Chapter 2

Abstraction

If you open an atlas you will often first see a map of the world. This map will show only the most significant features. For example, it may show the various mountain ranges, the ocean currents, and other extremely large structures. But small features will almost certainly be omitted.

A subsequent map will cover a smaller geographical region, and will typically possess more detail. For example, a map of a single continent (such as South America) may now include political boundaries, and perhaps the major cities. A map over an even smaller region, such as a country, might include towns as well as cities, and smaller geographical features, such as the names of individual mountains. A map of an individual large city might include the most important roads leading into and out of the city. Maps of smaller regions might even represent individual buildings.

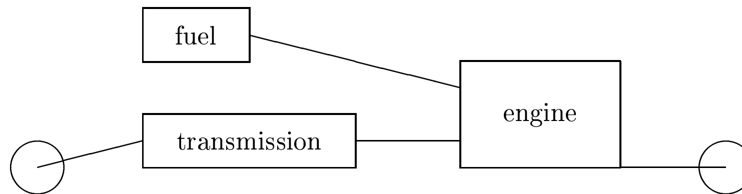
Notice how, at each level, certain information has been included, and certain information has been purposely omitted. There is simply no way to represent all the details when an artifact is viewed at a higher level of abstraction. And even if all the detail could be described (using tiny writing, for example) there is no way that people could assimilate or process such a large amount of information. Hence details are simply left out.

Fundamentally, people use only a few simple tools to create, understand, or manage complex systems. One of the most important techniques is termed *abstraction*.

> **Abstraction**
>
> Abstraction is the purposeful suppression, or hiding, of some details of a process or artifact, in order to bring out more clearly other aspects, details, or structure.

Consider the average persons understanding of an automobile. A laymans view of an automobile engine, for example, is a device that takes fuel as input and produces a rotation of the drive shaft as output. This rotation is too fast to

connect to the wheels of the car directly, so a transmission is a mechanism used to reduce a rotation of several thousand revolutions per minute to a rotation of several revolutions per minute. This slower rotation can then be used to propel the car. This is not exactly correct, but it is sufficiently close for everyday purposes. We sometimes say that by means of abstraction we have constructed a *model* of the actual system.



In forming an abstraction, or model, we purposely avoid the need to understand many details, concentrating instead of a few key features. We often describe this process with another term, *information hiding*.

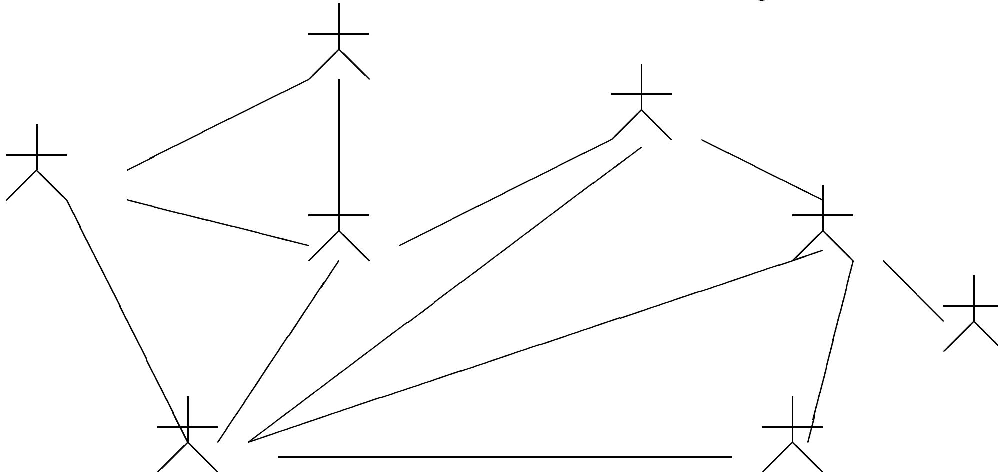
Information Hiding

Information hiding is the purposeful omission of details in the development of an abstract representation.

2.1 Layers of Abstraction

In a typical program written in the object-oriented style there are many important levels of abstraction. The higher level abstractions are part of what makes an object-oriented program object-oriented.

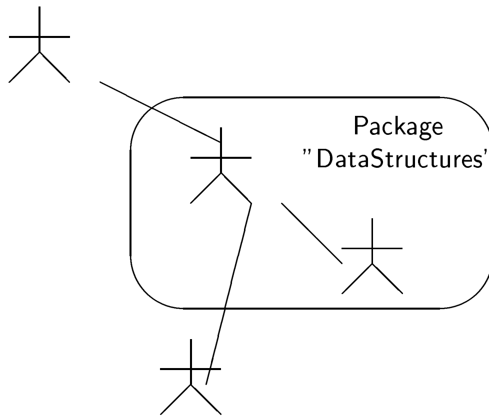
At the highest level a program is viewed as a "community" of objects that must interact with each other in order to achieve their common goal:



This notion of community finds expression in object-oriented development in two distinct forms. First there is the community of programmers, who must interact with each other in the real world in order to produce their application. And second there is the community of objects that they create, which must interact with each other in a virtual universe in order to further their common goals. Key ideas such as information hiding and abstraction are applicable to both levels.

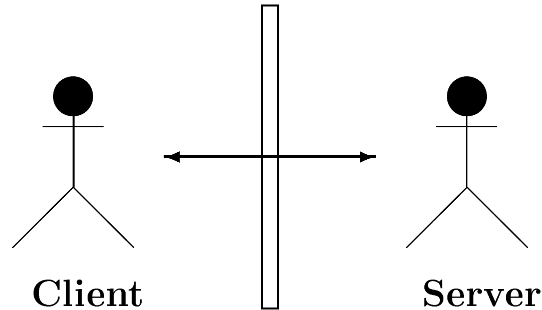
Each object in this community provides a service that is used by other members of the organization. At this highest level of abstraction the important features to emphasize are the lines of communication and cooperation, and the way in which the members must interact with each other.

The next level of abstraction is not found in all object-oriented programs, nor is it supported in all object-oriented languages. However, many languages permit a group of objects working together to be combined into a unit. Examples of this idea include *packages* in Java, *name spaces* in C++, or *units* in Delphi. The unit allows certain names to be exposed to the world outside the unit, while other features remain hidden inside the unit.



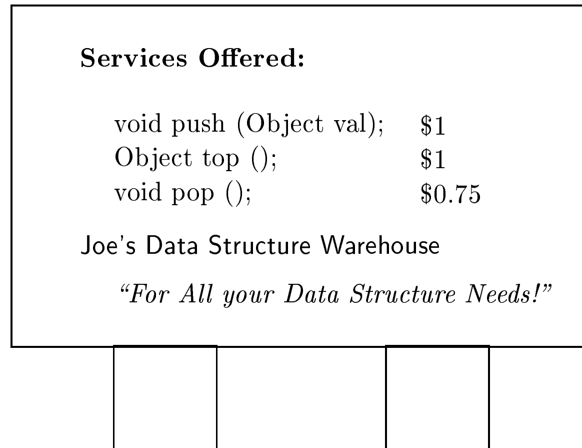
For readers familiar with concepts found in earlier languages, this notion of a unit is the heir to the idea of a *module* in languages such as C or Modula. Later in this chapter we will present a short history of programming language abstractions, and note the debt that ideas of object-oriented programming owe to the earlier work on modules.

The next two levels of abstraction deal with the interactions between two individual objects. Often we speak of objects as proving a *service* to other objects. We build on this intuition by describing communication as an interaction between a *client* and a *server*.



We are not using the term *server* in the technical sense of, say, a web server. Rather, here the term server simply means an object that is providing a service. The two layers of abstraction refer to the two views of this relationship; the view from the client side and the view from the server side.

In a good object-oriented design we can describe and discuss the services that the server provides without reference to any actions that the client may perform in using those services. One can think of this as being like a billboard advertisement:



The billboard describes, for example, the services provided by a data structure, such as a Stack. Often this level of abstraction is represented by an interface, a class-like mechanism that defines behavior without describing an implementation:

```
interface Stack {
    public void push (Object val);
    public Object top () throws EmptyStackException;
    public void pop () throws EmptyStackException;
}
```

Finding the Right Level of Abstraction

In early stages of software development a critical problem is finding the right level of abstraction. A common error is to dwell on the lowest levels, worrying about the implementation details of various key components, rather than striving to ensure that the high level organizational structure promotes a clean separation of concerns.

The programmer (or, in larger projects, the design team) must walk a fine line in trying to identify the right level of abstraction at any one point of time. One does not want to ignore or throw away too much detail about a problem, but also one must not keep so much detail that important issues become obscured.

The next level of abstraction looks at the same boundary but from the server side. This level considers a concrete implementation of the abstract behavior. For example, there are any number of data structures that can be used to satisfy the requirements of a Stack. Concerns at this level deal with the way in which the services are being realized.

```
public class LinkedList implements Stack ... {
    public void pop () throws  EmptyStackException { ... }
    ...
}
```

Finally, the last level of abstraction considers a single task in isolation; that is, a single method. Concerns at this level of abstraction deal with the precise sequence of operations used to perform just this one activity. For example, we might investigate the technique used to perform the removal of the most recent element placed into a stack.

```
public class LinkedList implements Stack ... {
    ...
    public void pop () throws  EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException();
        removeFirst(); // delete first element of list
    }
    ...
}
```

Each level of abstraction is important at some point during software development. In fact, programmers are often called upon to quickly move back and forth between different levels of abstraction. We will see analysis of object-oriented

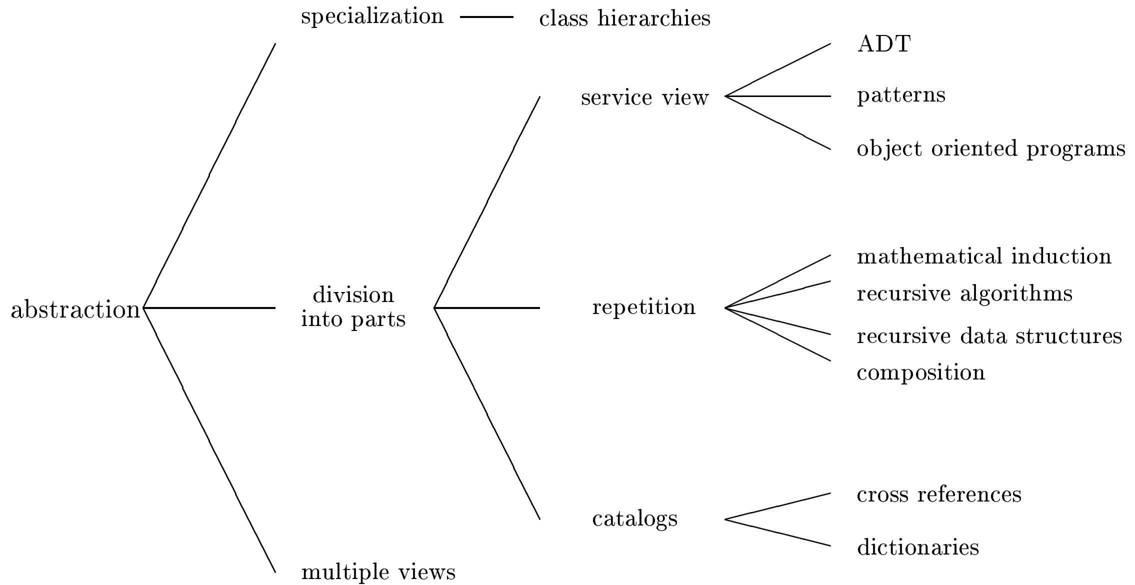


Figure 2.1: Some Techniques for Handling Complexity, with Examples

programs performed at each of these levels of abstraction as we proceed through the book.

2.2 Other Forms of Abstraction

Abstraction is used to help understand a complex system. In a certain sense, abstraction is the imposition of structure on a system. The structure we impose may reflect some real aspects of the system (a car really does have both an engine and a transmission) or it may simply be a mental abstraction we employ to aid in our understanding.

This idea of abstraction can be further subdivided into a variety of different forms (Figure 2.1). A common technique is to divide a layer into constituent parts. This is the approach we used when we described an automobile as being composed of the engine, the transmission, the body and the wheels. The next level of understanding is then achieved by examining each of these parts in turn. This is nothing more than the application of the old maxim *divide and conquer*.

Other times we use different types of abstraction. Another form is the idea of layers of specialization (Figure 2.2). An understanding of an automobile is based, in part, on knowledge that it is a *wheeled vehicle*, which is in turn a *means of transportation*. There is other information we know about wheeled vehicles, and that knowledge is applicable to both an automobile and a bicycle. There is other knowledge we have about various different means of transportation, and that information is also applicable to pack horses as well as bicycles. Object

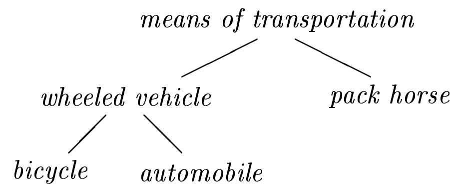


Figure 2.2: Layers of Specialization

Is-a and Has-a Abstraction

The ideas of division into parts and division into specializations represent the two most important forms of abstraction used in object-oriented programming. These are commonly known as *is-a* and *has-a* abstraction.

The idea of division into parts is has-a abstraction. The meaning of this term is easy to understand; a car "has-a" engine, and it "has-a" transmission, and so on.

The concept of specialization is referred to as "is-a" abstraction. Again, the term comes from the English sentences that can be used to illustrate the relationships. A bicycle "is-a" wheeled vehicle, which in turn "is-a" means of transportation.

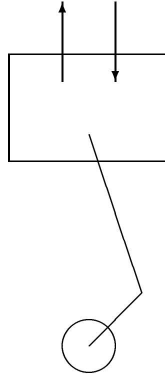
Both is-a and has-a abstractions will reappear in later chapters and be tied to specific programming language features.

oriented languages make extensive use of this form of abstraction.

Yet another form of abstraction is to provide multiple views of the same artifact. Each of the views can emphasize certain detail and suppress others, and thus bring out different features of the same object. A laymans view of a car, for example, is very different from the view required by a mechanic.

2.2.1 Division into Parts

The most common technique people use to help understand complex systems is to combine abstraction with a division into component parts. Our description of an automobile is an example of this. The next level of understanding is then achieved by taking each of the parts, and performing the same sort of analysis at a finer level of detail. A slightly more precise description of an engine, for example, views it as a collection of cylinders, each of which converts an explosion of fuel into a vertical motion, and a crankshaft, which converts the up and down motion of the cylinder into a rotation.



Another example might be organizing information about motion in a human body. At one level we are simply concerned with mechanics, and we consider the body as composed of bone (for rigidity), muscles (for movement), eyes and ears (for sensing), the nervous system (for transferring information) and skin (to bind it all together). At the next level of detail we might ask how the muscles work, and consider issues such as cell structure and chemical actions. But chemical actions are governed by their molecular structure. And to understand molecules we break them into their individual atoms.

Any explanation must be phrased at the right level of abstraction; trying to explain how a person can walk, for example, by understanding the atomic level details is almost certainly difficult, if not impossible.

2.2.2 Encapsulation and Interchangeability

A key step in the creation of large systems is the division into components. Suppose instead of writing software, we are part of a team working to create a new automobile. By separating the automobile into the parts *engine* and *transmission*, it is possible to assign people to work on the two aspects more or less independently of each other. We use the term *encapsulation* to mean that there is a strict division between the inner and the outer view; those members of the team working on the engine need only an abstract (outside, as it were) view of the transmission, while those actually working on the transmission need the more detailed inside view.

An important benefit of encapsulation is that it permits us to consider the possibility of *interchangeability*. When we divide a system into parts, a desirable goal is that the interaction between the parts is kept to a minimum. For example, by encapsulating the behavior of the engine from that of a transmission we permit the ability to exchange one type of engine with another without incurring an undue impact on the other portions of the system.

For these ideas to be applicable to software systems, we need a way to discuss the task that a software component performs, and separate this from the way in which the component fulfills this responsibility.

Catalogs

When the number of components in a system becomes large it is often useful to organize the items by means of a catalog. We use many different forms of catalog in everyday life. Examples include a telephone directory, a dictionary, or an internet search engine. Similarly, there are a variety of different catalogs used in software. One example is a simple list of classes. Another catalog might be the list of methods defined by a class. A reference book that describes the classes found in the Java standard library is a very useful form of catalog. In each of these cases the idea is to provide the user a mechanism to quickly locate a single part (be it class, object, or method) from a larger collection of items.

2.2.3 Interface and Implementation

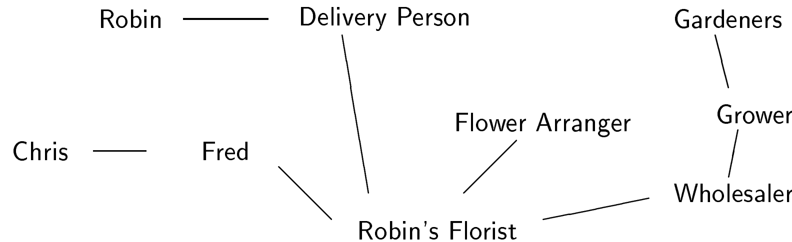
In software we use the terms *interface* and *implementation* to describe the distinction between the *what* aspects of a task, and the *how* features; between the outside view, and the inside view. An interface describes what a system is designed to do. This is the view that *users* of the abstraction must understand. The interface says nothing about how the assigned task is being performed. So to work, an interface is matched with an *implementation* that completes the abstraction. The designers of an engine will deal with the interface to the transmission, while the designers of the transmission must complete an implementation of this interface.

Similarly, a key step along the path to developing complex computer systems will be the division of a task into component parts. These parts can then be developed by different members of a team. Each component will have two faces, the interface that it shows to the outside world, and an implementation that it uses to fulfill the requirements of the interface.

The division between interface and implementation not only makes it easier to understand a design at a high level (since the description of an interface is much simpler than the description of any specific implementation), but also make possible the interchangeability of software components (as I can use any implementation that satisfies the specifications given by the interface).

2.2.4 The Service View

The idea that an interface describes the service provided by a software component without describing the techniques used to implement the service is at the heart of a much more general approach to managing the understanding of complex software systems. It was this sort of abstraction that we emphasized when we described the flower story in Chapter 1. Ultimately in that story a whole community of people became involved in the process of sending flowers:



Each member of the community is providing a service that is used by other members of the group. No member could solve the problem on their own, and it is only by working together that the desired outcome is achieved.

2.2.5 Composition

Composition is another powerful technique used to create complex structures out of simple parts. The idea is to begin with a few primitive forms, and add rules for combining forms to create new forms. The key insight in composition is to permit the combination mechanism to be used both on the new forms as well as the original primitive forms.

A good illustration of this technique is the concept of regular expressions. Regular expressions are a simple technique for describing sets of values, and have been extensively studied by theoretical computer scientists. The description of a regular expression begins by identifying a basic alphabet, for example the letters a, b, c and d. Any single example of the alphabet is a regular expression. We next add a rule that says the composition of two regular expressions is a regular expression. By applying this rule repeatedly we see that any finite string of letters is a regular expression:

$$\mathrm{abaccaba}$$

The next combining rule says that the alternation (represented by the vertical bar |) of two regular expressions is a regular expression. Normally we give this rule a lower precedence that composition, so that the following pattern represents the set of three letter values that begin with ab, and and end with either an a, c or d:

$$\mathrm{aba} \mid \mathrm{abc} \mid \mathrm{abd}$$

Parenthesis can be used for grouping, so that the previous set can also be described as follows:

$$\mathrm{ab(a|c|d)}$$

Finally the $*$ symbol (technically known as the kleene-star) is used to represent the concept "zero or more repetitions". By combining these rules we can describe quite complex sets. For example, the following describes the set of values that begin with a run of a's and b's followed by a single c, or a two character sequence dd, followed by the letter a.

(((a|b)*c)|dd)a

This idea of composition is also basic to type systems. We begin with the primitive types, such as int and boolean. The idea of a class then permits the user to create new types. These new types can include data fields constructed out of previous types, either primitive or user-defined. Since classes can build on previously defined classes, very complex structure can be constructed piece by piece.

```
class  Box { // a box is a new data type
    ...
    private int value; // built out of the existing type int
}
```

Yet another application of the principle of composition is the way that many user interface libraries facilitate the layout of windows. A window is composed from a few simple data types, such as buttons, sliders, and drawing panels. Various different types of layout managers create simple structures. For example, a grid layout defines a rectangular grid of equal sized components, a border layout manger permits the specification of up to five components in the north, south, east, west, and center of a screen. As with regular expressions, the key is that windows can be structured as part of other windows. Imagine, for example, that we want to define a window that has three sliders on the left, a drawing panel in the middle, a bank of sixteen buttons organized four by four on the right, and a text output box running along the top. (We will develop just such an application in Chapter 22. A screen shot is shown in Figure 22.4.) We can do this by laying simple windows inside of more complex windows (Figure 2.3).

Many computer programs can themselves be considered a product of composition, where the method or procedure call is the mechanism of composition. We begin with the primitive statements in the language (assignments and the like). With these we can develop a library of useful functions. Using these functions as new primitives, we can then develop more complex functions. We continue, each layer being built on top of earlier layers, until eventually we have the desired application.

2.2.6 Layers of Specialization

Yet another approach to dealing with complexity is to structure abstraction using layers of specialization. This is sometimes referred to as a *taxonomy.* For example, in biology we divide living things into animals and plants. Living things are then divided into vertebrates and invertebrates. Vertebrates eventually includes mammals, which can be divided into (among other categories) cats and dogs, and so on.

The key difference between this and the earlier abstraction is that the more specialized layers of abstraction (for example, a cat) is indeed a representative

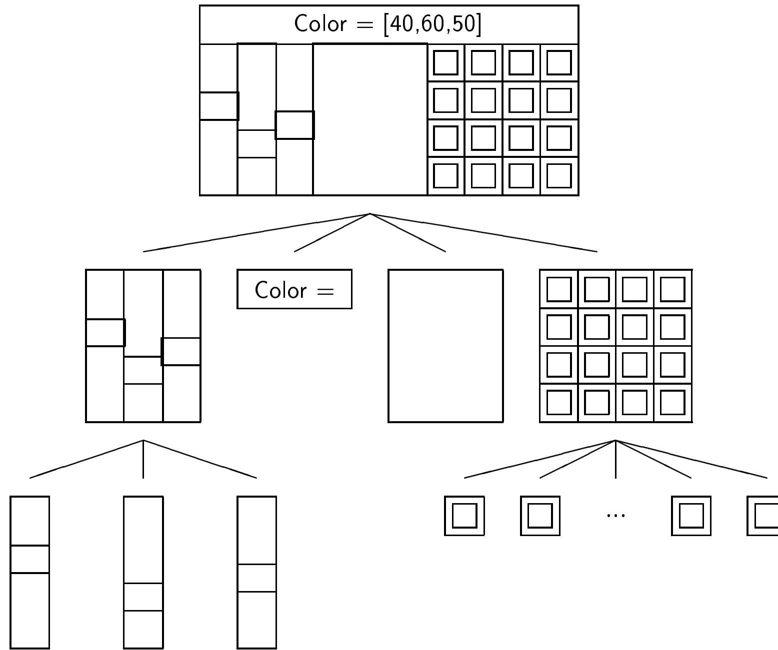


Figure 2.3: Composition in the Creation of User Interfaces

Nonstandard Behavior

Phyl and his friends remind us that there are almost never generalizations without their being exceptions. A platypus (such as phyl) is a mammal that lays eggs. Thus, while we might associate the tidbit of knowledge "gives birth to live young" with the category Mammal, we then need to amend this with the caveat "lays eggs" when we descend to the category Platypus.

Object-oriented languages will also need a mechanism to *override* information inherited from a more general category. We will explore this in more detail once we have developed the idea of class hierarchies.

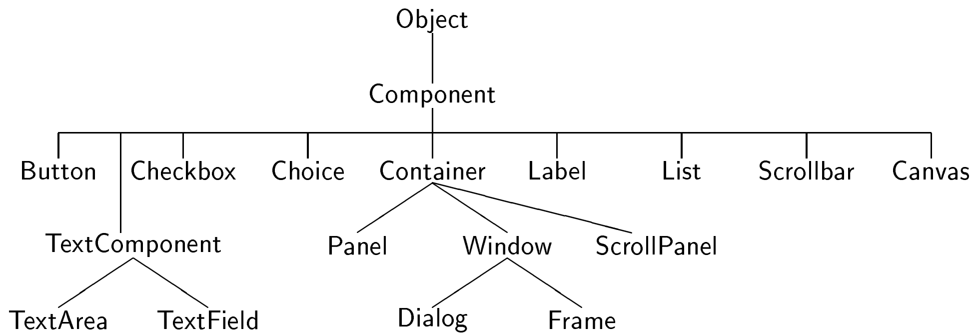


Figure 2.4: The AWT class hierarchy

of the more general layer of abstraction (for example, an animal). This was not true when, in an earlier example, we descended from the characterization of a muscle to the description of different chemical interactions. These two different types of relations are sometimes described using the heuristic keywords "is-a" and "has-a". The first relationship, that of parts to a whole, is a has-a relation, as in the sentence "a car has an engine". In contrast, the specialization relation is described using is-a, as in "a cat is a mammal".

But in practice our reason for using either type of abstraction is the same. The principle of abstraction permits us to suppress some details so that we can more easily characterize a fewer number of features. We can say that mammals are animals that have hair and nurse their young, for example. By associating this fact at a high level of abstraction, we can then apply the information to all more specialized categories, such as cats and dogs.

The same technique is used in object-oriented languages. New interfaces can be formed from existing interfaces. A class can be formed using inheritance from an existing class. In doing so, all the properties (data fields and behavior) we associate with the original class become available to the new class.

In a case study later in this book we will examine the Java AWT (Abstract Windowing Toolkit) library. When a programmer creates a new application using the AWT they declare their main class as a subclass of Frame, which in turn is linked to many other classes in the AWT library (Figure 2.4). A Frame is an special type of application window, but it is also a more specialized type of the general class Window. A Window can hold other graphical objects, and is hence a type of Container. Each level of the hierarchy provides methods used by those below. Even the simplest application will likely use the following:

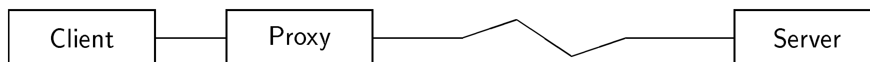
setTitle(String)	inherited from class Frame
setSize(int, int)	inherited from class Component
show()	inherited from class Window
repaint()	inherited from class Component
paint()	inherited from Component, then overridden in the programmers new application class

2.2.7 Patterns

When faced with a new problem, most people will first look to previous problems they have solved that seem to have characteristics in common with the new task. These previous problems can be used as a model, and the new problem attacked in a similar fashion, making changes as necessary to fit the different circumstances.

This insight lies behind the idea of a software *pattern*. A pattern is nothing more than an attempt to document a proven solution to a problem so that future problems can be more easily handled in a similar fashion. In the object-oriented world this idea has been used largely to describe patterns of interaction between the various members of an object community.

A simple example will illustrate this idea of a pattern. Imagine one is developing an application that will operate over a network. That means that part of the application will run on one computer, and part will run on another computer linked by a network connection. Creating the actual connection between the two computers, and transmitting information along this connection, are details that are perhaps not relevant to large portions of the application. One way to structure these relationships is to use a type of pattern termed a *proxy*. The proxy is an intermediary that hides the network connection. Objects can interact with the proxy, and not be aware that any type of network connection is involved at all. When the proxy receives a request for data or action, it bundles the request as a package, transmits the package over the network, recieves the response, unpackages the response and hands it back to the client. In this fashion the client is completely unaware of the details of the network protocol.



Notice how the description of the pattern has captured certain salient points of the interaction (the need to hide the communication protocol from the client) while omitting many other aspects of the interaction (for example, the particular information being communicated between client and server). We will have more to say about patterns later in Chapter 24.

2.3 A Short History of Abstraction Mechanisms*

Each of the abstraction mechanisms we have described in this chapter was the end product of a long process of searching for ways to deal with complexity. Another way to appreciate the role of object-oriented programming is to quickly review the history of mechanisms that computer languages have used to manage complexity. When seen in this perspective, object-oriented techniques are not at all revolutionary, but are rather a natural outcome of a progression from procedures, to modules, to abstract data types, and finally to objects.

2.3.1 Assembly Language

The techniques used to control the first computers were hardly what we would today term a language. Memory locations were described by address (i.e., location 372), not by name or purpose. Operations were similarly described by a numeric operation code. For example, an integer addition might be written as opcode 33, an integer subtraction as opcode 35. The following program might add the contents of location 372 to that of 376, then subtract from the result the value stored in location 377:

```
33   372   376
35   377   376
...  ...   ...
```

One of the earliest abstraction mechanisms was the creation of an assembler; a tool that could take a program written in a more human-friendly form, and translate it into a representation suitable for execution by the machine. The assembler permitted the use of symbolic names. The previous instructions might now be written as follows:

```
ADDI   a,x
SUBI   b,x
...    ...
```

This simple process was the first step in the long process of abstraction. Abstraction allowed the programmer to concentrait more effort on defining the task to be performed, and less on the steps necessary to complete the task.

2.3.2 Procedures

Procedures and functions represent the next improvement in abstraction in programming langauges. Procedures allowed tasks that were executed repeatedly, or executed with only slight variations, to be collected in one place and reused rather than being duplicated several times. In addition, the procedure gave the first possibility for *information hiding.* One programmer could write a procedure, or a set of procedures, that was used by many others. Other programmers

[0]Section headings followed by an asterisk indicate optional material.

```
int datastack[100];
int datatop = 0;

void init()
{
    datatop = 0;
}

void push(int val)
{
    if (datatop < 100)
        datastack [datatop++] = val;
}

int top()
{
    if (datatop > 0)
        return datastack [datatop - 1];
    return 0;
}

int pop()
{
    if (datatop > 0)
        return datastack [--datatop];
    return 0;
}
```

Figure 2.5: – Failure of procedures in information hiding.

did not need to know the exact details of the implementation–they needed only the necessary interface. But procedures were not an answer to all problems. In particular, they were not an effective mechanism for information hiding, and they only partially solved the problem of multiple programmers making use of the same names.

Example–A Stack

To illustrate these problems, we can consider a programmer who must write a set of routines to implement a simple stack. Following good software engineering principles, our programmer first establishes the visible interface to her work–say, a set of four routines: init, push, pop, and top. She then selects some suitable implementation technique. There are many choices here, such as an array with

a top-of-stack pointer, a linked list, and so on. Our intrepid programmer selects from among these choices, then proceeds to code the utilities, as shown in Figure 2.5.

It is easy to see that the data contained in the stack itself cannot be made local to any of the four routines, since they must be shared by all. But if the only choices are local variables or global variables (as they are in early programming languages, such as FORTRAN, or in C prior to the introduction of the `static` modifier), then the stack data must be maintained in global variables. However, if the variables are global, there is no way to limit the accessibility or visibility of these names. For example, if the stack is represented in an array named `datastack`, this fact must be made known to all the other programmers since they may want to create variables using the same name and should be discouraged from doing so. This is true even though these data values are important only to the stack routines and should not have any use outside of these four procedures. Similarly, the names `init`, `push`, `pop`, and `top` are now reserved and cannot be used in other portions of the program for other purposes, even if those sections of code have nothing to do with the stack routines.

2.3.3 Modules

The solution to the problem of global name space congestion was the introduction of the idea of a module. In one sense, modules can be viewed simply as an improved technique for creating and managing collections of names and their associated values. Our stack example is typical, in that there is some information (the interface routines) that we want to be widely and publicly available, whereas there are other data values (the stack data themselves) that we want restricted. Stripped to its barest form, a *module* provides the ability to divide a name space into two parts. The *public* part is accessible outside the module; the *private* part is accessible only within the module. Types, data (variables), and procedures can all be defined in either portion. A module encapsulation of the Stack abstraction is shown in Figure 2.6.

David Parnas, who popularized the notion of modules, described the following two principles for their proper use:

1. One must provide the intended user with all the information needed to use the module correctly, and *nothing more.*

2. One must provide the implementor with all the information needed to complete the module, and *nothing more.*

The philosophy is much like the military doctrine of "need to know"; if you do not need to know some information, you should not have access to it. This explicit and intentional concealment of information is what we have been calling *information hiding.*

Modules solve some, but not all, of the problems of software development. For example, they will permit our programmer to hide the implementation details of her stack, but what if the other users want to have two (or more) stacks?

```
module StackModule;
    export push, pop, top; (* the public interface *)

    var
        (* since data values are not exported, they are hidden *)
        datastack : array [ 1 .. 100 ] of integer;
        datatop : integer;

    procedure push(val : integer)...

    procedure top : integer ...

    procedure pop : integer ...

    begin      (* can perform initialization here *)
        datatop = 0;
    end;
end StackModule.
```

Figure 2.6: A Module for the Stack Abstraction

As a more extreme example, suppose a programmer announces that he has developed a new type of numeric abstraction, called Complex. He has defined the arithmetic operations for complex numbers–addition, subtraction, multiplication, and so on, and has defined routines to convert numbers from conventional to complex. There is just one small problem: only one complex number can be manipulated.

The complex number system would not be useful with this restriction, but this is just the situation in which we find ourselves with simple modules. Modules by themselves provide an effective method of information hiding, but they do not allow us to perform *instantiation*, which is the ability to make multiple copies of the data areas. To handle the problem of instantiation, computer scientists needed to develop a new concept.

2.3.4 Abstract Data Types

The development of the notion of an abstract data type was driven, in part, by two important goals. The first we have identified already. Programmers should be able to define their own new data abstractions that work much like the primitive system provided data types. This includes giving clients the ability to create multiple instances of the data type. But equally important, clients should be able to use these instances knowing only the operations that have been provided, without concern for how those operations were supported.

An *abstract data type* is defined by an abstract specification. The specification for our stack data type might list, for example, the trio of operations push, pop

and top. Matched with the ADT will be one or more different implementations. There might be several different implementation techniques for our stack; for example one using an array and another using a linked list. As long as the programmer restricts themselves to only the abstract specification, any valid implementation should work equally well.

The important advance in the idea of the ADT is to finally separate the notions of interface and implementation. Modules are frequently used as an implementation technique for abstract data types, although we emphasize that modules are an implementation technique and that the abstract data type is a more theoretical concept. The two are related but are not identical. To build an abstract data type, we must be able to:

1. Export a type definition.
2. Make available a set of operations that can be used to manipulate instances of the type.
3. Protect the data associated with the type so that they can be operated on only by the provided routines.
4. Make multiple instances of the type.

As we have defined them, modules serve only as an information-hiding mechanism and thus directly address only list items 2 and 3, although the others can be accommodated via appropriate programming techniques. *Packages*, found in languages such as CLU and Ada, are an attempt to address more directly the issues involved in defining abstract data types.

In a certain sense, an object is simply an abstract data type. People have said, for example, that Smalltalk programmers write the most "structured" of all programs because they cannot write anything but definitions of abstract data types. It is true that an object definition is an abstract data type, but the notions of object-oriented programming build on the ideas of abstract data types and add to them important innovations in code sharing and reusability.

2.3.5 A Service-Centered View

Assembly language and procedures as abstraction mechanisms concentrated the programmers view at the functional level–how a task should be accomplished. The movement towards modules and ADT are indicative of a shift from a function-centered conception of computation to a more data-centered view. Here it is the data values that are important, their structure, representation and manipulation.

Object-oriented programming starts from this data-centered view of the world and takes it one step further. It is not that data abstractions, per se, are important to computation. Rather, an ADT is a useful abstraction because it can be defined in terms of the *service* it offers to the rest of a program. Other types of abstractions can be similarly defined, not in terms of their particular actions or their data values, but in terms of the services they provide.

Assembly Language Functions and Procedures	*Function* Centered View
Modules Abstract Data Types	*Data* Centered View
Object-Oriented Programming	*Service* Centered View

Thus, object-oriented programming represents a third step in this sequence. From function centered, to data centered, and finally to service centered view of how to structure a computer program.

2.3.6 Messages, Inheritance, and Polymorphism

In addition to this service-centered view of computing, object-oriented programming adds several important new ideas to the concept of the abstract data type. Foremost among these is *message passing*. Activity is initiated by a *request* to a specific object, not by the invoking of a function.

Implicit in message passing is the idea that the *interpretation* of a message can vary with different objects. That is, the behavior and response that the message elicit will depend upon the object receiving it. Thus, `push` can mean one thing to a stack, and a very different thing to a mechanical-arm controller. Since names for operations need not be unique, simple and direct forms can be used, leading to more readable and understandable code.

Finally, object-oriented programming adds the mechanisms of *inheritance* and *polymorphism*. Inheritance allows different data types to share the same code, leading to a reduction in code size and an increase in functionality. Polymorphism allows this shared code to be tailored to fit the specific circumstances of individual data types. The emphasis on the independence of individual components permits an incremental development process in which individual software units are designed, programmed, and tested before being combined into a large system.

We will describe all of these ideas in more detail in subsequent chapters.

Chapter Summary

People deal with complex artifacts and situations every day. Thus, while many readers may not yet have created complex computer programs, they nevertheless will have experience in using the tools that computer scientists employ in managing complexity.

- The most basic tool is *abstraction*, the purposeful suppression of detail in order to emphasize a few basic features.

- *Information hiding* describes the part of abstraction in which we intentionally choose to ignore some features so that we can concentrate on others.

- Abstraction is often combined with a division into *components*. For example, we divided the automobile into the engine and the transmission. Components are carefully chosen so that they *encapsulate* certain key features, and interact with other components through a simple and fixed *interface*.

- The division into components means we can divide a large task into smaller problems that can then be worked on more-or-less independently of each other. It is the responsibility of a developer of a component to provide an *implementation* that satisfies the requirements of the interface.

- A point of view that turns out to be very useful in developing complex software system is the concept of a *service provider*. A software component is providing a service to other components with which it interacts. In real life we often characterize members of the communities in which we operate by the services they provide. (A delivery person is charged with transporting flowers from a florist to a recipient). Thus this metaphor allows one to think about a large software system in the same way that we think about situations in our everyday lives.

- Another form of abstraction is a taxonomy, in object-oriented languages more often termed an *inheritance hierarchy*. Here the layers are more detailed representatives of a general category. An example of this type of system is a biological division into categories such as Living Thing-Animal-Mammal-Cat. Each level is a more specialized version of the previous. This division simplifies understanding, since knowledge of more general levels is applicable to many more specific categories. When applied to software this technique also simplifies the creation of new components, since if a new component can be related to an existing category all the functionality of the older category can be used for free. (Thus, for example, by saying that a new component represents a Frame in the Java library we immediately get features such as a menu bar, as well as the ability to move and resize the window).

- Finally, a particular tool that has become popular in recent years is the *pattern*. A pattern is simply a generalized description of a solution to a problem that has been observed to occur in many places and in many forms. The pattern described how the problem can be addressed, and the reasons both for adopting the solution and for considering other alternatives. We will see several different types of patterns throughout this book.

Further Information

In the sidebar on page 33 we mention software catalogs. For the Java programmer a very useful catalog is *The Java Developers Almanac*, by Patrick Chan [Chan 2000].

The concept of *patterns* actually grew out of work in architecture, specifically the work of Christopher Alexander [Alexander 77]. The application of patterns to software is described by Gabriel [Gabriel 96]. The best-known catalog of software Patterns is by Gamma et al [Gamma 1995]. A more recent almanac that collects several hundred design patterns is [Rising 2000].

The criticism of procedures as an abstraction technique, because they fail to provide an adequate mechanism for information hiding, was first stated by William Wulf and Mary Shaw [Wulf 1973] in an analysis of many of the problems surrounding the use of global variables. These arguments were later expanded upon by David Hanson [Hanson 1981].

David Parnas originally described his principles in [Parnas 1972].

An interesting book that deals with the relationship between how people think and the way they form abstractions of the real word is Lakoff [Lakoff 87].

Self Study Questions

1. What is abstraction?
2. Give an example of how abstraction is used in real life.
3. What is information hiding?
4. Give an example of how information hiding is used in real life.
5. What are the layers of abstraction found in an object-oriented program?
6. What do the terms client and server mean when applied to simple object-oriented programs?
7. What is the distinction between an interface and an implementation?
8. How does an emphasis on encapsulation and the identification of interfaces facilitate interchangeability?
9. What are the basic features of composition as a technique for creating complex systems out of simple parts?
10. How does a division based on layers of specialization differ from a division based on separation into parts?
11. What goal motivates the collection of software patterns?
12. What key idea was first realized by the development of procedures as a programming abstraction?
13. What are the basic features of a module?
14. How is an abstract data type different from a module?
15. In what ways is an object similar to an abstract data type? In what ways are they different?

Exercises

1. Consider a relationship in real life, such as the interaction between a customer and a waiter in a resturant. Describe the interaction govering this relationship in terms of an interface for a customer object and a waiter object.

2. Take a relatively complex structure from real life, such as a building. Describe features of the building using the technique of division into parts, followed by a further refinement of each part into a more detailed description. Extend your description to at least three levels of detail.

3. Describe a collection of everyday objects using the technique of layers of specialization.