CSC 478: Software Engineering Capstone

Module 5

Design and Implementation

Objectives:

1. Understand what object-oriented design is and how it differs from procedural design.
2. Be aware of the existence of several object-oriented design techniques.
3. Know what a design pattern is, and be familiar with at least 2 examples of design patterns.
4. Know what a use case is.
5. Understand why UML was developed, and be introduced to some of the diagrams UML uses.
6. Understand some of the fundamental principles of user interface design.
7. Know why prototyping is especially useful in user interface construction.
8. Understand the importance of using the appropriate means to present information to users.
9. Be aware of some of the common pitfalls encountered in user interface design.
10. Understand that software engineering is not the same thing as writing code.

Reading:

1. Sommerville, chapter7 & Web Chapter 29 (Interaction Design, available at: https://iansommerville.com/software-engineering-book/static/web/).

Assignment:

Participate in the group discussion for Module 5.

Quiz:

Take the quiz for Module 5.

5.1 The Design Phase

The design phase is where it is determined how the software requirements are to be satisfied. Design includes system modeling and application architecture, but goes much further. For example, the requirements may state that the software should keep track of all employees at an organization. System modeling will determine the boundaries of how users should be able to access employee information, and may determine that all employee information should be stored in a database located on a central server, with nightly backups to an on-site as well as an off-site location. It may be decided that the architecture for the software will be transactional in nature. In the design phase, it will be determined what flavor of database will be used (Oracle, SQL Server, MySQL, etc.), the data types that will be used to store the various pieces of employee information, how the required security measures will be implemented, and so on. As design progresses, requirements that were missed during requirements elicitation may be discovered, questions may arise concerning the feasibility of certain requirements, and other issues may arise, all of which will prompt temporary suspension of at least a portion of the project while these issues are sorted out with the customer. This possibility of change illustrates the major pitfall of the waterfall process model, and supports the presence of the customer during development as prescribed by many of the agile process models.

A complete history of software design is beyond the scope of this course. However, as software engineering has evolved, the notions of structured design and modular programming have proven themselves essential. In this module, we'll talk about several of the most commonly encountered design concepts.

5.2 Information Hiding

The principle of information hiding was proposed by David Parnas in the early 1970's [1]. The essential argument is that a module, A, that invokes a method of a separate module, B, should not need to know the implementation details of the method, or of Module B. Rather, it should only need to know what inputs to provide and what type of output is produced. Unrestricted use of global variables is usually a good example of bad information hiding. The public visibility of global variables means they can be modified from anywhere in a program, which may be appropriate for some variables, but not for others. If the value of a variable is supposed to be determined by following some algorithm, modifying the variable directly short circuits that algorithm, and could lead to an unstable system state.

Information hiding is often erroneously confused with encapsulation. Encapsulation entails grouping together a set of related components (e.g., attributes, methods/functions, etc.) into a unit. An object-oriented class is a good example of encapsulation. The confusion with information hiding arises because many of the components that are encapsulated are hidden from public scope. The hidden components are representative of information hiding. However, it is also possible to make all the components publicly visible, or global, in which case the components are still encapsulated, but very little, if any, information hiding exists.

5.3 Module Coupling and Module Cohesion

Following the same line of thinking as information hiding are two properties every module has: **coupling** and **cohesion**. Coupling refers to the degree to which one module interacts with another module. There are six levels of coupling, described next, in order of increasing coupling:

1. **uncoupled** — the modules do not interact with each other at all
2. **data coupling** — only individual data units are passed (e.g., simple arguments) from one module to another
3. **stamp coupling** — an entire data structure, such as a list, is passed from one module to another
4. **control coupling** — a "control flag" is passed to allow one module to control the behavior of another module
5. **common coupling** — the modules access common or global data
6. **content coupling** — a module directly modifies another module's data or execution branches from one module directly into another module

In general, you should always try to keep the amount of coupling between modules to a minimum. Don't pass an entire data structure if only one specific item in that data structure is needed. Passing an entire list to a function, for example, allows the function to access all the elements contained in the list, which may not be desired. Excessive use of global data eliminates the ability of an application to regulate access to data, thereby violating the principle of information hiding. Never allow execution to pass from one module directly into another module (e.g., with a GOTO statement). With modern high-level programming languages, content coupling has pretty much been wiped out.

Unless you are writing a very trivial program, you cannot have zero coupling; some amount of coupling is necessary in order to have a functional program. What types of coupling you allow in an application will depend on context, of course. There is nothing wrong with using global constants to store values that must be widely accessible to many parts of an application, but there should be strong justification for using constants as opposed to attributes that are accessed via getter and setter functions.

Cohesion refers to the degree to which a module performs one, and only one, function. Modules that perform multiple functions can be confusing to understand, and can cause unforeseen problems. There are seven levels of cohesion, described below in order of increasing cohesion.

1. **coincidental** — the various functions of a module are not related; for example, consider a class containing methods that have nothing to do with each other, or with the purpose of the class
2. **logical** — logically related functions are placed in the same module, although they can have different purposes; for example, an input reader class having functions for reading data from several different types of devices - each function reads data, but the way they read the data will be different
3. **temporal** — multiple functions are related only through timing; e.g., the functions may be responsible for initializing system attributes on system startup
4. **procedural** — different functions related only because they must be performed in a certain sequence to accomplish a goal; e.g., functions that read data, validate data, process data, write data - these activities must be done in a certain order
5. **communicational** — an example of this is when unrelated data are read in together to avoid excessive disk accesses

6. **sequential** — output from one function of a module is needed as input by another function of the module

7. **functional** — every function of a module is essential to accomplish exactly one goal, and all essential functions are in the same module; this is the ideal situation

You should always try to achieve maximum cohesion. Doing so minimizes the responsibilities of your modules, and helps to localize faults, which makes debugging easier. From a purist standpoint, it also makes for a more elegant solution.

In summary, the goal is to maximize module cohesion while minimizing module coupling, but it's a balancing act. Sometimes it's necessary to trade off on one in order to optimize the other.

5.4 Object-Oriented Design

Object-oriented design (OOD) focuses on the representation of components as discrete, cohesive, units, commonly called classes. OOD also provides for functional descriptions of the relatedness of similar entities through abstraction and inheritance. OOD is far too broad a field to cover in any significant depth in a software engineering course, so we only have time to barely scratch the tip of the iceberg, so to speak. This is unfortunate, but if there is any specific topic you would like to see covered in more depth, let me know, and maybe I can post some additional material, or we can open up a discussion forum to talk about it.

Numerous object-oriented design techniques have been proposed over the years. Each technique had its pros and cons. Some could be used only with certain object-oriented languages. The list below will give you an idea of some of the techniques that have been developed and used throughout the last 20 years or so. A brief history of object-oriented design can be found at http://uml.tutorials.trireme.com/ [2].

1. Booch (by Grady Booch)
2. Object Modeling Technique (OMT)
3. Object-Oriented Software Engineering (OOSE)
4. CRC (Class-Responsibility-Collaborations)
5. Business Object Notation (BON)
6. Martin-Odell
7. Coad-Yourdon
8. MOSES
9. Object-oriented Systems Analysis (OSA)
10. Syntropy
11. Semantic Object Modeling Approach (SOMA)
12. Shlaer-Mellor

In 1994 there were reportedly at least 72 distinct object-oriented design techniques [2]. There were numerous overlaps between the various methods, and none of the approaches could really be used universally. Each technique had its own loyal faction of supporters. In an effort to standardize the field of object-oriented design, the creators of the Booch, OMT, and OOSE techniques got together and decided to build a single technique that combined the best of all the existing methods. What they came up with is the first version of the Unified Modeling Language, commonly known as UML. Those of you who are interested in finding out more should visit the official site of the Object Management Group (OMG), the official site of the creators of UML [3].

One of the primary goals of UML was to provide an object-oriented design technique that was independent of any programming language. A second primary goal was to make the technique extensible, meaning that users could change it by adding their own constructs as needed. UML has become very popular in certain sectors, although it has not yet received universal acceptance. This could be due largely to the fact that UML is quite large and complex, and there is a pretty steep learning curve involved in mastering it. You must thoroughly understand the UML specification in order to use it to its fullest potential. You also have to master at least one object-oriented language into which to translate your design. Finally, you need to have, and know how to use, one or more CASE tools that support UML, since it relies very heavily on diagrams.

UML uses a class diagram to represent the static architecture of the system. Sommerville uses several examples of class diagrams in the text, and you've probably had prior experience with them in your job or one of your other classes. The static architecture can be supplemented by using interaction diagrams, which are based on use cases. A **use case** is a view of the system as seen from the perspective of a particular user, which can be a human or some other software application that interacts with the system. Different users will use the system in different ways. Together, all the use cases for a system should cover all the functionality of the system's requirements.

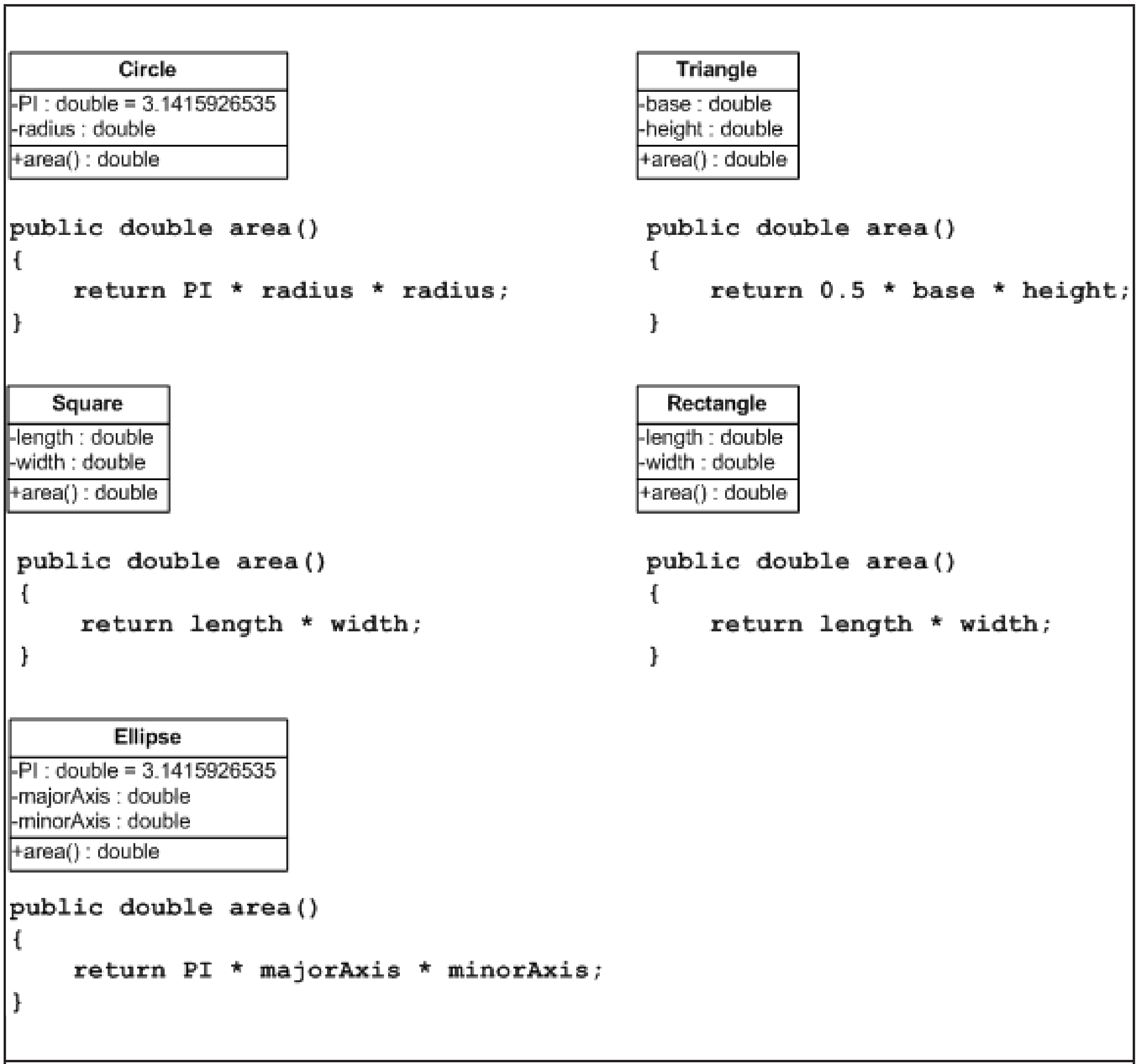
UML state diagrams can be used analogously to state-transition diagrams to show how the system goes from state to state. For a more detailed look, an interaction diagram can be used to show precisely how certain objects will interact to accomplish a particular function. In total, there are over a dozen or so diagrams UML supports. It's not always necessary to use every diagram for every project, although in my personal opinion a class diagram should be mandatory for the design of all object-oriented systems.

Unfortunately, we don't have time to cover UML extensively in this course, but I will mention a couple of things about CASE tools that support UML. Some, like Rational Architect, are exorbitantly expensive if you want to use them for commercial purposes, but others can be downloaded and used for free. For Java, two decent choices I have seen are Poseidon for UML [4] and ArgoUML [5]. Microsoft Visio supports the creation of class diagrams, but not surprisingly it is limited to Microsoft-centric languages for its default data types. Some tools will take a class diagram and generate skeleton code automatically. Skeleton code consists of the parts of a class' source code such as the class header, method headers, attribute declarations, and the getter and setter (often called accessor and mutator) methods. Basically, skeleton code refers to any code that can be generated automatically from a design document.

5.5 Abstraction

The concept of **abstraction** is central to object-oriented design. Abstraction simply means you take a design and extract those elements that are common to other related designs. As a simple contrived example, suppose we are writing an object-oriented toolkit for analyzing geometrical shapes. We first come up with several classes to model various two-dimensional shapes: circle, triangle, square, rectangle, ellipse, et al. Suppose one requirement we must meet states that we need to provide a way to compute the areas of these shapes, so we add a method called `area()` to each class. To aid in designing the toolkit we come up with the UML class diagram shown in Figure 5.1.

Looking at this class diagram, it's easy to spot the commonalities. First of all, every shape has an `area()` method; only the implementation differs from shape to shape. Second, it can be seen that a square is simply a special case of a rectangle, and a circle is just a special case of an ellipse. If we extract the commonalities, we might end up with a class diagram like the one shown in Figure 5.2. Here, an abstract class, `Shape2D`, has been created and all other shapes are derived from this class. The `area()` method, common to all shapes, has been pulled out of the individual shape classes and placed into the `Shape2D` class. Using polymorphism, each subclass can inherit the method and override it appropriately. For the shapes that are special cases, `Circle` and `Ellipse`, I've shown them in the diagram, but there are arguments that can be made for and against including them. On one hand, they don't contain any additional functionality, so they are redundant in that respect. On the other hand, it may be desirable to allow users of the toolkit to be able to declare variables of type `Circle` or `Square`, rather than using `Ellipse` or `Rectangle`.



```
public double area()
{
    return PI * radius * radius;
}
```

```
public double area()
{
    return 0.5 * base * height;
}
```

```
public double area()
{
    return length * width;
}
```

```
public double area()
{
    return length * width;
}
```

```
public double area()
{
    return PI * majorAxis * minorAxis;
}
```

Figure 5.1. Initial class diagram for geometry toolkit.

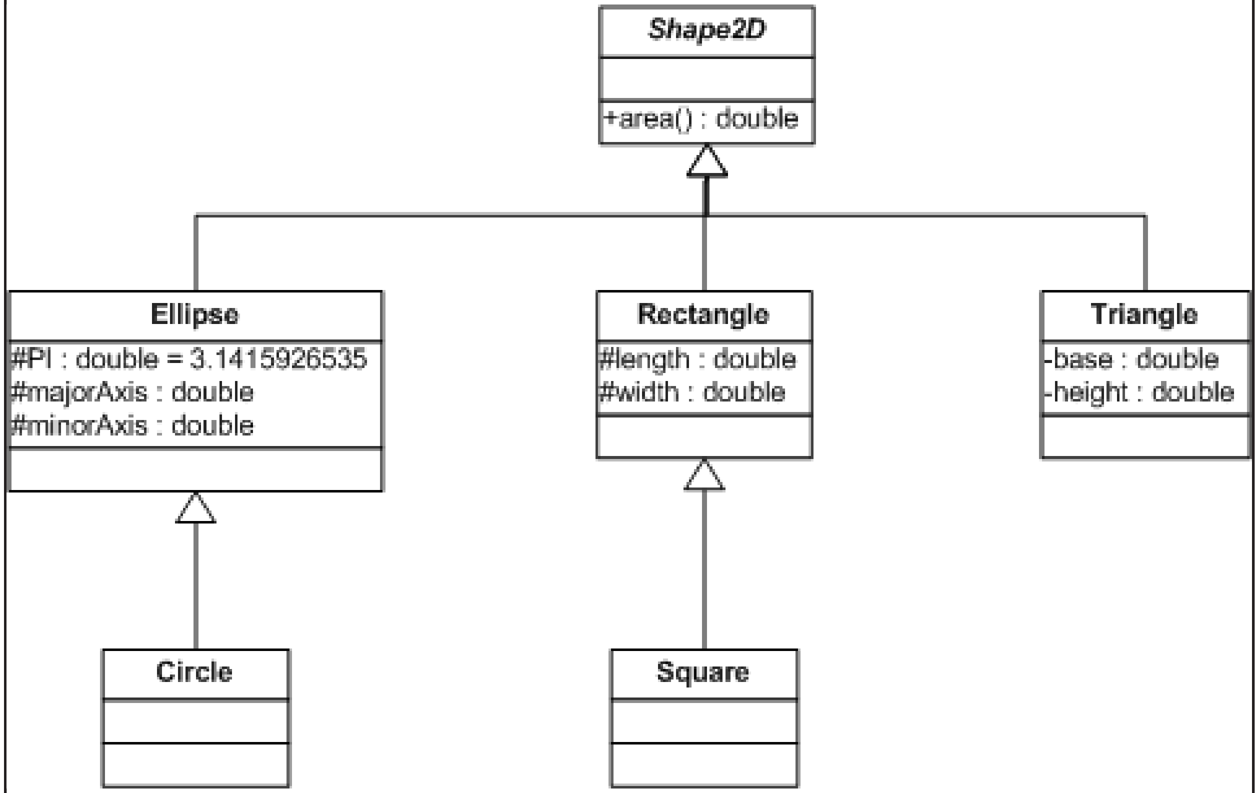


Figure 5.2. Class diagram for geometry toolkit, after commonalities have been extracted.

5.6 Pros and Cons of Object-Oriented Design

Object-oriented design has been touted by many in the field as the new paradigm for software development. Of course, not everyone agrees. Advantages of object-oriented design include:

- It more closely approximates the way people think about real world problems.
- It promotes good programming techniques such as encapsulation, modularization, and information hiding.
- It can help to localize faults during debugging.
- It promotes the development and use of repositories of reusable components.

Disadvantages include:

- Designing a solution based on the way people tend to think about the problem may not be the best design strategy.
- Developers accustomed to designing in a procedural manner sometimes have trouble switching to an object-oriented mindset.
- Complex systems can be difficult to trace since execution jumps from one source file to another (i.e., consecutively executed statements will not always appear in consecutive order)

- Significant overhead can be incurred during runtime, since more indirect addressing frequently occurs than in procedural systems.

5.7 Design Patterns

Software design is a vast topic — far too large to cover it all in this course, or even a course dedicated solely to design. We'll begin our discussion of design with a topic that is common to most types of software systems: design patterns. Credit for the idea of design patterns is usually ascribed to Christopher Alexander, who in his book, *A Pattern Language*, stated:

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."* [6]

Alexander was a building architect, but his idea of design patterns spans virtually every domain, thus it is no surprise that design patterns would become central to the design of software. In software, there are certain problems, architectural elements, and algorithms that recur frequently. Very often a solution found to a certain type of problem can be reused for other, similar problems. The solution to one problem can therefore be used as a pattern from which to build solutions to similar problems. Although design patterns are almost always considered strictly object-oriented, some design patterns are not restricted to the object-oriented approach. Design patterns have been grouped into three broad categories:

1. **Creational patterns** — concerned with how objects and components are instantiated

2. **Structural patterns** — concerned with how objects and components interact with each other, and how they can be assembled into larger structures

3. **Behavioral patterns** — concerned with algorithms, flow of execution, and delegation of responsibilities

We'll look at some of the patterns in each category in more detail in the following sections.

Creational Patterns

Singleton

The **Singleton** pattern is used when you know there should only be a single instance of a particular class, and you need to have global access to it. A good example is the `Runtime` class in Java. Every java program uses a single instance of `Runtime` to interface with its environment. This makes sense, since if there were multiple `Runtime` instances allowed it might be possible for two different instances to issue contradictory commands, which could place the program in a deadlocked state. A singleton does not have to be a class, it can also be a static method or a static attribute. Constants are an example of singletons at the attribute level. Some programming languages, like Java, allow for declarations of singletons through a reserved word, like `static`. Not all programming languages have this support, so specifying singletons in those languages require care on the part of the programmer to ensure that only a single instance can be created.

Factory Method

A **Factory Method** allows us to define an abstract interface for creating an object, but lets subclasses decide which concrete class to instantiate. An example of this can be found in Java's `List` interface. Java supports two types of lists: 1) `ArrayList`, which is an array-based list, and 2) `LinkedList`, which is a list created from sequentially linked nodes. Both types of lists need to provide the capability to iterate over their elements, but the mechanism for iterating over an `ArrayList` is different from the mechanism for iterating over a `LinkedList`. The abstract `List` interface specifies the capability for iteration with its `listIterator()` and `listIterator(int index)` factory methods. These methods must be overridden in the `ArrayList` and `LinkedList` classes to create the appropriate iterator for the list. Because the factory methods are implemented within the `ArrayList` and `LinkedList` classes, implementation-specific details of the list can be accessed in the implemented methods, but not in the abstract `List` interface. One potential drawback of this pattern is that a factory method can effectively hide a concrete class from the public scope. While this is good from the standpoint of information hiding, it can sometimes create problems, for example when the provided factory methods do not provide the needed functionality.

Abstract Factory

The Abstract Factory pattern is used to create an interface for creating families of related or dependent objects without specifying concrete details about those objects. This pattern is used heavily in the creation of different types of GUI components. Figure 5.3 shows an example of how this pattern is used in Java to create different types of borders for GUI components. All borders are created by calling special factory methods of the `BorderFactory` class. There are more than the three types of borders shown in the figure. The point here, though, is that a single class instantiates and returns a concrete border type. The `BorderFactory` class ensures that a specific type of border is created, without corrupting those features common to all border types; i.e., consistency is preserved. Another advantage of using the factory is that it makes it possible to swap one border type for another easily, simply by calling a different factory method. The downside to using a factory is that if new family members need to be added, the factory class must be updated to give it the appropriate factory methods. Depending on the properties of the new family member, this update may be relatively trivial, or it may be extremely complex and require modifications beyond just the factory class.

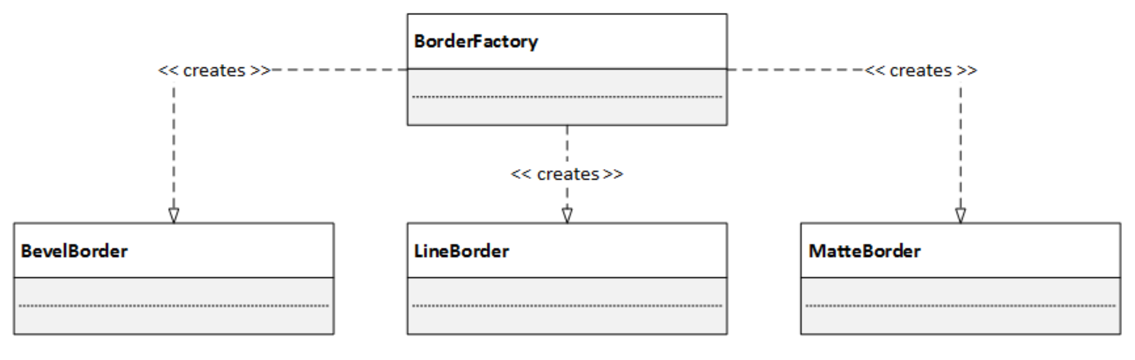


Figure 5.3. Example of the Abstract Factory pattern.

Builder

The Builder pattern separates the creation of an object from its representation. This is helpful when an object's representation can be very complex. Consider the `Calendar` class in Java. A `Calendar` has a considerable number of attributes, including month, day, year, hour of day, minute, second, locale, etc. Typically a class includes a separate constructor to handle each possible combination of initialization values for the class' attributes. However, as the number of attributes grows, the number of combinations of those attributes in the constructors quickly becomes too large to handle. A builder class, in this case an inner class of `Calendar`, called `Builder`, provides separate methods for specifying which values to set. Now, rather than having to provide a large number of different constructors, one simply needs to invoke the set of builder methods needed to set the desired attributes. Figure 5.4 shows a partial example of the `Calendar` class, along with its `Builder` class. For the 6 attributes of `Calendar` shown, if we were to provide constructors for all combinations of inputs for those attributes; i.e., all pairwise combinations, all 3-way combinations, all 4-way combinations, etc., we would need 63 different constructors. That's a lot of code, and a lot of potential for error. Using the Builder class we need only invoke a single, default constructor, then invoke only those builder methods we need to set the initial values we want to set.

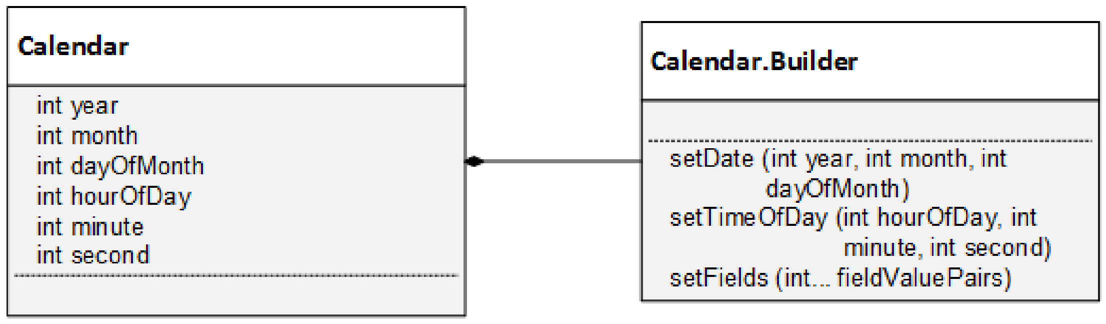

Calendar
int year
int month
int dayOfMonth
int hourOfDay
int minute
int second
Calendar.Builder
setDate (int year, int month, int dayOfMonth)
setTimeOfDay (int hourOfDay, int minute, int second)
setFields (int... fieldValuePairs)

Figure 5.4. Example of the Builder pattern, using Java's `Calendar` class.

Prototype

There are times when you may need to create instances of many different types of objects that are somehow related to each other. A good example of this is a game that has many different types of creatures that can roam around the game environment. One solution to this problem would be to use factory methods or an Abstract Factory to instantiate the various creature types. The drawback here is that you end up with a set of factory methods that parallels the set of possible creature types. This is shown in Figure 5.5. The problem with this is that if the number of creatures becomes very large, you now have a factory class that contains a separate method for each creature type. If the game is extended to add more creature types, the factory class must also be modified to add the appropriate factory methods.

An alternative is to give each creature class the capability of creating new instances of itself, and dispensing with the Abstract Factory and factory methods. In other words, each creature can serve as a prototype for creating other creatures of the same type. This can be accomplished by giving each creature class a `clone()` method that returns a new, default-state instance of itself. This is shown in Figure 5.6. Using this Prototype pattern, the factory class and its factory methods can be eliminated, thus allowing extensibility while avoiding the hassle of updating this class every time a new creature type is added (or one is removed).

One drawback to using the Prototype pattern is that it may be difficult to implement the clone() method, particularly if the project is already well underway. Circular references can also be an issue, but can be avoided if care is taken when designing the classes. Another potential problem is that the `clone()` method is ideally suited to returning default-state instances of a class. If specific parameters need to be set for the clone, this can be difficult, since specifying parameters for the `clone()` method may violate the principle of information hiding, and require clients to know about private details of a class in order to invoke the `clone()` method.

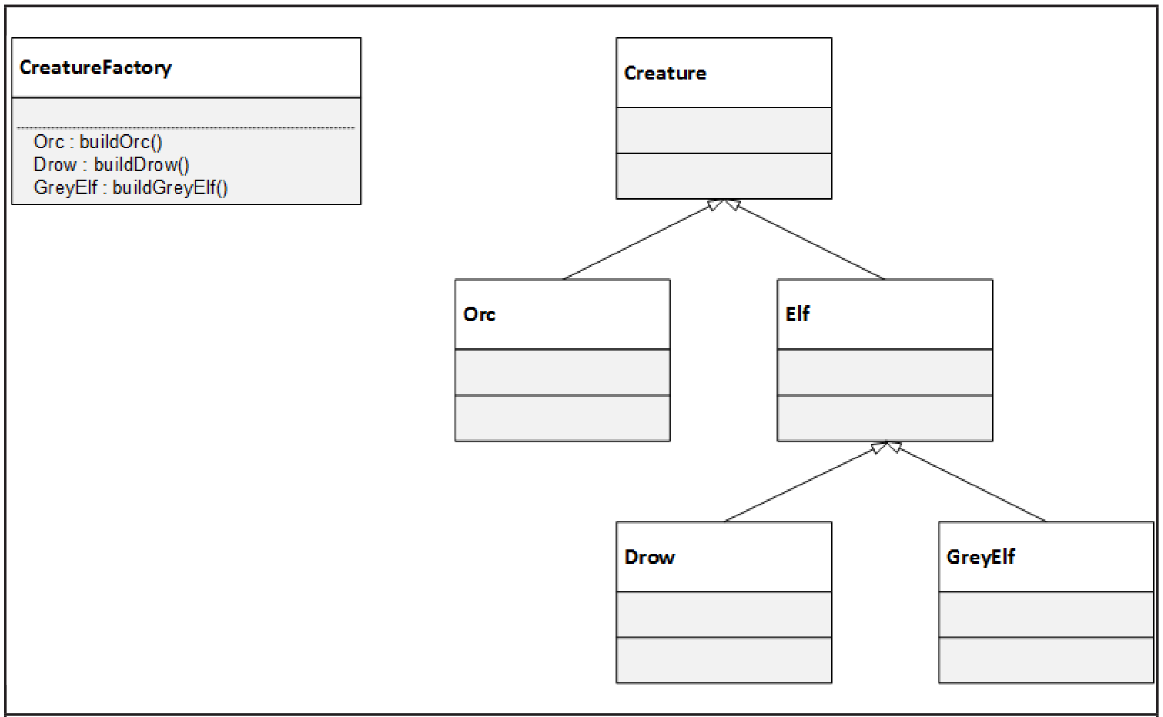


Figure 5.5. Creating different types of creatures using an Abstract Factory and factory methods. Note the set of factory methods parallels the hierarchy of creature types.

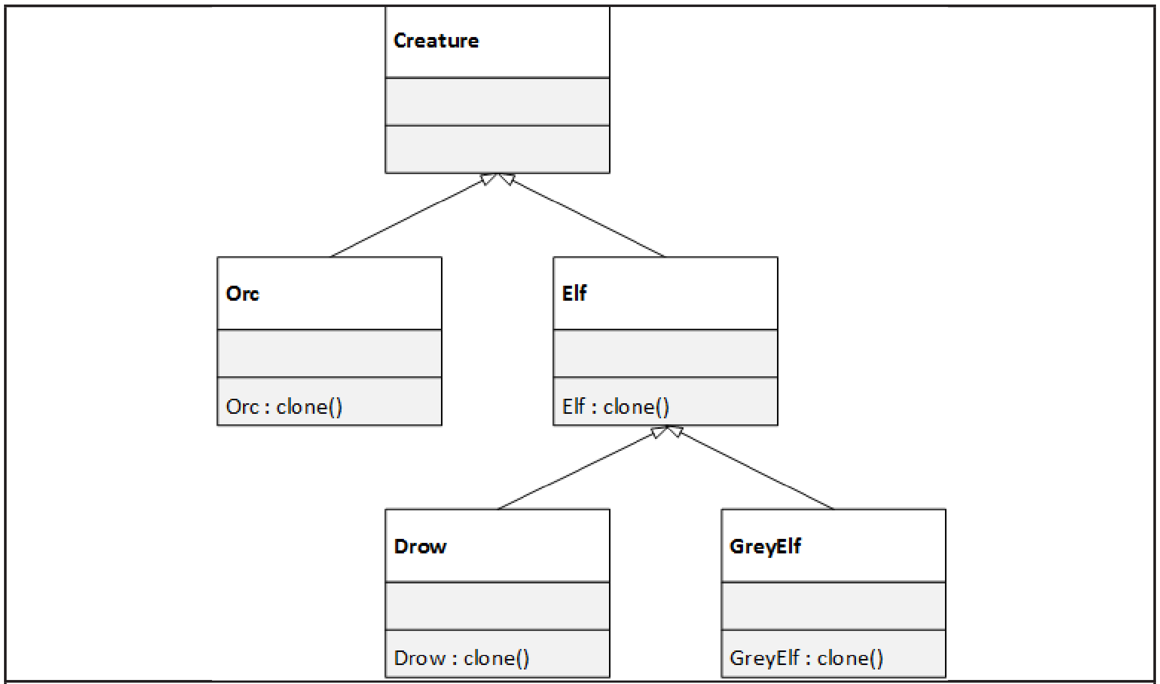


Figure 5.6. Creating different types of creatures using the Prototype pattern.

Structural Patterns

Adapter

The Adapter pattern serves to make two interfaces compatible that would not otherwise be compatible. Use of this pattern typically arises when attempting to use existing components that are now needed to interact in ways that were not foreseen when the components were built. The Adapter pattern is also known as the Wrapper pattern. The Java programming language uses adapter classes extensively in its AWT and Swing components. Some examples are `MouseAdapter`, `ContainerAdapter`, and `KeyAdapter`. These classes contain empty methods that can be conveniently implemented to provide specific event handling for these components. Since event handling is specific to particular applications, it is impossible to write common code to handle events in different ways, so using adapters allows each application to specify its own behavior in response to events. One common problem with using adapters is ensuring that the principle of information hiding is maintained. Adapters should be general enough that they don't rely on detailed knowledge of application components.

Bridge

When an abstraction can have multiple possible implementations, the usual way to deal with this is to use inheritance. Inheritance works, but can also limit flexibility, and it can lead to too much dependency of a class on its superclasses. An example where inheritance can be problematic is in the creation of GUI components that must be supported on multiple platforms. An abstract Window class, for example, can be subclassed to support different platforms. The inheritance works because each subclass can implement the abstraction according to the needs of the specific platform it is designed to support. For example, a Win_MSWindows subclass will render a Window on Microsoft Windows; a WinLinux subclass will render the same Window on Linux; a Win_MacOSX subclass will render the same window on Mac OS X; and so on. The problem with this approach is that a separate subclass must be created for every platform that is to be supported. This in turn makes the entire codebase platform dependent; i.e., each platform must have a completely different set of classes for use on that specific platform. This can become very complex if there are multiple subclasses of windows that can be displayed. Furthermore, if the abstraction of a Window is changed, all the subclasses and their dependencies must also be changed.

The Bridge pattern seeks to avoid this problem by decoupling the implementation of an object from its abstraction. Using this pattern, a second inheritance hierarchy is created to model the implementation behavior. This permits the abstraction to have its own hierarchy to model different subclasses of Window. The abstraction maintains a reference to its implementation. This allows any number of different implementations to be created, and that can be easily plugged into the Window abstraction just by setting the value for the implementation reference. Figure 5.7 shows an example of the Bridge pattern could be used in this example.

Composite

Some objects, like graphics, can be extremely complex. A container graphic, for example, might contain multiple containers within it, along with any number of other graphics, some of which may be primitive, like lines, and some of which may be more complex, like buttons or list boxes. Each container within the larger container can also have multiple graphics, including more containers, and so on. This design is recursive in nature, where any graphic that can contain other graphics represents the recursive case, and primitive graphics that cannot contain other graphics represent the base case. What makes this situation even more complex is the need to be able to treat both individual graphics and compositions of graphics identically for certain purposes. The

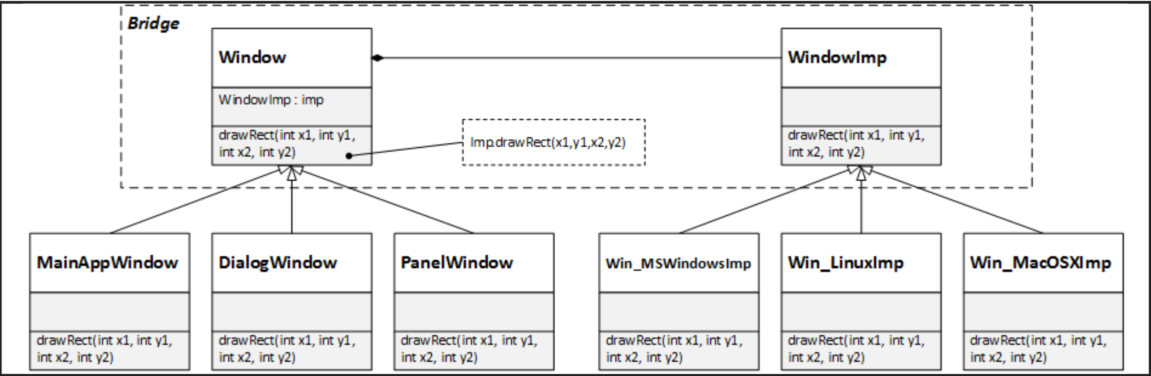

Bridge
Window
WindowImp : imp
drawRect(int x1, int y1, int x2, int y2)
Imp.drawRect(x1,y1,x2,y2)
WindowImp
drawRect(int x1, int y1, int x2, int y2)
MainAppWindow
drawRect(int x1, int y1, int x2, int y2)
DialogWindow
drawRect(int x1, int y1, int x2, int y2)
PanelWindow
drawRect(int x1, int y1, int x2, int y2)
Win_MSWindowsImp
drawRect(int x1, int y1, int x2, int y2)
Win_LinuxImp
drawRect(int x1, int y1, int x2, int y2)
Win_MacOSXImp
drawRect(int x1, int y1, int x2, int y2)

Figure 5.7. Using the Bridge pattern to separate the implementation of a Window from its abstraction.

typical approach of using inheritance to create container classes, and primitive graphic classes causes problems, because client code must be able to distinguish between primitive graphics and containers.

The Composite pattern attempts to remedy these problems by creating an abstract class that represents both containers and primitives. In this example, an abstract `Graphic` class can be created that declares operations such as drawing lines and rectangles that are common to all graphics, plus operations that containers use for managing the child graphics they contain. Primitive graphics do not implement container-specific operations. Figure 5.8 shows the basic structure of the Composite pattern. In our graphics example, `Leaf` represents a primitive graphic, while Composite represents a container graphic. The “has-a” relationship recursively directs back to the Component class to indicate that a container can contain instances of Leaf (primitives), or other Composites (containers). Graphic operations in Composites are performed on all child graphics contained in the Composite.

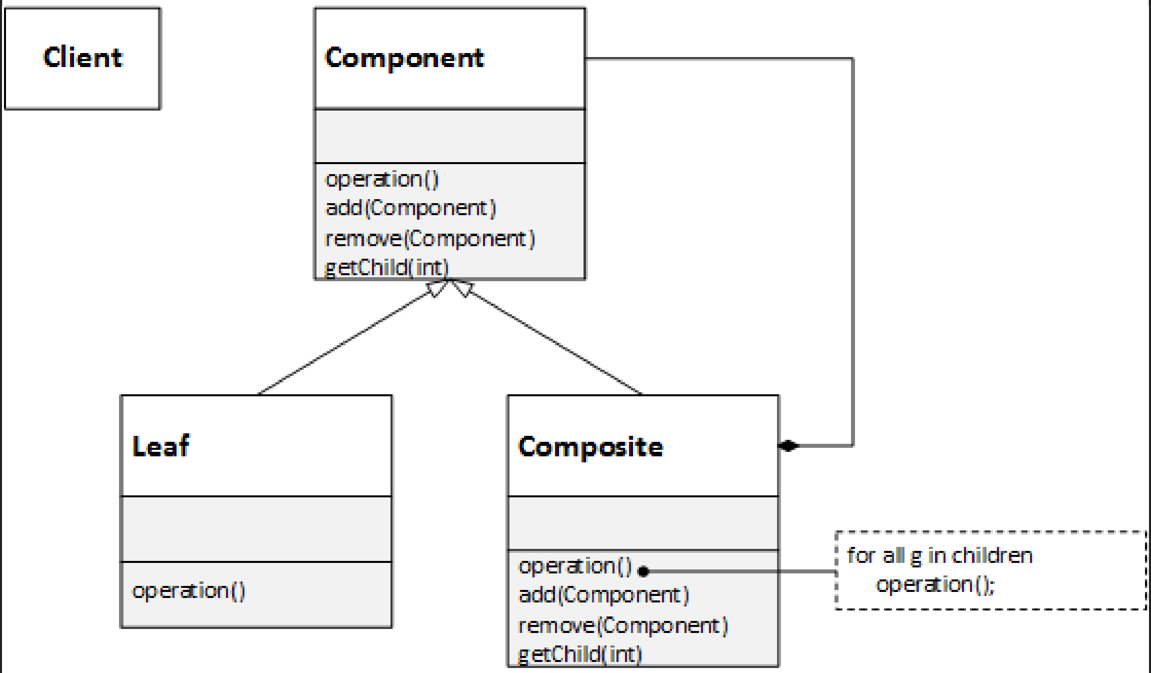


Figure 5.8. The basic structure of the Composite pattern.

Decorator

Adding functionality to a class is often done by creating a subclass that contains the additional functionality. The Decorator pattern provides an alternative to subclassing, allowing you to add functionality or responsibilities to an object dynamically, rather than to an entire class. A good example of where this would be applicable is adding elements like borders or scrollbars to a window in a GUI. Composition is frequently used to achieve this goal. For example, in Java, certain graphical components can be made scrollable by placing them inside a `JScrollPane`. The `JScrollPane` has the scrollbars and a reference to the component being scrolled, so it can obtain information it needs about the component to ensure scrolling is smooth. The elements that are added to a component are analogous to decorations, and the enclosing container serves to add the decorations, hence the pattern name, Decorator.

Not only does using the Decorator pattern increase flexibility over static inheritance, it also avoids the problem of overloading classes with functionality that may or may not get used. If the functionality does not get used, it takes up space, and could be a potential source for error at some point in the inheritance hierarchy. One potential problem with using Decorator, as it applies to our example, is that the Decorator and the object being decorated should both be derived from the same common superclass to ensure consistency and conformance throughout the interface. This superclass also needs to be as lightweight a component as possible, since it may be used repeatedly, and recursively as we saw with the Composite pattern.

Facade

The Facade pattern can be used to accomplish two goals:

1. It can be used to provide a unified interface to a set of subsystem interfaces.

2. It can be used to provide a single, high-level interface to a set of low-level sub-interfaces.

In the first case, suppose you have a system like an office application suite, where you have a word processor, a spreadsheet, a slide maker, and other such applications, where it is common to use content created in one application in one of the other applications. Users who use more than one application in the suite will be confused if the user interface is different for each application. Consistency and unification is important to ensure usability. The Facade pattern can solve this problem by providing a unified interface for all the applications in the suite. For example, all the applications might share the same basic command menus, since they all need to perform actions such as opening and saving files, copying and pasting using a clipboard, selecting which parts of the interface to show and which ones to hide, and so on. The Facade pattern can be used to define the commonality between the separate interfaces. Each separate application can extend this commonality as needed to provide access to functionality that is specific to those applications.

As an example for the second case, suppose you are using an IDE such as eclipse or Visual Studio to write and compile code. Compiling code requires several subsystems, such as a scanner, a tokenizer, a parser, a semantic analyzer, etc. Each of these can be a separate subsystem, and can be controlled by its own public API. Thus, one solution for allowing one to compile code in an IDE might be to provide individual menu items for each of these subsystems. After writing code, the Scanner command could be invoked to read the code into memory. Then the Tokenizer command could be issued to group the text into tokens. A Parse command could then be called to process the tokens to determine whether or not they form valid constructs according to the programming language being used. A Semantic Analysis command could subsequently be called to verify whether the syntactically correct tokens form sensible constructs. Eventually, a Generate Machine Code command could be issued to translate the source code into machine code to be executed. This is a very long, drawn out process. What happens if the commands are issued out of sequence? What if one of them is accidentally skipped? The Facade pattern can solve this problem by providing a single command; e.g., "Build Project" that will carry out all the necessary steps in the correct order, to compile the source code. Some flexibility is lost here, in that by encapsulating the entire sequence of actions into a single command it makes it impossible to perform each task alone. (For example, you might only be interested in running the code through to the parsing step, just to quickly check whether the code contains any syntax errors.) However, the benefits of hiding the individual functionality behind the Facade usually outweigh the disadvantages.

Flyweight

The Flyweight pattern supports the need for large numbers of objects by sharing a pool of common objects. If you think about a text document, the document contains a number of potential objects. Rows of text, columns of text, images, tables, paragraphs, and even individual characters could be objects. Such fine-grained use of objects promotes maximum flexibility, but at the expense of memory. A document that has 100,000 characters would have 100,000 character objects, plus objects for images, tables, and other elements. Using individual objects for all these elements is prohibitively expensive in terms of memory usage.

A Flyweight is a shared object that can be used in multiple contexts simultaneously. It behaves the same as a non-shared object would, and it can behave as an independent object in each context in which it is placed. A Flyweight knows nothing about the context it is in, and it cannot make assumptions or inferences about its context. The secret to the success of a Flyweight is its ability to separate its intrinsic state, which is that information about the Flyweight that is independent of the Flyweight's context; and its extrinsic state, which is that information about the Flyweight that is dependent on the Flyweight's context. In the text document example, a Flyweight would be a character whose intrinsic state consists of the character's identity (e.g., 'A', '4', '%'), which is usually stored as a character code. The extrinsic state consists of the character's location in the document, its font face, its size, etc. Formatting algorithms can determine the extrinsic information for the Flyweight.

Figure 5.9 shows an example of how a Flyweight pool works. In the figure, the word "ENGINEERING" is present in a text document. Each letter in the word is a Flyweight object taken from the Flyweight pool, which consists of all 26 letters in the English alphabet. There is only one instance of each letter object in the Flyweight pool, yet there are multiple occurrences of some of the letters in the word ENGINEERING. The Flyweights store only the character codes for the letters. Information about each letter's position, font, size, etc., will be stored elsewhere. This can be extended to all the words in a document. There may be 100,000 words in the document, but there will only be 26 Flyweight letter objects.

Proxy

Using the Proxy pattern, an object called a proxy, or surrogate, is used as a placeholder, and allows another object to control access to the proxy. The Proxy pattern is often used to reduce the cost of creating expensive objects when those objects, usually when those objects don't need to be fully instantiated until later. In other words, a Proxy defers the cost of creating an expensive object until it's absolutely necessary to create it. An example of where this situation might occur is a document that contains very large images that might take time to load, especially if they must be loaded over a slow network connection. Since only part of a document can be displayed on the screen at a given time, it doesn't make sense to load large images until the user scrolls to the point in the document where the images would be visible. A proxy object can be used to hold the place for the image until it is time to load the actual image. The proxy isn't just a static graphic, however, it is responsible for loading the image when the document instructs it to do so. Information about the dimensions of the image may be stored as part of the proxy's state to help facilitate proper display of the document.

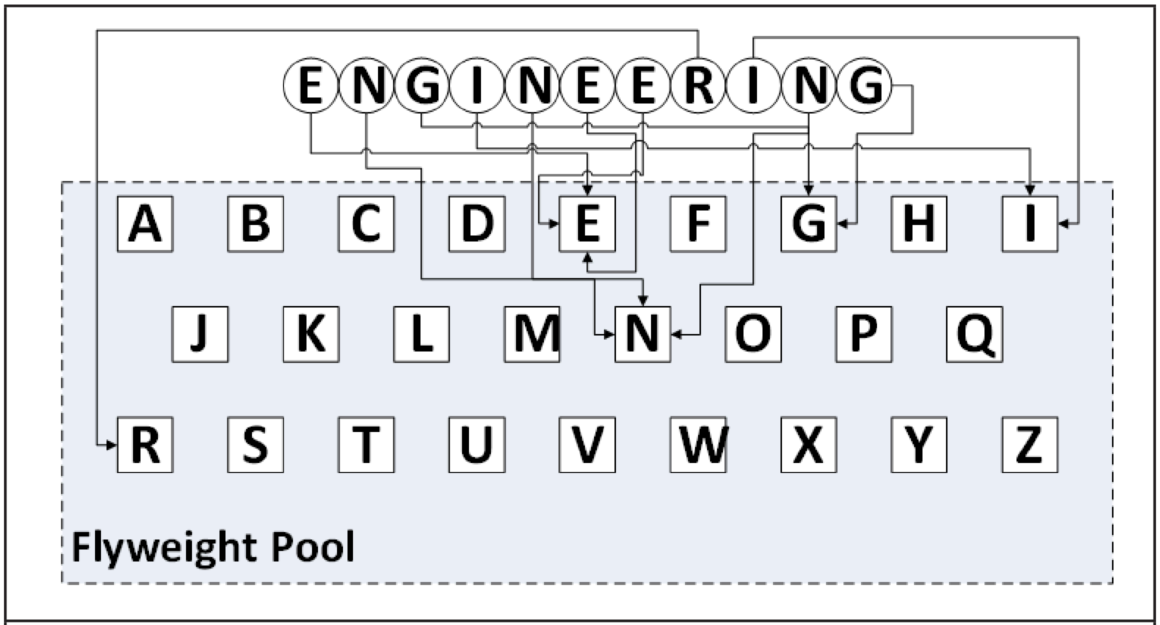


Figure 5.9. Example of how the Flyweight pattern can be used to model characters in a text document.

Behavioral Patterns

Chain of Responsibility

Coupling between objects; i.e., one object has an explicit reference to another object, is something that should be minimized in software design. The presence of coupling introduces dependencies in code that can be difficult to deal with if the software needs to be modified. It also provides opportunities for error during coding by violating the principle of information hiding. One of the most common situations where coupling occurs is when a sender object sends a message, but doesn't (and shouldn't) know in advance which object will receive the message. A good example is event handling in a GUI. Suppose, for example, you have a GUI that contains various buttons, list boxes, check boxes, etc., and you want to provide context-sensitive help for these elements. So if the user clicks to get help for the function of a particular button, the button sends an event to the help subsystem to display the context-sensitive help for the button. This poses no problem as long as there is help information available for the button. But what if there is no help information for a particular button? The user expects to receive some kind of help, since after all the software is advertised to display context-sensitive help. While there might not be any help available for the button, it might be useful to display some more general information about the button panel in which the button is located. The help information for the panel might not state anything about the specific button, but the user may be able to infer the information they need if they know more about what the function of the button panel is. But what if there is no help for the button panel, either? The help request could then move up to the next level in the hierarchy, perhaps the window that contains the button panel. In the worst case, the request might have to get sent all the way up to the application level.

The problem here is that the button shouldn't need to know anything about the hierarchy of the components in the GUI in which it resides. This would not only violate the principle of information hiding, but could also make code unmanageable if elements are added to, or removed from the GUI. What is needed is a way to propagate the message up the chain of GUI components while minimizing coupling between the individual components. This is where the Chain of Responsibility pattern comes in. Its purpose is to provide a mechanism where a message can be propagated up a chain of objects, while minimizing the extent to which those objects are coupled. There are a couple of ways this can be done. Each component could be allowed to have a reference to its successor component; i.e., the component that comes next in the chain, and only its successor component. There is minimal coupling here, since each component now knows about its successor, but this doesn't pose too serious a problem. The Chain of Responsibility pattern can be used in conjunction with the Composite pattern, to allow each element to have a reference to its parent, which serves as the successor for the message.

Another, more flexible, way to approach this is to use a Handler object to relay messages. The sender invokes the handler, with a reference to itself so the Handler knows which component is sending the message. The Handler also knows about the hierarchy in the chain of responsibility, so if a given component does not handle a request, the Handler can propagate the request on to the next object, and so on. Within the Handler there is significant coupling, since the Handler knows about the entire chain of responsibility hierarchy, but because the coupling is contained within the Handler, it is not a serious problem.

Command

There are instances where requests must be sent with no information about the type of request being sent, or about the receiver of the request. An excellent example that illustrates this situation is the set of menu commands for an application. The toolkits that are used to create the menu items can't know what the commands will be used for, nor can they know anything about the receivers of the requests issued by the commands. Thus, this information cannot be implemented as part of the menu items, but must be implemented as part of the application. The Command pattern addresses this issue by encapsulating commands and requests as objects. The encapsulation begins with an abstract Command class that declares an interface for executing a command or handling a request, for example via an `execute()` method. Concrete subclasses of the Command class implement the `execute()` method, and store a specified receiver-action pair. The receiver is typically a reference to an instance of the receiving class, and this instance may be abstract in order to accommodate families of receiver objects. When the `execute()` method is invoked, the concrete Command subclass sends the command or request to the receiver, yet doesn't know anything about the receiver's implementation. The receiver, however, knows how to deal with the request.

Figure 5.10 shows an example of how the Command pattern can be used to encapsulate the cut, copy, and paste commands for a Document object. The abstract Command class specifies the common interface that all Command subclasses must obey. The concrete subclasses, Cmd_Cut, Cmd_Copy, and Cmd_Paste, all provide specific implementations for the `execute()` method, and each one has a reference to the Document object. When the Cut command is invoked, for example, the Cmd_Cut object uses its reference to Document to invoke the `cut()` method of that Document. The individual commands don't know what it means to cut, copy, or paste, but each one knows just enough about the Document object to invoke the method that does know what it means to cut, copy, or paste.

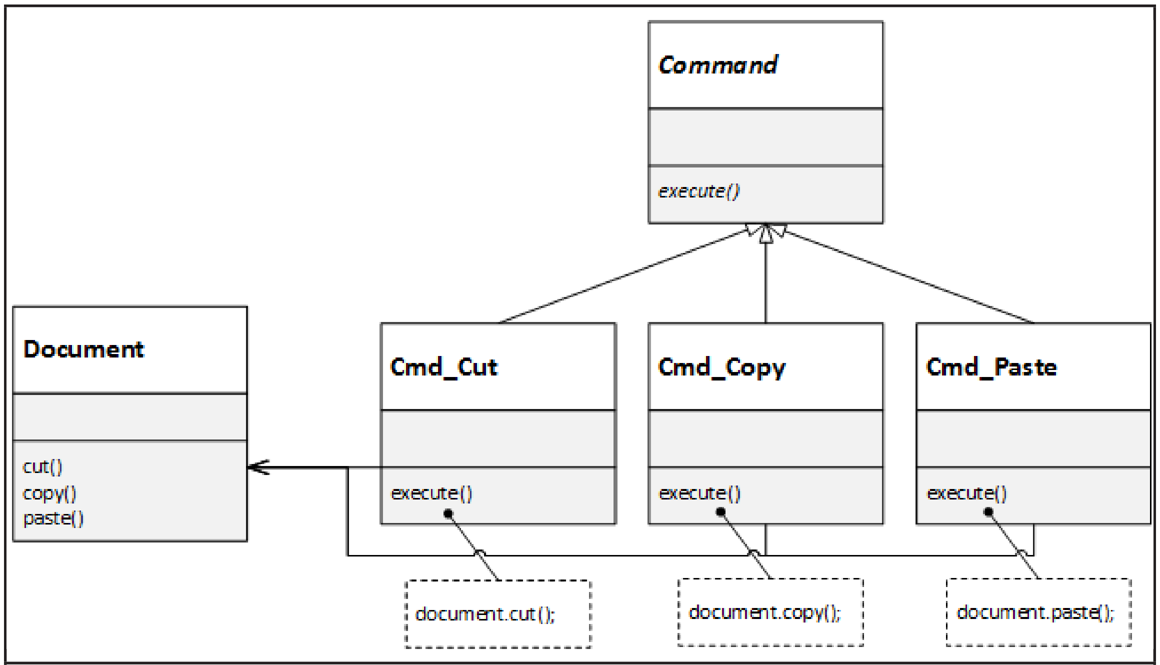


Figure 5.10. Example using the Command pattern to encapsulate various commands for a Document.

Interpreter

The Interpreter pattern defines a representation for the grammar of a language, along with an interpreter that uses the grammar representation to interpret the language. The term language is used a bit loosely here. It can refer to a programming language, a natural language, a regular expression, or any combination of symbols plus a grammar that defines what permutations of those symbols are valid. As an example, consider the snippet from the Java language grammar shown in the left panel of Figure 5.11. The Interpreter pattern would use a separate class to represent each of the grammar productions, so for the above example there would be 9 classes. Each terminal and non-terminal on the right-hand side of each expression would be instances of one of these classes. The Interpreter combines one or more of this instances to form an abstract syntax tree, that can be used to determine whether the combination of terminals/non-terminals is a valid construct according to the grammar. The right panel of Figure 5.11 shows an example of an abstract syntax tree based on the above grammar, for the decimal long integer literal, 12L.

Generally speaking, the Interpreter pattern works best when the grammar is relatively simple. Complex grammars result in unmanageable syntax trees due to the numbers of terminals and non-terminals on the right-hand side of the productions. The Interpreter pattern can be used in conjunction with the Flyweight pattern, where Flyweights are used to store frequently occurring elements such as the non-terminal values.

DecimalIntegerLiteral ::= DecimalNumeral[IntegerTypeSuffix]

DecimalNumeral ::= 0 | NonZeroDigit[Digits] | NonZeroDigit Underscores Digits

NonZeroDigit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Digits ::= Digit | Digit[DigitsAndUnderscores] Digit

Digit ::= 0 | NonZeroDigit

DigitsAndUnderscores ::= DigitOrUnderscore {DigitOrUnderscore}

DigitOrUnderscore ::= Digit | _

Underscores ::= _ {_}

IntegerTypeSuffix ::= l | L

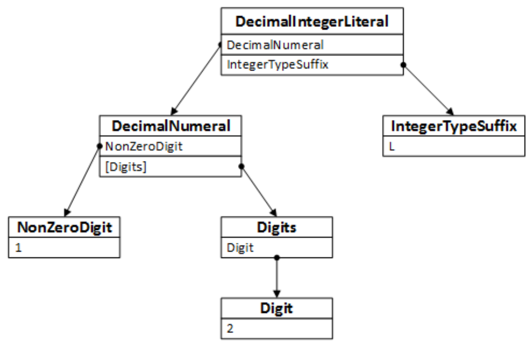


Figure 5.11. Example of an abstract syntax tree created by an Interpreter. The left panel shows the grammar used by the Interpreter. The right panel shows the abstract syntax tree for building the decimal long integer literal `12L`.

Iterator

The Iterator pattern provides a way to traverse elements of a collection without needing to know how the collection is implemented internally. Array-based collections, for example, use an underlying array to store elements. Linked collections use nodes that are linked to each other in some fashion. Both types of collections can be traversed, and for a traversal inside the class of one of these collections it is fine to exploit knowledge of the underlying structure. But the public interface to collections should not require clients to know what the underlying implementation is, nor should it provide this information to clients, in keeping with the principle of information hiding.

The Iterator pattern solves this problem by providing a consistent, public interface for traversing a collection. This interface will have methods for advancing to the next element, retreating to the previous element, and possibly adding an element at the current position or removing the current element. The same method signatures will be used for all collections. Since the implementation details of the collection are hidden, it is the responsibility of the collection to support traversal via an Iterator. Typically, factory methods are defined within the collection class that return the correct type of Iterator. The Iterator itself can be subclassed to accommodate implementation-specific iteration. However, the main Iterator class is usually allowed to function as a generic iterator that can be used on all types of iterable collections.

One common problem encountered with using an Iterator is the possibility that a collection can be modified via the collection's methods while the collection is being traversed by the Iterator. Concurrent modification of this nature is problematic, so a mechanism should be provided to allow an Iterator to detect when a collection has been modified externally, thus enabling it to fail fast before it can create a more serious problem. Consistent with preventing concurrency problems, an Iterator must be deleted, or deallocated once it is no longer needed.

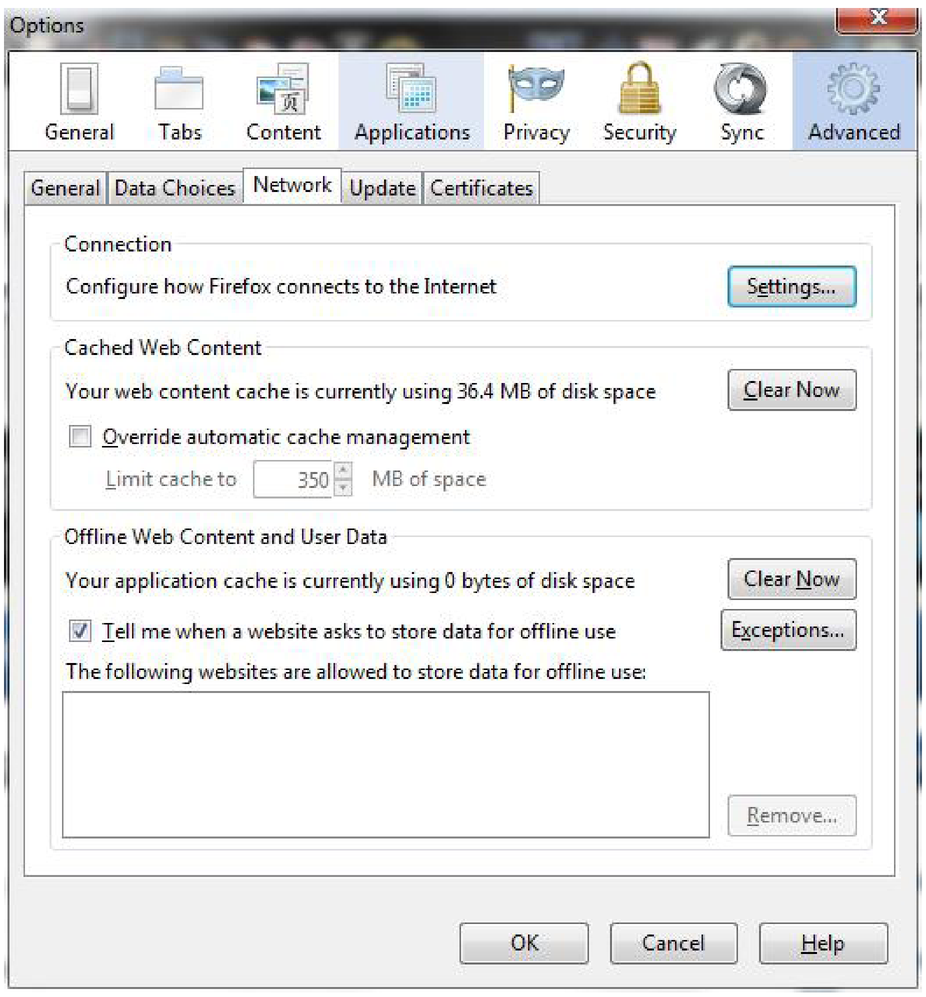
Mediator

Modular design encourages the distribution of behavior among the components of a system, in order to obey the principle of information hiding, and to limit the impact of change within the system. For example, if you have an undistributed, monolithic system, every change can require reviewing the entire system for potential side effects of the change. In a modular, distributed system, however, it is often possible to limit the impact of a change to only a few modules. One major consequence of this modularity is the need for the separate components to interact with each other. This interaction inevitably leads to coupling, where one component explicitly knows about the existence of another component, and may even know certain implementation details of that component. In a system with many components it is easy to have so much coupling between components that for all intents and purposes the system has become monolithic in spite of its modularity.

The Mediator pattern seeks to avoid the coupling dilemma by providing a component that is specifically responsible for handling the interconnections between components. This single component may have a very high degree of coupling, since it knows about many other components and how they interact, but at least the coupling is restricted to a single component. Inheritance and polymorphism can be used to anonymize component references to reduce the number of explicit references to concrete objects, and thus reduce the impact of coupling within the Mediator.

The complexities of a GUI, with all its buttons, list boxes, checkboxes, etc., make it an ideal example for how the Mediator pattern can work. It is common to have a GUI where some of the buttons or menu items are grayed out until the proper prerequisites have been met to make them active. For example, consider the Network preferences tab on the Advanced Options dialog box for Firefox, shown in Figure 5.12. The checkbox, “Override automatic cache management” is currently unchecked. Managing the cache limit requires specifying a certain number of megabytes of storage space. Since the specification of storage space is dependent on having cache management turned on, the spinner component where the storage space is specified is grayed out to prevent setting this value unless cache management is on. The act of turning on cache management needs to activate the spinner, yet there is the question of whether or not the checkbox should know about the spinner. It should not, because although a spinner is used in this case, a simple text box could just as easily have been provided, as could a list box containing pre-populated values. In fact, the checkbox shouldn’t even be aware that a storage limit needs to be set, since that information may or may not be required.

Enter the Mediator pattern, which in this case could be represented by the dialog window, or it could be represented by a separate, non-GUI component. When the user checks the cache management check box to turn on cache management, a message is sent to the Mediator, which determines what action to take. This action could be to use a default value for storage, no limit on storage, or prompt the user for a value. In this case, the Mediator responds by activating the spinner to allow the user to select a storage space value, within the limits of the populated values. Selecting a value may result in sending another message to the Mediator, which responds by taking some other, appropriate action. The key point here is that the component for turning on cache management has been decoupled from the component for specifying the amount of storage space — neither component knows about the other. However, the Mediator knows about both components, and is able to detect when they have been manipulated by the user, and take the appropriate course of action.



Connection

Configure how Firefox connects to the Internet [Settings...]

Cached Web Content

Your web content cache is currently using 36.4 MB of disk space [Clear Now]

☐ Override automatic cache management

Limit cache to 350 MB of space

Offline Web Content and User Data

Your application cache is currently using 0 bytes of disk space [Clear Now]

☑ Tell me when a website asks to store data for offline use [Exceptions...]

The following websites are allowed to store data for offline use:

[Remove...]

Figure 5.12. The Advanced --> Network preferences tab for Firefox.

Memento

The Memento pattern captures and externalizes an object's internal state, without violating encapsulation; i.e., the principle of information hiding. This pattern is useful in situations where you need to keep track of an object's internal state so that even after the object's state changes, you can return to exactly that stored state. The history feature of many software applications is an excellent example where the Mediator pattern is used. For example, suppose you are editing an image in Photoshop, and you perform a series of image processes; e.g., resize, crop, set transparency, blur, etc. Now suppose you don't like the current result, and you want to go back to the state of the image 3 manipulations prior to the current state. Photoshop's history tab allows you to go back to this state by clicking on the action in the editing history that led to the state you are trying to return to. Storing a snapshot of an object's state in this way has advantage over trying to algorithmically compute the reverse of an action. The reversal could be slightly off, and some actions are non-reversible. For example, if you decrease the resolution of an image, you can't algorithmically restore the original resolution, because information was lost when the resolution was decreased. Maintaining snapshots of the image's states, however, doesn't rely on reversing an action, it simply lets you return to a clone of the image at a given point in the editing history.

Observer

In the Model-View-Controller (MVC) paradigm, the application logic for a system is encapsulated in a set of classes that forms the model, while the way in which information about the model is displayed is handled by a separate set of classes known as the view. The model and the view must be synchronized to ensure that what is being displayed on the screen accurately reflects the state of the model. A naive way to facilitate this would be to let the view have an explicit reference to the model, and periodically poll the model for its current state. Alternatively, the model could have an explicit reference to the view, and "push" updates to the view. While this works, it introduces coupling between the model and the view that should be avoided. The model, for example, doesn't need to know how the view is displaying the model's state information. The view also should not need to know how to query the model for information, because that would require knowing something about how the model is implemented.

The Observer pattern solves this problem by allowing the dependents of an object to be notified and updated when the object changes state, without explicit coupling between the object and its dependents. The dependents are known as observers, while the object is referred to as being observable. The explicit coupling between objects is replaced with anonymous references to more generic objects; e.g., Observers and Observable. The Observable object can store a collection of Observers, knowing nothing about these objects except their public, Observer interface. When the Observable object changes state, it can broadcast the update to all the Observers in its Observer list.

One of the main stumbling blocks to effectively using the Observer pattern is the issue of how to generically define what constitutes an "update". An update can mean any number of things, yet all updates must be able to be generically passed to Observers. For example, in a calculator application an update might consist of displaying the number 9 after the user presses the '9' button. An update in a word processing application might be in the form of 10 pages of text pasted into the current location. There is no way to efficiently generically declare an update, since an update that consists of a digit is not compatible with an update that consists of 10 pages of text. One possible solution to this problem is to use the Mediator pattern to encapsulate knowledge of the details of an update in a separate class, and allow the Mediator to sort out what should be reflected in the view, and how it should be displayed. The Controller in the MVC paradigm is often used in this way.

State

In some cases an object's behavior depends tightly on the object's current state, and it must be able to change its behavior as its state changes. Accomplishing this goal is the purpose of the State pattern. As an example, assume you have a TCPConnection class that represents a network connection. Assume an instance of TCPConnection can be in one of three possible states: 1) Established, 2) Listening, or 3) Closed. The behavior of a TCPConnection object depends on its current state. So for example, a request to open a connection will have different effects depending on whether the TCPConnection object is currently in the Established state, the Listening state, or the Closed state.

The State pattern deals with this problem by encapsulating the state of a TCPConnection in a separate class; e.g., TCPState. The abstract TCPState class has the same set of relevant methods as TCPConnection. TCPState can be subclassed, and the methods overridden, to model the three possible states of a TCPConnection. Each subclass will behave in a state-specific manner, satisfying the requirement that TCPConnection be able to change its behavior as its state changes. When the state of

TCPConnection changes, TCPConnection simply changes the class it uses for TCPState. Polymorphism ensures all subclasses of TCPState can be treated through the common interface, which mirrors the capabilities of TCPConnection. There is flexibility in this design, since the behavior of a given state can be easily changed without having to go into the TCPConnection class. In other words, a new implementation of a state can be swapped in to replace an old implementation. Note there is some mandatory coupling here, as the TCPState classes must know enough about TCPConnection to mirror the relevant method signatures. However, this coupling is outweighed by the flexibility offered by using the State pattern.

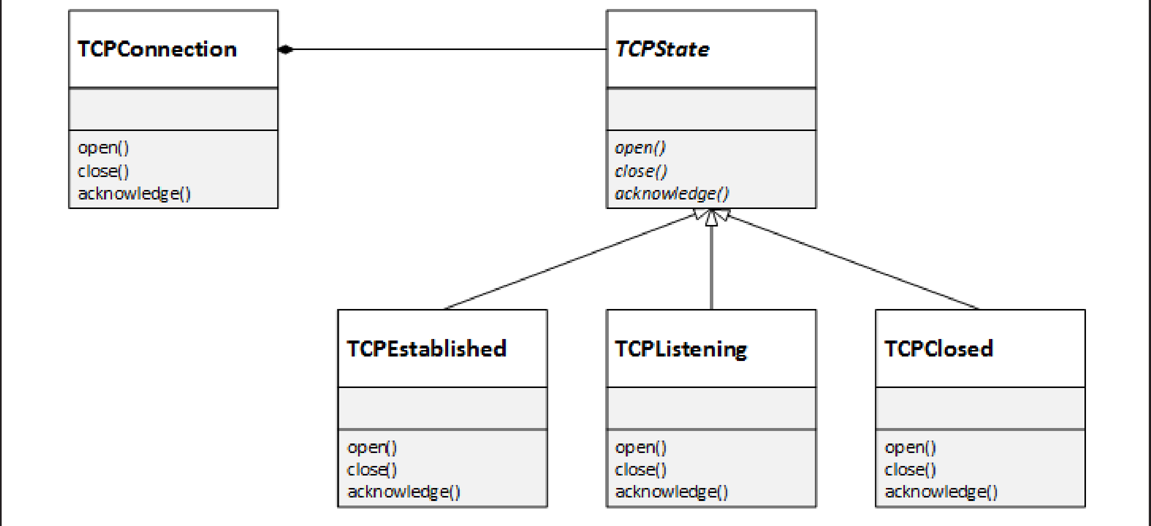


Figure 5.13. Example of using the State pattern to model a TCP connection, whose behavior must change as its state changes.

Strategy

The Strategy pattern functions similarly to the State pattern. With the Strategy pattern, the goal is to be able to define a family of algorithms or behaviors, encapsulate them, and have them be interchangeable. In other words, one algorithm should be able to seamlessly replace another. An obvious example where the Strategy pattern would be useful is in a game application, where a computer-controlled opponent can have its difficulty level altered simply by swapping one playing strategy for another. Figure 5.14 shows how this can be applied to a chess game.

Using the Strategy pattern can eliminate highly complex conditional structures, which while they will achieve the same goal, are cumbersome and error-prone. The number of strategies can be easily extended by creating new subclasses of the abstract strategy class. The only caveat is that the client must know about the different strategies that are available, in order to interchange strategies.

Template Method

Suppose you have several classes that have methods that perform behaviors that are partially similar to each other, but also have class-specific behavior. Figure 5.15 illustrates this using the example of a Document class that must be able to display three different types of documents: 1) plain text documents, 2) HTML documents, and 3) PDF documents. One way to handle this is to write separate classes for each document type, and implement each one individually according to the needs of the document type it handles. This approach is shown in Figure 5.15. As can be seen from the figure, while the implementation details of the printPage() method will certainly differ between the three classes, the same basic process is being done: 1) print the header, 2) print the page body, and 3) print the footer.

This approach works, but a more efficient solution can be achieved by abstracting out the commonality between the three classes into an abstract class. The printPage() method can be defined in the abstract class to set the sequence in which the parts of the document are displayed. The printPage() method is common to all the document types, but the implementations for the methods invoked in printPage() will vary depending on the document type. This is where the Template Method pattern comes in. Using this pattern, the printPage() method can be moved up to an abstract Document class. The document type-specific methods can be defined in subclasses of the Document class. This is shown in Figure 5.16.

To summarize, the sequence of actions is the same for all three document types, but the implementations for those actions are encapsulated in other classes. This is the hallmark characteristic of the Template Method pattern. The skeleton of an algorithm is shared among multiple subclasses, but each subclass is free to redefine how each part of the algorithm is carried out.

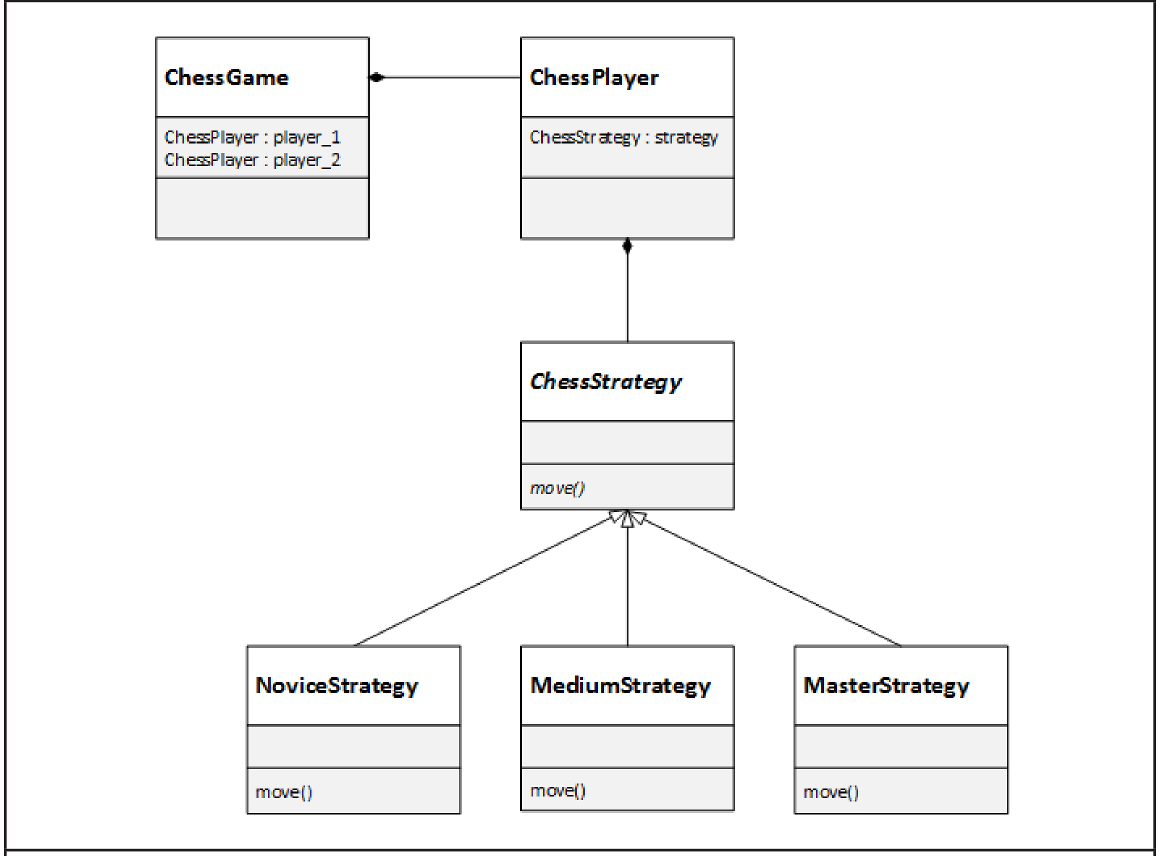


Figure 5.14. Example of using the Strategy pattern to encapsulate different playing strategies for a computer-controlled chess player.

```
public class PlainTextDocument
{
      // printing a page is common to all document types
      public void printPage(Page page)
      {
            // Unique to PlainTextDocument
            printPlainTextHeader();
            printPlainTextPageBody();
            printPlainTextFooter();
      }
}
```

```
public class HTMLDocument
{
      // printing a page is common to all document types
      public void printPage(Page page)
      {
            // Unique to HTMLDocument
            printHTMLHeader();
            printHTMLPageBody();
            printHTMLFooter ();
      }
}
```

```
public class PDFDocument
{
      // printing a page is common to all document types
      public void printPage(Page page)
      {
            // Unique to PDFDocument
            printPDFHeader();
            printPDFPageBody();
            printPDFFooter ();
      }
}
```

Figure 5.15. Example of three document types, all of which have the common behavior of printing a page.

Visitor

The Visitor pattern lets you define a new operation without changing the classes of the elements on which it operates. As an example of the application of the Visitor pattern, let's go back to the compiler example we discussed previously. The abstract syntax tree that is generated by a compiler will need to support several types of operations, such as checking for valid variable definitions, checking for proper object initializations, checking for dead code, etc. These operations may need to treat the nodes in the tree differently, which leads to a design where the tree is composed of different classes of nodes; e.g., a node class for variable references, another node class for assignments, yet another node class for exception handling, etc. While all the different operations we need could be defined in each node class, this creates a couple of problems:

1. Every node in the tree will have copies of the operations. If the tree is large, the space consumption could be prohibitively expensive.

2. There may be cases where certain node classes inherit methods that don't really apply to them.

What we really need is a way to add the operations we want separately, as each node is visited, and have the node classes themselves be independent of the operations that can be performed on them. The Visitor pattern enables us to accomplish this goal by creating two separate hierarchies: 1) one hierarchy for the elements stored in the nodes, and 2) a second hierarchy for the visitors that define the operations that can be performed on the elements. Each element node has an accept() method that takes a node visitor as an argument.

In the compiler example, suppose we want to perform type checking on the variables. The compiler would create the abstract syntax tree, with each node having implemented an accept() method that takes a visitor as an argument. The compiler would then create the appropriate type of Visitor; e.g., TypeCheckingVisitor, and call the accept() method on each element node in the syntax tree, passing the TypeCheckingVisitor object as an argument.

```
public abstract class Document
{
        // printing a page is common to all document types
        public void printPage(Page page)
        {
                printHeader(page);
                printPageBody(page);
                printFooter(page);
        }

        public abstract void printHeader(Page page);
        public abstract void printPageBody(Page page);
        public abstract void printFooter(Page page);
}
```

```
public class PlainTextDocument extends Document
{
        public void printHeader(Page page)
        { // Unique to PlainTextDocument }

        public void printPageBody(Page page)
        { // Unique to PlainTextDocument }

        public void printFooter(Page page)
        { // Unique to PlainTextDocument }
}
```

```
public class HTMLDocument extends Document
{
        public void printHeader(Page page)
        { // Unique to HTMLDocument }

        public void printPageBody(Page page)
        { // Unique to HTMLDocument }

        public void printFooter(Page page)
        { // Unique to HTMLDocument }
}
```

```
public class PDFDocument extends Document
{
        public void printHeader(Page page)
        { // Unique to PDFDocument }

        public void printPageBody(Page page)
        { // Unique to PDFDocument }

        public void printFooter(Page page)
        { // Unique to PDFDocument }
}
```

Figure 5.16. Example from Figure 5.15, but using the Template Method pattern to allow subclasses of Document to redefine methods for printing a page.

5.8 User Interface Design

Software that will be used directly by humans must provide some way for humans to interact with, and use it. The user interface is the user's gateway to the functionality the software provides. Considerable effort must go into the design of the user interface, because if the user interface is no good, people are less likely to use the software, regardless of its benefits. Sommerville notes six design principles that should be taken into account when designing a user interface. I repeat them here, and give an example for each principle.

User Familiarity

The language and features of the interface should be recognizable to the people who will use the software. For example, if you are building an interface for software used to create music CD's, it is probably better for users to have a menu command called, "Burn CD", than to have one called, "Write .WAV bytes to optical disk". Even the term "burn" is meaningless unless you understand how data are written to a CD. Nevertheless, the term has successfully made itself a part of our culture.

Consistency

Commands that can be performed in multiple ways should be given the same label. For example, "Burn CD" should be labeled as such whether it appears in the main menu, a popup menu, or on a button. When designing software product families, commonly used commands such as opening and closing files, copying and pasting, etc., should all be labeled consistently throughout each product in the family, and should use the same set of shortcut keys.

Minimal Surprise

The user interface should never produce any message or result that the user does not expect. Unfortunately, these are still all too common in much of today's software. Whenever the user executes a command, there should always be some clear notification that the command was executed successfully, or that the attempt failed. Users should never have to ask the question, "Did it do what I told it to?"

Recoverability

The interface should allow users to recover from errors. This refers to user errors, not software failures. For example, providing an "Undo" feature allows the user to erase the most previous action in case the user accidentally does something wrong. All destructive actions (delete, cut, overwrite, etc.) should prompt the user if these actions cannot be undone.

User Guidance

If the user needs assistance, there should be help available, preferably in a context-sensitive format. If something goes wrong, meaningful error messages should be displayed. Error messages like, "Illegal stack operation at address 0x038A" are of little use to most users.

User Diversity

Creators of a user interface should realize that the software may be used by people with varying degrees of experience, people from different cultures, and people with different physical capacities. Even though it's seldom possible to build an interface that will satisfy everyone, care should be taken not to alienate or offend any potential users.

Together, all the aforementioned principles comprise what is commonly referred to as "user-friendliness". They are also a measure of software's usability, one of the key characteristics used to determine software quality.

5.9 Prototyping

User interface construction lends itself very well to the prototyping process model, since it's usually difficult to determine precisely how the customer wants the interface to be. Mock-ups can be created very quickly and shown to the customer. These mock-ups can be static images created with an application such as Photoshop, or even drawn on paper. Alternatively, applications such as Visual Basic can be used to quickly build a dynamic, pseudo-functional interface that will allow users to play around with it and get a feel for what it will actually be like to use the software. Dummy data can be used, if necessary, to enhance the realism of the prototype. One word of caution, however, is in order. If the prototype appears to be too "real" it may lead the customer to believe that the back-end is also finished, and they may try to move up the release date or demand additional features. This generally stems from ignorance on the part of the customer as to what is involved in building software. Customers and end users typically only see the front end, and fail to realize that the front end is nothing more than the control panel.

5.10 At what point in development can (should) the user interface be developed?

In a well-designed system, once the requirements have been finalized, the user interface should be able to be developed in parallel with the rest of the system. Back end development should not depend on the user interface, and user interface development should not depend on completion of the back end. In general, the user interface should be very loosely coupled to the back end. It should be possible to swap out one version of the user interface with another version without any loss of functionality. The Model-View-Controller design pattern illustrates this point. The view represents the user interface, the model represents the back end, and the controller provides a link between the two. If designed well, a graphical view could be swapped with a text-based view, for example. The functionality of the software is still all there, it's just accessed differently.

From a testing standpoint, it can be helpful to have portions of the user interface completed prior to testing, although this isn't always possible. The user interface can serve as the test harness, and reduce the amount of scaffolding code that must be written.

5.11 GUIs

Most software built for use in personal computers uses a graphical user interface, or GUI. GUIs are popular because they provide the very valuable "WYSIWYG" (What You See Is What You Get) capability. Anyone who has used older word processors, such as WordPerfect prior to version 6.0, probably remembers seeing text of various colors, parts of which may be highlighted in other colors, on a blue background. As a user, you have to remember that white text is normal, yellow text is bold, a certain color of highlight means the text is underlined, etc. GUIs show the text exactly as it is supposed to appear in print form. GUIs also provide extremely valuable capabilities such as drag-and-drop, multiple windows, clickable icons, etc.

However, there is a tradeoff for using a GUI. That tradeoff is slower running time. Graphical artifacts take longer to draw on the screen, and each open window consumes memory. If enough windows are opened it will eventually force the operating system to begin using virtual memory, which itself causes a performance hit. As computers have gotten faster, this decrease in performance has become less noticeable, but it is there.

5.12 Customizable User Interfaces

In an effort to appease all users, some developers provide user interfaces that are customizable or extensible. Most software already provides multiple ways to accomplish the same task (main menu, context menu, shortcut key). If you want to use a different shortcut key for a particular function, you can change it. If a custom toolbar is desired, it can be created by removing icons for unwanted or unused tools, and adding icons for tools that are not part of the default toolbar. The products of the Microsoft Office suite all allow this capability. Some software, such as Dreamweaver, even allows you to add your own commands to the main menu. (For those of you unfamiliar with Dreamweaver, it is a program used for creating web-based content. Its main menu can be extended by writing a custom function in Javascript and adding it to the Command menu).

Some degree of GUI customization is expected. There are people who, due to some physical ailment, are unable to use a mouse. Such individuals must therefore be able to do everything they need to do using the keyboard or some other input device. People with vision problems often need the ability to enlarge the text on the screen in order to read it. These types of customization are more appropriately categorized as accessibility features.

Extensive customization can present development problems. Every control of the user interface, whether it is a menu command, push button, scrollbar, etc., must be tested to ensure that it works properly. This means that each user interface artifact must be capable of being manipulated with reasonable ease. The target area must be sufficiently large. Expecting users to click on controls that are only a single pixel wide is probably not a good idea. Proper functioning also means that when a particular artifact is chosen, the correct function is performed. If there are three different ways to perform a function, all three ways must be tested, and they must all produce the same result.

5.13 Coding

Sommerville doesn’t really have a chapter in his text dedicated to the coding phase of software development, so I’ll devote part of the lecture to this topic. Part of the purpose of this course is to dispel any ideas you may have that software engineering is simply a glorified title for code hacker. Many people still have an unfortunate tendency to equate coding with software engineering to the point they consider the two to be synonymous. There exist notions of software engineers as introverted geeks who sit in tiny cubicles for 15-20 hours a day doing nothing but writing code. Sadly, in many cases this is closer to truth than fiction, which makes it all the more difficult to relay a distinction between software engineering and coding. To give credit where it’s due, there are some code hackers out there who are good enough to design some of their programs on the fly, and still end up with reasonably good products. I don’t think most of us can do that, except for relatively simple problems. Writing software is easy; writing high quality software can be quite hard.

Probably the reason why the code receives so much focus is because it is the code that actually gets compiled and translated into the working software. If you’re following the Extreme Programming process model, you are taught that the code should be self-documenting for the most part, and that supplemental documentation is largely a waste of time because no one will ever look at it. Movies and television also tend to glorify programmers as “geeks” who do nothing but write code all day, and eventually become so good at it they don’t need the shackle of documentation or design—they’re just good enough to whip something out in a matter of hours (or maybe a few days if the problem is really complex). In the early days of software engineering, where all the computing was done on time-shared mainframes, and the software engineers of the day were only allotted a precious, limited amount of time to actually compile and run their code, those engineers spent a lot more time on design than may have been apparent. They had to, because there was no telling how long they might have to wait before they were able to try again. So they would review their code for faults while they waited for their next turn on the mainframe.

As personal computers became more widely available, and as they became faster, it became possible to take shortcuts. Since people tend to be short on patience, rather than spend a lot of time trying to figure out on paper whether or not something would work, programmers could just run the code and let the errors (or lack thereof) tell them how they did. Unfortunately, it's easy to end up with code that appears to work, at least for all the test cases that have been run, but then crash later when a case is encountered that exposes a design flaw. The market's demand for rapid software availability, combined with the insatiable corporate thirst for profit, only exacerbates the situation by promoting more time spent on writing code. This could partially explain why a lot of software written in the 1960's and 1970's seemed to be more stable than software written later. Of course, another driving factor is the fact that some software is becoming ever more complex as it evolves.

An interesting philosophy regarding coding relates to the notion of an executable specification, which I mentioned briefly in a previous module. With an executable specification, functional software is generated automatically from the requirements specification. There essentially is no separate design phase, except possibly to deal with a few non-functional requirements, because all the design details follow naturally from the requirements. Source code can be generated, compiled, and tested automatically. Alternatively, machine code can be generated directly from the specification, eliminating the need for source code. In order for this to work, the requirements must be expressed in such a way that they are readable by a software application. Typical human languages contain too many ambiguities to work for this. What we would need is a language that is capable of expressing the requirements unambiguously. Such languages exist, and we will talk about them later when we cover formal methods in more detail. For now, suffice it to say that this is not a viable approach for most software projects because for formal methods to work requires the customer to know, completely and unambiguously, what they want the software to do, and this is never the case. Also, it is exceedingly difficult and costly to correctly define requirements using formal methods, so the motivation to use them is low.

5.14 Coding Styles and Readability

I only want to spend a brief amount of time on this topic. I'm sure all of you have been exposed to at least one style of coding style and documentation. Many styles exist, and each has its own cadre of loyal supporters. Some styles have advantages over others, but I doubt anyone could prove that one style is "best". The goal of any coding style is to make the code readable and understandable to anyone (i.e., other software engineers) who may have a need to look at it. It's always a good idea to assume that at some point in time someone may have to make sense of another person's code, even though they may not have been a member of the team that originally developed the project. Even if no one but the author will ever need to look at the code, it's never a good idea to assume memory is infallible. An author may look at code they wrote sometime in the past and wonder what the code does, or why it does what it does.

5.15 Code Obfuscation

Obfuscated code is code that has been written in such a way that it is not easily intelligible. This contradicts the notion that code should be readable and understandable by others. There are two primary reasons why someone might want to obfuscate (or “shroud”) their code:

1. Security

Obfuscation deters reverse engineering by preventing others from easily understanding how a particular problem was solved by simply looking at the source code. This is good if you want to have an edge over competitors, although the edge is likely to be only temporary. It can also deter hacker attacks by making it difficult for a hacker to pinpoint vulnerabilities in the code. Of course, the assumption here is that a human will be looking at the code. Obfuscated code will not hinder a computer program designed to analyze code, because even though obfuscated code may look strange to a person, it must still follow the syntactical grammar of the language in which it is written.

2. Recreation

Those who are interested in a challenge may try to write cleverly obfuscated code simply for enjoyment, or to enter a contest. One of my personal favorites is the following obfuscated C code which, when executed, displays the lyrics to the “12 Days of Christmas”:

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,”%s %d %d\n”):9:16:t<0?t<-72?main(_,t,
“@n’+,#’/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/’r :’d*’3,}{w+K w’K:’+}e#’;dq#’l \
q#’+d’K#!/+k#;q#’r}eKK#}w’r}eKK{nl]’/#;#q#n’){)#}w’){){nl]’/+#n’;d}rw’ i;# \
){nl]!/n{n#’; r{#w’r nc{nl]’/#{l,+’K {rw’ iK{;[{nl]’/w#q#n’wk nw’ \
iwk{KK{nl]!/w{%’l##w#’ i; :{nl]’/*{q#’ld;r’}{nlwb!/*de}’c \
;;{nl’-{}rw]’/+,}##’*}#nc,’,#nw]’/+kd’+e}+;#’rdq#w! nr’/ ‘) }+}{rl#’{n’ ‘)# \
}’+}##(!!/”)
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a==’/’)+t,_,a+1)
:0<t?main(2,2,”%s”):*a==’/’||main(0,main(-61,*a,
“!ek;dc i@bK’(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry”),a+1);}
```

References

1. Parnas, D. L., On the criteria to be used in decomposing systems into modules, *Communications of the ACM* 15:12 (Dec. 1972), 1053-8.
2. Trireme International, Ltd. UML Tutorial. http://uml-tutorials.trireme.com/ (date unknown).
3. Object Management Group website. http://www.omg.org.
4. Website for Gentleware, creators of Poseidon for UML. http://www.gentleware.com
5. Website for Tigris.org, creators of ArgoUML. http://argouml.tigris.org/
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995.
7. Alexander, C., Ishikawa, S., and Silverstein, M., with Jacobson, M., Fiksdahl-King, I., and Angel, S., *A Pattern Language*, p. x, Oxford University Press, New York, NY, 1977.