Chapter 2

Object-Oriented Design

When programmers ask other programmers, "What exactly is this object-oriented programming all about anyway?", the response often tends to emphasize the syntactic features that are introduced in languages, such as C++ or Object Pascal, as opposed to their older, non-object oriented versions, C or Pascal. Thus, discussion usually turns rather quickly to issues such as classes and inheritance, message passing, virtual and static methods, and so on. But such conversations miss the most important point of object-oriented programming, which has nothing to do with syntax.

Working in an object-oriented language (that is, one that supports inheritance, message passing and classes) is neither a necessary nor a sufficient condition for doing object-oriented programming. As we emphasized in the first chapter, the most important aspect of object-oriented programming is a design technique that is driven by the determination and delegation of responsibilities. This technique has been called *responsibility-driven design* [Wirfs-Brock 89b, Wirfs-Brock 90].

2.1 Responsibility Implies Noninterference

As anyone can vouch who can remember being a child, or who has raised children, responsibility is a sword that can cut both ways. When you make an object (be it a child or a software system) responsible for specific actions, you expect a certain behavior, at least when the rules are observed. But just as important, responsibility implies a degree of independence or noninterference. If you tell a child that he or she is responsible for cleaning his or her room, you do not normally proceed to stand over them and watch all the time the activity is being performed – that is not the nature of responsibility. Instead, you expect that, having issued a directive in the correct fashion, the desired outcome will be produced.

Similarly, in the flowers example from Chapter 1, I give the request to deliver flowers to my florist without stopping to think about how my request will be serviced. The florist, having taken on the responsibility for this service, is free to operate without interference on my part.

The difference between conventional programming and object oriented programming is in many ways similar to the difference between actively supervising a child while they perform a task, versus delegating to the child responsibility for the performance of the task. Conventional programming proceeds largely by doing something *to* something else; modifying a record or updating an array, for example. Thus, one portion of code in a software system is frequently intimately tied, by control and data connections, to many other sections of the system. Such dependencies can come about through the use of global variables, through use of pointer values, or simply through inappropriate use of and dependence on implementation details of other portions of code. A responsibility-driven design attempts to cut these links, or at least to reduce them to as unobtrusive a level as possible.

This notion might at first seem no more subtle than the lessons of information hiding and modularity, which are important to programming even in conventional languages. But the philosophy of responsibility-driven design elevates information hiding from a technique to an art. This principle of information hiding becomes vitally important when one moves from programming-in-the-small to programming-in-the-large.

One of the major benefits of object-oriented programming occurs when software subsystems are reused from one project to the next. For example, a simulation manager (such as the one we will develop in Chapter 6) might work both for a simulation of balls on a billiards table and for a simulation of fish in a fish tank. This ability to reuse code implies that the software can have almost no domain-specific components; it must delegate responsibility for domain-specific behavior totally to application-specific portions of the system. The ability to create such reusable code is not one that is easily learned – it requires experience, careful examination of case studies (paradigms, in the original sense of the word), and use of a programming language in which such delegation is natural and easy to express. In subsequent chapters, we will present several such examples.

2.2 Programming in the Small and Programming in the Large

The difference between the development of individual projects and more sizable software systems is often described using the terms "programming in the small" and "programming in the large."

Briefly, programming in the small characterizes projects with the following attributes:

- Code is developed by a single programmer, or perhaps by a very small collection of programmers. A single individual can understand all aspects of a project, from top to bottom, beginning to end.
- The major problem in the software development process is the design and development of algorithms for dealing with the problem at hand.

Programming in the large, on the other hand, characterizes software projects with features such as the following:

- The software system is developed by a large team of programmers. Individuals involved in the specification or design of the system may differ from those involved in the coding of individual components, who may differ as well from those involved in the integration of various components into the final product. No single individual can be said to be responsible for the entire project, or even necessarily understand all aspects of the project.
- The major problem in the software development process is the management of details, and the communication of information between diverse portions of the project.

While the beginning student will usually be better acquainted with the task of programming in the small, aspects of many object-oriented languages are best understood as responses to the problems encountered while programming in the large. Thus, some appreciation of the difficulties involved in developing large systems is a helpful prerequisite to understanding OOP.

2.3 Why Begin with Behavior ?

Why begin the design process with an analysis of behavior? The simple answer is that the behavior of a system is usually understood long before any other aspect.

Earlier software development techniques concentrated on ideas such as characterizing the basic data structures, or the overall structure of function calls, often in the context of the creation of a formal specification of the desired application. But structural elements of the application can be identified only after a considerable amount of analysis of the problem has been performed. Similarly, a formal specification often ended up as a document understood by neither programmer nor client. But *behavior* is something that can be described almost from the moment an idea is first conceived, and (often unlike a formal specification) can be described in terms meaningful to both the programmers and the client.

We will illustrate the application of responsibility-driven design by means of a case study.

2.4 A Case Study in Responsibility-Driven Design

Imagine you are the chief software architect in a major computer firm. Your boss one day walks in to your office with an idea that, it is hoped, will be the next major success in the product line handled by your company. Your assignment is to develop the *Interactive Intelligent Kitchen Helper* (Figure 2.1).

The fundamental cornerstone of the Object-Oriented style of programming The task given to your software team is stated in very few words (written on what appears to be the back of a slightly-used dinner napkin, in handwriting that appears to be that of your boss).

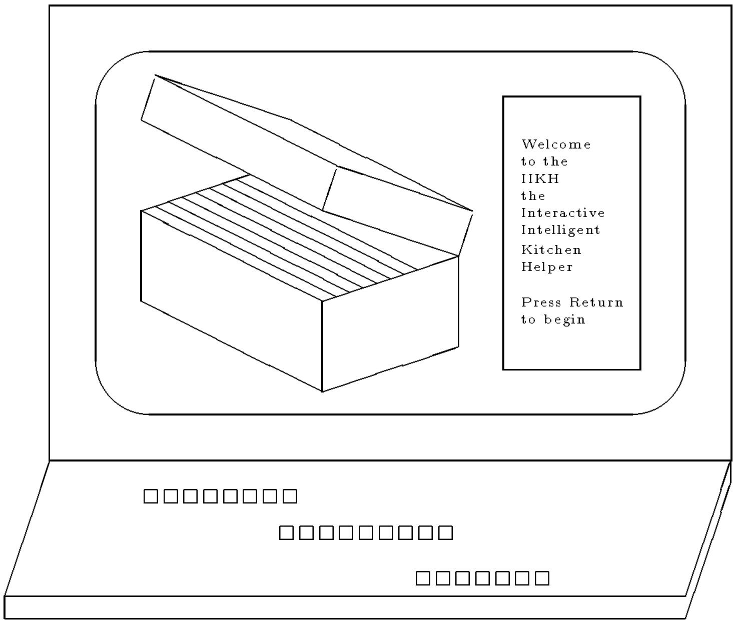


Figure 2.1: View of the Interactive Intelligent Kitchen Helper

2.4.1 The Interactive Intelligent Kitchen Helper

Briefly, the interactive intelligent kitchen helper (IIKH) is a PC-based application that is a modern replacement for the index card case of recipes found in the average household kitchen. But more than simply maintaining a database of recipes, the kitchen helper assists in the planning of meals for an extended period, for example, a week. The user of the IIKH can sit down at a terminal, browse the database of recipes, and interactively create a series of menus. The IIKH will automatically scale the recipes to any desired number of individuals. The user of the IIKH can print out menus for the entire week, for a particular day or for a particular meal. Additionally, the IIKH will print an integrated grocery list of all the items needed to satisfy the recipes for the entire time period, or for any shorter period (a day, a single meal, or a single recipe).

As is usually true with the initial descriptions of most software systems, the specification for the IIKH is highly ambiguous on a number of important points. It is also true that, in all likelihood, the eventual design and development of the software system to support the IIKH will require the efforts of several programmers working together. Thus, the initial goal of the software team must be to clarify the ambiguities in the description, and to start to outline how the project can be divided into components that can then be assigned for development to individual team members.

The fundamental cornerstone of the Object-Oriented style of programming is to characterize software in terms of *behavior*; that is, actions to be performed. We will see this repeated on many levels in our description of the development of the IIKH. Initially, your team will try to characterize, at a very high level of abstraction, the behavior of the entire application. This then leads to the development of a description of the behavior of various software subsystems. Only when all behavior has been identified and described will the software design team proceed to the coding step. In the next several sections we will trace the tasks your software design team will follow in producing this application.

2.4.2 Working Through Scenarios

The first task is to refine the specification. As we have already noted, initial specifications are almost always ambiguous and unclear on anything except the most general points. There are several goals for this step. One objective is to get a better handle on the "look and feel" of the eventual product. This information can then be carried back to the client (in this case, your boss) to see if it is in agreement with their original conception. It is likely, perhaps inevitable, that the specifications for the final application will change during the creation of the software system, and it is important that the design be developed so as to easily accommodate change, and also that potential changes be noted as early as possible. (See the subsequent discussion of "preparing for change"). Equally important, at this point very high level decisions can be made concerning the structure of the eventual software system. In particular, the activities to be performed can be mapped onto components.

2.4.3 Identification of Components

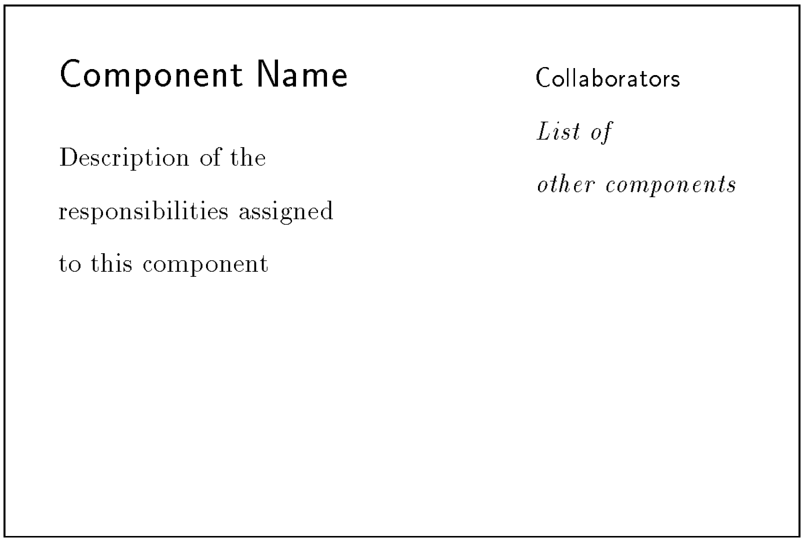
The engineering of a complex physical system, such as a building or an automobile engine, is simplified by dividing the design into smaller units. So, too, is the engineering of software simplified by the identification and development of software components. A **component** is simply an abstract entity that can perform tasks, that is, can fulfill some responsibilities. At this point, it is not necessary to know exactly the eventual representation for a component, or how a component will perform a task. A component may ultimately be turned into a function, into a structure or class, or into a collection of other components (a *pattern*). At this level of development there are just two important characteristics of components:

- A component must have a small well defined set of responsibilities, and
- A component should interact with other components to the minimal extent possible.

We will shortly discuss the reasoning behind the second characteristic. For the moment we will simply be concerned with the identification of responsibilities of components.

2.5 CRC Cards – Recording Responsibility

In order to discover components and their responsibilities, your team proceeds by walking through scenarios. That is, the team acts out the running of your application, just as if it already possessed a working system. Every activity that must take place is identified, and assigned to some component as a responsibility.



As part of this process, it is often useful to represent components using small index cards. Written on the face of the card is the name of the software component, the responsibilities of the component, and the name of other components with which the component must interact. Such cards are sometimes known as CRC cards, which stands for **Component**, **Responsibility** and **Collaborator** [Beck 89]. An index card is associated with each software component. As responsibilities for the component are discovered, they are recorded on the face of the index card.

2.5.1 Give Components a Physical Representation

During the process of working through scenarios, it is useful to assign different CRC cards to different members of the design team. The person holding the card representing a component is charged with the task of recording the responsibilities of the associated software component, as well as with acting as the "surrogate" for the software during the scenario simulation. The design team member describes the activities of the software system, passing "control" to another design team member when the software system requires the services of another component.

An advantage of index cards is that they are widely available, inexpensive, and erasable. This encourages experimentation, since alternative designs can be tried, explored or abandoned with little investment. The physical separation of the cards encourages an intuitive understanding of the importance of the logical separation of the various components, helping to emphasize the concepts of cohesion and coupling (which we will shortly describe). The constraints of an index card are also a good measure of approximate complexity – a component that is expected to perform more tasks than can fit easily in this space is probably too complex, and effort should be expended to find a simpler solution, perhaps by moving some responsibilities elsewhere to divide a task between two or more new components.

2.5.2 The What – Who Cycle

As we noted at the beginning of this section, the identification of components is performed during the process of imagining the execution of a working system. Often this proceeds as a cycle of what/who questions. First, the programming team identifies *what* activity needs to be performed next. This is immediately followed by answering the question *who* is the agent that performs the action. In this manner, designing a software system is much like organizing a collection of people, such as a club. Any activity that is to be performed must be assigned as a responsibility to some component.

There is a popular bumper sticker that asserts that phenomenon can and will spontaneously occur. (The bumper sticker uses a slightly shorter phrase). We know, however, that in real life that this is seldom true. If any action is to take place then there must be an agent assigned to perform the action. Just as in the running of a club any action to be performed must be assigned to some individual, in organizing an object-oriented program all actions must be the responsibility of some component. The secret to good object-oriented design is to first establish for each action an agent that holds the responsibility for the correct performance of the task.

2.5.3 Documentation

It is at this point, as well, that the development of documentation should begin. There are two essential documents that should be part of any software system. These are the user manual, and the system design documentation. Work on both of these can commence even before the first line of code has been written.

The user manual describes the interaction with the system from the user's point of view, and is an excellent means of verifying that the conception of the application held by the development team matches that of the client. Since the decisions made in creating the scenarios will closely match the decisions the user will be required to make in the eventual application, the development of the user manual naturally dovetails with the process of walking through scenarios.

It is at this point in development of an application, before any actual code has been written, that the mindset of the software team is most similar to that of the eventual users. Thus, it is at this point that the developers can most easily anticipate the sort of questions a novice user will need answers to.

The second most important category of documents to produce at this point is the design documentation. The design documentation records the major decisions made during the process of the software design, and should thus also be produced during the process of design when these decisions are fresh in the mind of the creators, and not after the fact when many of the relevant details will have been forgotten. It is often far easier to write a general global description of the software system early in the development. Too soon, the focus of concentration will move to the level of individual components or modules. While it is also important to document the module level, an overly large concern with the details of each module will make it difficult for subsequent software maintainers to initially form a picture of the larger structure.

CRC cards are one aspect of the design documentation, but many other decisions are also important and are not reflected in the CRC cards. Arguments for and against any major design alternatives should be recorded, as well as factors that influenced the final decisions. A log or diary of the project schedule should be maintained. Both the user manual and the design documents are refined and evolve over time in exactly the same manner as the software is evolving and being refined.

2.6 Components and Behavior

To return to the IIKH, your team decides that when the system begins, the user will be presented with an attractive informative window (see Figure 2.1). The responsibility for displaying this window is assigned to a component called the **Greeter**. In some as yet

Greeter	Collaborators
Display Informative Initial Message	Database Manager
Offer User Choice of Options	Plan Manager
Pass Control to either	
Recipe Database Manager or	
Plan Manager for processing	

Figure 2.2: CRC card for the greeter

unspecified manner, (perhaps by pull-down menus, perhaps by button presses, perhaps by pressing a key, perhaps using a pressure sensitive screen), the user can select one of several actions. Initially, you identify just five actions. These are:

1. Casually browse the database of existing recipes, but without reference to any particular meal plan.

2. Add a new recipe to the database.

3. Edit or annotate an existing recipe.

4. Review an existing plan for several meals.

5. Create a new plan of meals.

These activities seem to naturally divide themselves into two groups. The first three are associated with the recipe database, while the latter two are associated with menu plans. As a result, your team next decides to create components corresponding to these two responsibilities. Continuing with the scenario, your team elects to ignore the meal plan management for the moment, and move on to refine the activities of the **Recipe Database** component. A Figure 2.2 shows the initial CRC card representation of the **Greeter**.

Broadly speaking, the responsibility of the recipe database component is to simply maintain a collection of recipes. We have already identified three elements of this task. Namely, the recipe component database must somehow facilitate the user browsing and editing from the library of existing recipes, and must also permit the inclusion of new recipes into the database.

2.6.1 Postponing Decisions

There are a number of decisions that must eventually be made concerning how to best let the user browse the database. For example, should the user be presented first with a list of categories, such as "soups", "salads", "main meals", and "desserts"? Alternatively should the user be able to describe keywords to narrow a search, such as providing a list of ingredients and seeing all the recipes that contain those items ("Almonds, Strawberries, Cheese"), or a list of previously inserted keywords ("Bob's favorite cake")? Should scroll bars be used, or simulated thumb holes in a virtual book? While these are fun to think about, the important point to realize is that such decisions do not need to be made at this point (see "Preparing for Change"). Since these affect only a single component, and do not impact the functioning of any other system, all that is necessary in order to continue the scenario is to simply assert that by some means the user can select a specific recipe.

2.6.2 Preparing for Change

It has been said that all that is constant in life is the inevitability of uncertainty and change. The same is true of software. No matter how carefully one tries to develop the initial specification and design of a software system, it is almost certain that changes in the user's needs or requirements will sometime, during the life of the system, force changes to be made in software. Programmers and software designers need to anticipate this, and plan accordingly.

- The primary objective is that changes should impact as few components as possible. Even major changes in the appearance or functioning of an application should be possible with changes to only one or two sections of code.

- Try to predict the most likely sources of change, and isolate the effects of such changes to as few software components as possible. The most likely sources of change are features such as interfaces, communication formats, and output formats.

- Try to isolate and reduce the dependency of software on hardware. For example, the interface for recipe browsing in our application may depend in part on the hardware on which the system is running. Future releases may be ported to different platforms. A good design will anticipate this change.

- Reducing coupling between software components will reduce the dependence of one upon another, and increase the likelihood that one can be changed with minimal effect on the other.
- Maintain in the design documentation careful records of the design process and the discussions surrounding all major decisions. It is almost certain that the individuals responsible for maintaining the software and designing future releases will be, if not entirely different, then at least partially different from the team of individuals producing the initial release. The design documentation will assist future teams to know the important factors behind a decision, and help them avoid spending time in discussions of issues that have already been resolved.

2.6.3 Continuing the Scenario

Each recipe will be identified with a specific recipe component. Once a recipe is selected, control is passed to the associated recipe object. A recipe must contain certain information. For example, a recipe consists of a set of ingredients, and a description of the steps needed to transform the ingredients into the final product. In our scenario, the recipe component must also perform certain other activities. For example, the recipe component will display the recipe interactively on the terminal screen. The user may be given the ability to annotate or change the recipe, either the list of ingredients or the instruction portion. Alternatively the user may request a printed copy of the recipe. All of these actions are the responsibility of the **Recipe** component. (For the moment, we will continue to describe the **Recipe** in the singular form. During design we can think of this as a prototypical recipe, that stands in place of a multitude of actual recipes. We will return to a discussion of singular versus multiple components in a later section.)

Having pursued the actions that will result as consequence of the user wishing to browse the database, we return to the recipe database manager and now pretend the user indicated a desire to add a new recipe. The database manager somehow decides what category to place the new recipe (again, the details concerning how this is done are unimportant for our development at this point), requests the name of the new recipe, then creates a new recipe component, permitting the user to edit this new blank entry. Thus, the responsibilities needed to perform this new task are a subset of those we already identified, in permitting users to edit existing recipes.

Having explored the browsing and creation of new recipes, we return to the **Greeter** and investigate the development of daily menu plans. The **Plan Manager** is charged with this task. In some way (again, the details are unimportant here) the user can save existing plans. Thus, the plan manager can either be started by retrieving an already developed plan, or by creating a new plan. In the latter case, the user is prompted for a list of dates for which the plan is being developed. Each date is associated with a separate **Date** component. The user can select a specific date for further investigation, in which case control is passed to the corresponding **Date** component. Another activity of the plan manager includes printing out

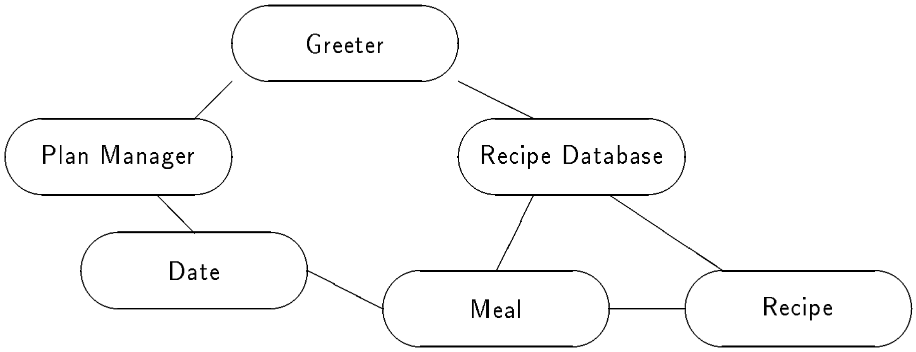


Figure 2.3: Communication between the six components in the IIKH

the recipes for the entire planning period. Finally, the user can instruct the plan manager to produce a grocery list for the entire period.

The **Date** component maintains a collection of meals, as well as any other annotations provided by the user (indications of birthday celebrations, anniversaries, reminders, and so on). The **Date** component prints information on the display concerning the specified date. By some means (again unspecified), the user can indicate a desire to print all the information concerning a specific date, or can choose to explore in more detail a specific meal. In the latter case, control is passed to a **Meal** component.

The meal component maintains a collection of augmented recipes, where the augmentation refers to the fact that the user may indicate a desire to double, triple, or otherwise increase a recipe. The meal component displays information about the meal. The user can add or remove recipes from the meal, or can instruct that information about the meal be printed. In order to discover new recipes, the user must be permitted at this point to browse the recipe database. Thus, the meal component must interact with the recipe database component. The design team will continue in this fashion, investigating every possible scenario. The major category of scenarios we have not developed here involve exceptional cases. For example, what happens if a user selects a number of keywords for a recipe, and no matching recipe is found? How can the user cancel an activity, such as entering a new recipe, if after they begin they decide not to proceed? Each possibility must be explored, and the responsibilities for handling the situation assigned to one or more components.

Having walked through the various scenarios, your software design team eventually decides that all activities can be adequately handled by 6 components (Figure 2.3). The **Greeter** needs to communicate only with the **Plan Manager** and the **Recipe Database** component. The **Plan Manager** needs, in turn, only to communicate with the **Date** component, and the **Date** agent only with the **Meal** component. The **Meal** component communicates with the recipe manager and, through this agent, with individual recipes.

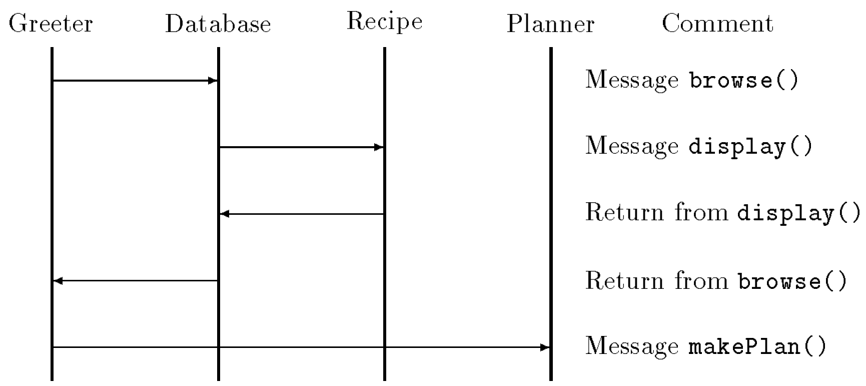


Figure 2.4: An Example Interaction Diagram

2.6.4 Interaction Diagrams

While a description such as that shown in Figure 2.3 may describe the static relationships between components, it is not very good for describing their dynamic interactions during the execution of a scenario. A better tool for this purpose is known as an *interaction diagram*. Figure 2.4 shows the beginning of an interaction diagram for the interactive kitchen helper. In the diagram, time moves forward from the top of diagram to the bottom. Each component is represented by a labelled vertical line. A component sending a message to another component is represented by a horozontal arrow from one line to another. Similarly, a component returning control and perhaps a result value back to the caller is represented by an arrow. (Some authors use two different forms of arrow, such as a solid line to represent message passing and a dashed line to represent returning control.) A commentary on the right-hand side explains more fully the interaction taking place.

By providing a time axis, the interaction diagram is able to better describe the sequencing of events occuring during a scenario. For this reason, interaction diagrams can be a useful documentation tool for complex software systems.

2.7 Software Components

In this section we will explore in more detail the concept of a software component. As is true of all but the most trivial ideas, there are many different aspects to this seemingly simple concept.

2.7.1 Behavior and State

We have already seen how components are characterized by their behavior, that is, by what they can do. But components may also hold within them certain information. Let us take as our prototypical component a **Recipe** structure from the IIKH. One way to view a component is as a pair consisting of *behavior* and *state*.

- The *behavior* of a component is the set of actions the component can perform. The complete description of all the behavior for a component is also sometimes called the *protocol* for the component. For the **Recipe** component this will include activities such as editing the preparation instructions for the recipe, displaying the recipe on an interactive terminal screen, or printing a paper copy of the recipe.

- The *state* of a component represents all the information held within a component. For our **Recipe** component the state will include the table of ingredients and the list of preparation instructions. Notice that the state is not static, and can change over time. For example, by using the ability to edit a recipe (a behavior) the user can make changes to the preparation instructions (part of the state).

It is not necessary that all components maintain state information. For example, it is possible the **Greeter** component will not have any state since it does not need to remember any information during the course of execution. But most components will consist of a combination of behavior and state.

2.7.2 Instances and Classes

The separation of the concepts of state and behavior permits us to clarify a point we avoided in our earlier discussion. Note that in the real application there will probably be many different recipes. An important issue to note, however, is that all of these recipes will *perform* in the same manner. That is, the behavior of each of the recipes is the same, it is only the state, the individual lists of ingredients and instructions for preparation, that is different between individual recipes. In the early stages of development our interest is in characterizing the behavior common to all recipes, and the details particular to any one recipe are unimportant.

The term *class* is used to describe a set of objects with similar behavior. We will see in later chapters, that the idea of a class is also used as a syntactic mechanism in almost all object-oriented languages. An individual representative of a class is known as an *instance*. It is important to note that behavior is associated with a class, and not with an individual. That is, all instances of a class will respond to the same instructions and perform in a similar manner. On the other hand, state is a property of an individual. We see this in the various different instances of the class **Recipe**. They can all perform the same actions (editing, displaying, printing) but use different data values. We will investigate the class concept in more detail in Chapter 3.

2.7.3 Coupling and Cohesion

Two important concepts to understand in conjunction with the design of software components are the ideas of coupling and cohesion. Cohesion is a description of the degree to which the responsibilities of a single component seem to form a meaningful unit. High cohesion is achieved by associating, in a single component, tasks that are related in some manner. Probably the most frequent way in which tasks are related is through the necessity to access a common data area. This is the overriding theme that joins, for example, the various responsibilities of the **Recipe** component.

Coupling, on the other hand, describes the relationship between software components. In general, it is desirable to reduce the amount of coupling as much as possible, since connections between software components inhibits ease of development, modification, or reuse.

In particular, coupling is increased when one software component must access data values, the state, held by another component. Such situations should almost always be avoided, in favor of moving a task to be performed into the list of responsibilities of the component that holds the necessary data. For example, one might conceivably first assign the responsibility for the task "edit a recipe" to the **Recipe Database** component, since it is during the course of performing tasks associated with this component that the need to edit a recipe first occurs. But if we did so, the recipe database agent would then need the ability to directly manipulate the state (the internal data values representing the list of ingredients and the preparation instructions) of an individual recipe. It is better to avoid this tight connection by moving the responsibility for editing to the recipe itself.

We will discuss coupling and cohesion, and other software engineering issues, in more detail in Chapter 17.

2.7.4 Interface and Implementation – Parnas' Principles

The emphasis on characterizing a software component by its behavior has one extremely important consequence. It is possible for one programmer to know how to *use* a component developed by another programmer, without needing to know how the component is *implemented*. For example, suppose each of the six components in the IIKH is assigned to a different programmer. The programmer developing the **Meal** component needs to be able to allow the IIKH user to browse the database of recipes, and select a single recipe for inclusion in the meal. To do this, the **Meal** component can simply invoke the **browse** behavior associated with the **Recipe Database** component, which is defined so as to return an individual **Recipe**. This description is valid regardless of the particular implementation used by the **Recipe Database** component to perform the actual browsing action.

The purposeful omission of implementation details behind a simple interface is known as *information hiding*. We say the component *encapsulates* the behavior, showing only how the component can be used, not the detailed actions it performs. This naturally leads to two different views of a software system. The interface view is the face seen by other programmers. It describes *what* a software component can perform. The implementation view, on the other hand, is the face seen by the programmer working on a particular component. It describes *how* a component goes about completing a task.

The separation of interface and implementation is perhaps *the* most important concept in software engineering. Yet it is also, often, difficult for students to understand, or to motivate. The concept of information hiding is largely meaningful only in the context of multi-person programming projects. In such efforts, the limiting factor is often not the amount of coding involved in a project, but the amount of communication required between the various programmers and between their respective software systems. As we will describe shortly, software components are often developed in parallel by different programmers, and in isolation from each other.

Increasingly, there is also an emphasis on the reuse of general purpose software components in multiple projects. For both of these to be successful, there must be minimal and well understood interconnections between the various portions of the system. As we noted in the last chapter, these ideas were captured by computer scientist David Parnas in a pair of rules, which are known as **Parnas's Principles:**

- The developer of a software component must provide the intended user with all the information needed to make effective use of the services provided by the component, and should provide **no** other information.
- The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provided with **no** other information.

A consequence of the separation of interface from implementation is the fact that a programmer can experiment with several different implementations of the same structure without impacting other software components.

2.8 Formalize the Interface

We continue with the description of the development of the IIKH. In the course of the next several steps the descriptions of the components will slowly be refined. The first step in this process is to formalize the patterns and channels of communication.

A decision should be made as to the general structure that will be used to implement each component. A component with only one behavior and no internal state may be made into a function. An example might be a component that simply takes a string of text and translates all capital letters to lower case. Components with many tasks are probably more easily implemented as classes. Names are given to each of the responsibilities identified on the CRC card for each component. These will eventually be mapped onto function or procedure names. Along with the names, the types of any arguments to be passed to the function are identified. Next, the information maintained within the component itself should be described. All information must be accounted for. If a component requires some data to perform a specific task, the source of the data, either through argument, global value, or maintained internally by the component, must be clearly identified.

2.8.1 Coming up with Names

Careful thought should be given to the names associated with various activities. Shakespeare claims that a change in the name of an object does not alter the characteristics of the entity being denoted,[1] but it is certainly not the case that all names will conjure up the same mental images in the listener. As government bureaucrats have long known, obscure and idiomatic names can make even the simplest operation sound intimidating. Thus, the selection of useful names is an extremely important task, as names create the vocabulary with which the eventual design will be formulated. Names should be internally consistent, meaningful, preferably short, and evocative in the context of the problem at hand. Often a considerable amount of time is spent finding just the right set of terms to describe the tasks being performed and the objects being manipulated. Far from being a barren and useless exercise, the proper selection of names early in the design process greatly simplifies and facilitates later steps.

The following general guidelines have been suggested for selecting names: [Keller 90]

- Use pronounceable names. As a rule of thumb, if you cannot read a name out loud, it is not a good name.
- Use capitalization (or underscores) to mark the beginning of a new word within a name, such as "CardReader" or "Card_reader", rather than cardreader.
- Examine abbreviations carefully. An abbreviation that is clear to one person may be confusing to the next. Is a "TermProcess" a terminal process, something that terminates processes, or a process associated with a terminal?
- Avoid names with several interpretations. Does the **empty** function tell whether something is empty, or empty the values from the object?
- Avoid digits within a name. Digits are easy to misread (0 and O, 1 and l, 2 and Z, 5 and S, and so on).
- Name functions and variables that yield boolean values so as to describe clearly the interpretation of a true or false value. For example, "PrinterIsReady" clearly indicates that a true value means the printer is working, whereas "PrinterStatus" is much less precise.

[1] "What's in a name? That which we call a rose, by any other name would smell as sweet; So Romeo would, were he not Romeo call'd, retain that dear perfection which he owes without that title." William Shakespeare, *Romeo and Juliet*, Act II, Scene 2.

- Extra care should be taken in the selection of names for operations that are costly and infrequently used. By doing so, errors caused by using the wrong function can be avoided.

Once names have been developed for each activity, the CRC cards for each component are redrawn, with the name and formal arguments of the function used to elicit each behavior identified. An example CRC card for the Date is shown below. What is not yet specified is how each component will perform the associated tasks.

Once more, scenarios or role playing should be performed at a more detailed level in order to ensure that all activities are accounted for, and that all necessary information is maintained and made available to the responsible components.

2.9 Design the Representation for Components

At this point, if not before, the design team can be divided into different groups, each group responsible for one or more software components. The task now is to transform the description of a component into an implementation of a software subsystem. The major portion of this process is to design the data structures that will be used by each subsystem to maintain the state information required to fulfill the assigned responsibilities.

It is here that the classic data structures studied as part of the basic core of computer science will come into play. The selection of data structures is an important task, central to the software design process. Once the data structures have been chosen, the code used by a component in the fulfillment of a responsibility is often almost self-evident. But data structures must be carefully matched to the task at hand. A wrong choice of data structure can result in complex and inefficient programs, while an intelligent choice can result in just the opposite.

It is also at this point that descriptions of behavior must be transformed into algorithms. These descriptions should then be matched against the expectations of each component listed as a collaborator, in order to ensure that expectations are fulfilled and necessary data items are available to carry out each process.

2.10 Implementing Components

Having laid out the design of each software subsystem, the next step is to implement the behavior desired for each component. If the previous steps have been correctly addressed, then each responsibility or behavior will be characterized by a short description. The task at this step is to implement in computer language the desired activities. In a later section we will describe some of the more common heuristics that are used in this process.

If they have not been determined earlier (say, as part of the specification of the system), then it is at this point that decisions can be made on issues that are entirely self-contained within a single component. An example decision we saw in our example problem was the decision on how best to let the user browse the database of recipes.

Note that, as multiperson programming projects become the norm, it is increasingly rare that any one programmer will work on all aspects of a system. More often, the skills a programmer will need to master will be the ability to understand how one section of code fits into a larger framework, and the capability to work well with other members of a team.

Often, in the implementation of one component it will become clear that certain information or actions might be assigned to yet another component that will act "behind the scene", with little or no visibility to users of the software abstraction. Such components are sometimes known as *facilitators*. We will see examples of facilitators in some of the later case studies.

An important part of analysis and coding at this point is to characterize and document the necessary preconditions a software component requires in order to complete a task, and to verify that the software component will indeed perform correctly when presented with legal input values. This is establishing the correctness aspect of the algorithms used in the implementation of a component.

2.11 Integration of Components

Once software subsystems have been individually designed and tested, they can be integrated into the final product. Often this is not a single step, but part of a larger process. Starting from a simple base, elements are slowly added to the system and tested, using *stubs*, simple dummy routines with no behavior or with very limited behavior, for the as-yet unimplemented parts.

For example, in the development of the IIKH, it would be reasonable to start integration with the **Greeter** component. To test the **Greeter** component in isolation, stubs are written for the recipe database manager and the daily meal plan manager. These stubs need not do any more than print an informative message and return. With these, the component development team can test various aspects of the greeter system (for example, that button presses elicit the correct response). Testing of an individual component is often referred to as *unit testing*.

Next, one or the other of the stubs can be replaced by more complete code. For example, the team might decide to replace the stub for the recipe database component with the actual system, maintaining the stub for the other portion. Further testing can be performed, until it appears the system is performing as desired. (This is sometimes referred to as *integration testing*).

The application is finally complete when all stubs have been replaced with working components. The ability to test components in isolation is greatly facilitated by the conscious design goal of reducing connections between components, since this reduces the need for extensive stubbing.

During integration it is not uncommon for an error to be manifest in one software system, and yet to be caused by a coding mistake in another system. Thus, testing during integration often involves the discovery of errors, which then result in changes to some of the components. Following these changes the components should be once again tested in isolation before attempting to reintegrate the software, once more, into the larger system. Reexecuting previously developed test cases following a change to a software component is sometimes referred to as *regression testing.*

2.12 Maintenance and Evolution

It is tempting to think that once a working version of an application has been delivered that the task of the software development team is finished. This is, unfortunately, almost never true. The term *software maintenance* is used to describe activities that take place subsequent to the delivery of the initial working version of a software system. A wide variety of activities can be placed into this category.

- Errors, or *bugs*, can be discovered in the delivered product. These must be corrected, either in *patches* to existing releases or in subsequent releases.
- Requirements may change, say as a result of government regulations, or standardization among similar products.
- Hardware may change. The system may be moved to different platforms. Input devices, such as a pen-based system or a pressure sensitive touch screen, may become available that were not previously present. Output technology may change, for example from a textual based system to a graphical window based arrangement.
- Users expectations may change. Users may expect greater functionality, lower cost, easier use. This can perhaps occur as a result of competition with similar products.
- Better documentation may be requested by users.

A good design recognizes the inevitability of change, and plans an accommodation for these activities from the very beginning.

Exercises

1. Describe the responsibilities of an organization that comprises at least six types of individuals. Examples of such organizations are a school (students, teachers, principal, janitor) a business (secretary, president, worker) and a club (president, vice-president, member). For each type of individual, describe the responsibilities and the collaborators.
2. Describe a scenario for the organization you provided in Exercise 1 using an interaction diagram.

3. Take a common game, such as the card games solitaire or twenty-one. Describe a software system that will interact with the user to play the game. Example components include the deck, and the discard pile.

4. Describe the software system to control an ATM (Automated Teller Machine). Give interaction diagrams for various scenarios that describe the most common uses of the machine.