method to build the entity object, and returns it.

The load() method is as follows:

```
protected AbstractEntity load(ResultSet rs) throws SQLException{
    Integer id = new Integer(rs.getInt(1));
    if(registry.containsKey(id))
        return (AbstractEntity) registry.get(id);
    AbstractEntity entity = doLoad(id, rs);
    registry.put(id, entity);
    return entity;
}
```

Notice two things in the load() method. First, it performs a registry lookup one more time. The reason is that the load() method might be called by other finder methods, for example, findBySubtructure, which do not perform a lookup. Second, the load() method puts the newly queried object into the registry.

Many entity objects have a corp id in addition to its primary key. These identifiers are often used by people to specify a specific entity (in the user community, nobody cares about primary keys). Examples are lab sample identifiers (compound id, sample id, parent id), and employee identifier (worldwide employee id or social security number). For this reason, we introduce a findByCorpId() method. It is very similar to the find() method.

The methods findMany() and loadAll() are used by all finder methods, such as findByParentId() (a parent id may have multiple salt forms) and findBySubstructure() in the CompoundMapper object, that return a list of entities.

The insert() method inserts a new entity object into the database table. It also puts it into the cached registry so that the finder methods can return it from the cache.

A few abstract protected methods are left to the concrete mapper object to implement. They are declared in AbstractMapper so that they can be called by the Template Methods to maximize code reuse.

The CompoundMapper class—the data mapper for the compound object—is as follows:

```
package com.abcpharma.crs.datamapper;

import java.sql.*;
import java.util.*;

import com.abcpharma.crs.entity.*;
import com.abcpharma.crs.molstructure.*;
import com.abcpharma.crs.registrationservice.*;
import com.abcpharma.crs.lsi.*;
import com.abcpharma.db.DBUtil;

public class CompoundMapper extends AbstractPersistenceManager implements Compound
Finder{

   private static String columns = "id, compound_id, molfile";
   private static final String selectStatement = "select " + columns + " from compounds a
   where ";

   private static final String updateStatement = "update compound set ctab = ?";

   private static final String deleteStatement = "delete from compound where id = ?";

   private ChemDBQueryBuilder chemQueryBuilder = null;

   public CompoundMapper(ChemDBQueryBuilder chemQueryBuilder, Access
   ControlStrategy
         accessController){
      super(accessController);
      this.chemQueryBuilder = chemQueryBuilder;
   }

   protected String findStatementPrimary() {
      return selectStatement + " id = ? ";
   }

   protected String findByCorpIdStatementPrimary(){
      return selectStatement + " compound_id = ?";
   }

   public Compound find(Integer id) throws DataSourceAccessException{
      return (Compound) abstractFind(id);
   }
```

```
public Compound find(int id) throws DataSourceAccessException{
   return find(new Integer(id));
}

public Compound findByCompoundId(String compoundId) throws DataSource
AccessException{
   return (Compound) abstractFindByCorpId(compoundId);
}

public List findByParentId(String parentId) throws DataSourceAccessException{
   return findMany(new FindByParentId(parentId));
}

static class FindByParentId implements StatementSource{
   private String parentId;
   public FindByParentId(String parentId){
      this.parentId = parentId;
   }

   public String getSql() {
      return selectStatement + " parent_id = ?";
   }

   public Object[] getParameters() {
      Object[] result = {parentId};
      return result;
   }
}

public List findByComponentGroupId(int componentGroupId) throws DataSource
AccessException{
   return findMany(new FindByComponentGroupId(componentGroupId));
}

class FindByComponentGroupId implements StatementSource{
   private int componentGroupId;

   public FindByComponentGroupId(int componentGroupId){
      this.componentGroupId = componentGroupId;
   }

   public String getSql() {
      return selectStatement + " and component_group_id = ?";
   }

   public Object[] getParameters() {
      Object[] result = {new Integer(componentGroupId)};
      return result;
   }
}
```

```
public List findBySubstructure(String molfile, boolean negate) throws DataSource
AccessException{
   return findMany(new FindBySubstructure(molfile, negate));
}

class FindBySubstructure implements StatementSource{
   private String molfile;
   private boolean negate;

   public FindBySubstructure(String molfile, boolean negate){
      this.molfile = molfile;
      this.negate = negate;
   }

   public String getSql() {
      return selectStatement + chemQueryBuilder. buildSubstructure
      SearchStatement("CTAB", negate);
   }

   public Object[] getParameters() {
      Object[] result = {molfile};
      return result;
   }
}

public List findBySimilarity(String molfile, String simType, String simRange,
booleannegate) throws
      DataSourceAccessException{
   return findMany(new FindBySimilarity(molfile, simType, simRange,
   negate));
}

class FindBySimilarity implements StatementSource{
   private String molfile;
   private String simType;
   private String simRange;
   private boolean negate;

   public FindBySimilarity(String molfile, String simType, String
   simRange,boolean negate){
   this.molfile = molfile;
      this.simType = simType;
      this.simRange = simRange;
      this.negate = negate;
   }

   public String getSql() {
      return selectStatement + chemQueryBuilder. buildMolsim
      Statement("CTAB", simRange,
         negate);
   }
```

```
    public Object[] getParameters() {
        Object[] result = {molfile, simType};
        return result;
    }
}

public List findByFlexmatch(String molfile, String flexParameters, boolean negate)
throws
        DataSourceAccessException{
    return findMany(new FindByFlexmatch(molfile, flexParameters, negate));
}

class FindByFlexmatch implements StatementSource{
    private String molfile;
    private String flexParameters;
    private boolean negate;

    public FindByFlexmatch(String molfile, String flexParameters, boolean
    negate){
        this.molfile = molfile;
        this.flexParameters = flexParameters;
        this.negate = negate;
    }

    public String getSql() {
        return selectStatement + chemQueryBuilder.build Flexmatch
        Statement("CTAB", negate);
    }

    public Object[] getParameters() {
        Object[] result = {molfile, flexParameters};
        return result;
    }
}

public List findByFormulaLike(String formula, boolean negate) throws DataSource
AccessException{
    return findMany(new FindByFormulaLike(formula, negate));
}

class FindByFormulaLike implements StatementSource{
    private String formula;
    private boolean negate;

    public FindByFormulaLike(String formula, boolean negate){
        this.formula = formula;
        this.negate = negate;
    }

    public String getSql() {
```

```
        return selectStatement + chemQueryBuilder.build FormulaLike
        Statement("CTAB",
            negate);
    }

    public Object[] getParameters() {
        Object[] result = {formula};
        return result;
    }
}

protected AbstractEntity doLoad(Integer id, ResultSet rs) throws SQLException{
    String compoundId = rs.getString("compound_id");
    String molfile = rs.getString("molfile");
    Compound compound = new Compound(id.intValue());
    Molstructure structure = new Molstructure(Molstructure.MOLFILE,
    molfile);
    compound.setMolStructure(structure);
    compound.setLsi(LsiFactoryImpl.getInstance().createCompoundID
    (compoundId));
    return compound;
}

protected void doInsert(AbstractEntity entity, PreparedStatement stmt) throws
SQLException{
    Compound compound = (Compound) entity;
    stmt.setObject(2, compound.getMolStructure().getValue());
    stmt.setObject(3, compound.getLsi().getCompoundIDString());
}

protected String insertStatement(){
    return "insert into compound values (?, ?, ?)";
}

public void update(AbstractEntity entity) throws DataSourceAccessException{
    PreparedStatement stmt = null;
    Compound compound = (Compound) entity;
    try{
        stmt = DBUtil.prepare(updateStatement);
        stmt.setObject(1, compound.getMolStructure().getValue());
        stmt.execute();
    }catch(SQLException ex){
        throw new DataSourceAccessException(ex);
    }finally{
        DBUtil.cleanUp(null, stmt);
    }
}

public void delete(AbstractEntity entity) throws DataSourceAccessException{

}
```

```
    public List uniquenessSearch(String structure){
        return new ArrayList();
    }
}
```

The CompoundMapper defines specific data access logic for the Compound object. Its findStatementPrimary() method defines how to query the compound table using its primary key. The findByCorpIdPrimary() defines how to query the compound table using its corporate id and so on.

The CompoundMapper has a key collaborator—ChemDBQueryBuilder. This object builds SQL operators that are specific to a particular vendor's chemistry database. The implementation for the MDLDirect cartridge and the MDL RCG database is as follows:

```
package com.abcpharma.crs.datamapper;

public class MDLDirectQueryBuilder implements ChemDBQueryBuilder{
    private static final MDLDirectQueryBuilder instance = new MDLDirect
    QueryBuilder();

    private MDLDirectQueryBuilder() {
    }

    public static MDLDirectQueryBuilder getInstance(){
        return instance;
    }

    public String buildSubstructureSearchStatement(String columnName, boolean
    negate){
        StringBuffer sql = new StringBuffer();
        negate(sql, negate);
        sql.append(" sss(").append(columnName).append(", ?) = 1");
        return sql.toString();
    }

    public String buildFlexmatchStatement(String columnName, boolean negate){
        StringBuffer sql = new StringBuffer();
        negate(sql, negate);
        sql.append(" flexmatch(").append(columnName).append(", ?, ?) = 1");
        return sql.toString();
    }

    public String buildUniquenessStatement(String columnName, boolean negate){
        StringBuffer sql = new StringBuffer();
        negate(sql, negate);
        sql.append(" (flexmatch(")
            .append(columnName)
            .append(", ?, ?) = 1 AND ")
```

```
        .append("BATCH_JOB_ID").append(" != ? ) OR ")
        .append("(flexmatch(")
        .append(columnName)
        .append(", ?, ?) = 1 AND ")
        .append("BATCH_JOB_ID").append(" = ? )")
        ;
        return sql.toString();
    }

    public String buildMolsimStatement(String columnName, String range, boolean
    negate){
        StringBuffer sql = new StringBuffer();
        negate(sql, negate);
        sql.append(" molsim(").append(columnName).append(", ?, ?) ").append
        (range);
        return sql.toString();
    }

    public String buildFormulaLikeStatement(String columnName, boolean negate){
        StringBuffer sql = new StringBuffer();
        negate(sql, negate);
        sql.append(" fmla_like(").append(columnName).append(", ?) = 1");
        return sql.toString();
    }

    private void negate(StringBuffer sql, boolean negate){
        if(negate)
            sql.append( " NOT ");
    }
}
```

Readers who are familiar with the MDL database should not be surprised to see MDL's structure query operators such as sss, molsim, and flexmatch. The CompoundMapper's StatementSource objects use ChemDBQueryBuilder to append these structure search operators to the structure search SQL statements. Take the substructure search as an example. In CompoundMapper, we have an inner class FindBySubstructure whose getSql() method is as follows:

```
    public String getSql() {
        return selectStatement + chemQueryBuilder.buildSubstructureSearch
        Statement("CTAB", negate);
    }
```

The buildSubstructureSearchStatement looks as follows:

```
    public String buildSubstructureSearchStatement(String columnName, boolean
    negate){
```

```
    StringBuffer sql = new StringBuffer();
    negate(sql, negate);
    sql.append(" sss(").append(columnName).append(", ?) = 1");
    return sql.toString();
}
```

which appends sss("CTAB", ?) = 1 to the SQL statement if MDL implementation is used.

The implementation of the ChemDBQueryBuilder interface is determined at runtime depending on which implementation is passed into the CompoundMapper's constructor. This design makes the CompoundMapper open for extension while remaining closed for changes. Switching to a different vendor database does not require changes in CompoundMapper. All it requires is an implementation of ChemDBQueryBuilder for that vendor, which is passed into CompoundMapper's constructor so the CompoundMapper object knows which one to use at runtime.

For each table in the database, a mapper object is needed. To manage a transaction that involves multiple tables, which is the case for compound registration, use a Unit of Work (Fowler, 2003b) object that groups the inserts and updates together.

The above approach is the basic idea of the Persistence Layer, which decouples the business layer from the data storage layer. Nowadays, commercial and open-source object-relational mapping tools should be considered. They all have a persistence management framework that eases the job of creating one from scratch. Examples are Hibernate, Apache's Relational-Object Bridge, and Oracle's Toplink. They usually define object-relational mapping using an XML file for the persistence manager to do the work at runtime. These tools when used effectively can be quite powerful.

CHAPTER 16

Put Everything Together

In this book, I have demonstrated how to develop chemical information systems using the Java technology and an object-oriented approach, using the chemical registration system as an example. This final chapter summarizes some key points that were discussed in the book.

Technologies have evolved to a point where it is no longer necessary to rely on vendor proprietary technologies such as MDL ISIS to develop chemical information systems (Chapter 1). Enterprise Java and .NET, combined with some chemical information tool kits, are fully capable of developing these systems. The outcome is increased productivity and reduced costs and systems with enterprise strength.

Organizations should use off-the-shelf tools as much as possible (Chapters 1 and 4). Structure drawing packages, chemistry databases and data cartridges, and chemistry rules and property calculations packages are readily available from major chemistry software vendors. They have matured over time, and many provide open APIs. Although there is no end-to-end development framework available in the chemistry information space, we can use industry standard frameworks such as Enterprise Java and .NET to develop chemical information systems. This approach is not special to the chemistry domain because other industries also take the same approach—using open standards combined with domain-specific tool kits. MDL Isentris is aimed at providing such a framework for chemical information systems, but it was not ready for prime time when this book was written. I wonder whether products such as Isentris are really necessary, and whether it is the right approach to develop a life science-specific application server when J2EE and .NET application servers are available that are backed by other industries. In my view, J2EE or .NET combined with life science-specific tool kits can offer the best-of-breed solutions with the lowest total cost of ownership possible.

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

Whether to develop in-house or outsource the project, the development team must have strong technical skills combined with strong business knowledge and must interact with users on a regular basis (Chapters 1 and 5). Try to embed a user in the development team if possible. The idea of handing off a requirement document to an outsourcer and expecting a product in 6 months or 1 year that meets user expectations and gets delivered to the users on time simply does not work.

Again regardless of in-house or outsource development, document a list of features and prioritize them. Break the project down into short, timeboxed iterations, each focusing on one or two of these features (Chapter 5). Do not let the iteration deadline slip. Reduce the scope of the iteration if necessary. Implement features with high business values and high business and technical risks in early iterations. Make sure each iteration delivers a production quality partial system to solicit feedback and let the system grow incrementally. The project plan should be adjusted based on the feedback. It is OK if the initial project plan is not accurate. However, it should become more and more accurate as more iterations are completed. Test and integrate early and frequently.

Design the architecture in early iterations (Chapters 7 and 8). Both user requirements (functional and nonfunctional) and corporate IT standards drive architecture decisions. However, user requirements always take precedence.

Develop use case specification documents to capture detailed functional requirements (Chapter 9). Use System Sequence Diagrams and Activity Diagrams as complements. Use case specifications should be developed, communicated, and reviewed at the beginning of each iteration.

Develop a domain object model based on use case specifications (Chapter 10). This is the bridge between the requirements and the design.

Evolve the domain object model into a design object model by applying object design principles (Low–High, Open–Closed) and patterns (Chapters 2, 3, 12, 14, and 15). Introduce new objects and layers of indirections to reduce coupling and increase cohesion. Use the Design by Interface technique to keep the dependencies at abstraction levels rather than at implementation levels. Insulate vendor specifics using vendor-neutral APIs. The goal is to be able to swap out current vendor tools and replace them with others without affecting the rest of the system.

As software professionals, we are not developing software systems for the sake of developing software systems. Our goal is to deliver software systems that provide the highest business values to our users (e.g., helping them to increase drug discovery productivity and providing them with better decision support). When developing these software systems, always keep in mind that business evolves and so should the software systems under development. The system must adapt to changes, or otherwise it will not sustain and will soon

become obsolete. Everything we can do to facilitate the evolution is worth the investment, and hence, the design principles and patterns are valuable. We cannot do much to the fact that we are always working within the resource and deadline constraints. What we can do, though, is work with business to prioritize tasks and deliver the highest business values quickly and all the time. Sticking to a plan that is outdated and no longer has business value is a waste of time and money. This is what iterative and incremental development methodologies are all about.

BIBLIOGRAPHY

Agarwal, P. 2004. *Struts Best Practices: Build the Best Performing Large Applications.* http://www.javaworld.com/javaworld/jw-09-2004/jw-0913-struts.html.

Ambler, S. 2005. *The Elements of UML(TM) 2.0 Style*. Cambridge University Press.

Alur, D., Crupi, J., and Malks, D. 2003. *Core J2EE Design Patterns: Best Practices and Design Strategies,* Second Edition. Prentice Hall.

Beck, K. 2004. *Extreme Programming Explained: Embrace Change,* Second Edition. Addison-Wesley Professional.

Booch, G. 1991. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings.

Booch, G. 1994. *Object-Oriented Analysis and Design with Applications,* Second Edition. Benjamin/Cummings.

Booch, G., Rumbaugh, J., and Jacobson, I. 1999. *The Unified Modeling Language User Guide.* Addison-Wesley Professional.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley.

Cockburn, A. 2001. *Writing Effective Use Cases.* Addison-Wesley.

Fowler, M. 1997. *Analysis Patterns: Reusable Object Models*. Addison-Wesley.

Fowler, M. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

Fowler, M. 2003a. *Patterns of Enterprise Application Architecture.* Addison-Wesley.

Fowler, M. 2003b. *UML Distilled,* Third Edition. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Jacobson, I., et al. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley.

Kruchten, P. 2003. *The Rational Unified Process: An Introduction,* Third Edition. Addison-Wesley Professional.

Larman, C. 2004. *Agile and Iterative Development: A Manager's Guide.* Addison-Wesley.

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li

Larman, C. 2005. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development,* Third Edition. Prentice Hall.

Martin, R. 2003. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall.

Palmer, S. R., and Felsing, J. M. 2002. *A Practical Guide to Feature-Driven Development (The Coad Series).* Prentice Hall.

Rumbough, J., et al.1990. *Object-Oriented Modeling and Design.* Prentice Hall.

Schwaber, K., and Beedle, M. 2001. *Agile Software Development with SCRUM,* First Edition. Prentice Hall.

INDEX

Developing Chemical Information Systems: An Object-Oriented Approach Using Enterprise Java, by Fan Li