Chapter 3

Object-Oriented Design

A superficial description of the distinction between an object-oriented language, such as Java, and a conventional programming language, such as Pascal, might concentrate on syntactic differences. In this area discussion would center on topics such as classes, inheritance, message passing, and methods. But such an analysis 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 sufficient condition for doing object-oriented programming. As we emphasized in Chapter 1, an object-oriented program is like a community of interacting individuals, each having assigned responsibilities, working together towards the attainment of a common goal. As in real life, a major aspect in the design of such a community is determining the specific responsibilities for each member. To this end, practicioners of object-oriented design have developed a design technique driven by the specification and delegation of responsibilities. This technique has been called *responsibility-driven design* [Wirfs-Brock 1989b, Wirfs-Brock 1990].

3.1 Responsibility Implies Noninterference

As anyone can attest who can remember being a child, or who has raised children, responsibility is a sword that cuts 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 she is responsible for cleaning her room, you do not normally stand over her and watch while that task 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. Flora, 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 the difference between actively supervising a child while she performs a task, and delegating to the child responsibility for that performance. 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 often 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 make them as unobtrusive as possible.

This notion might at first seem no more subtle than the notions of information hiding and modularity, which are important to programming even in conventional languages. But 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 system might work for both a simulation of balls on a billiards table and 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 totally delegate responsibility for domain-specific behavior 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.

3.2 Programming in the Small and in the Large

The difference between the development of individual projects and of more sizable software systems is often described as programming in the small versus programming in the large. 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 in the final product. No single individual can be considered responsible for the entire project, or even necessarily understands 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 acquainted with 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.

3.3 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 sequence of function calls, often within 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 problem analysis. 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 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 (RDD) with a case study.

3.4 A Case Study in RDD

Imagine you are the chief software architect in a major computer firm. One day your boss walks into your office with an idea that, it is hoped, will be the next major success in your product line. Your assignment is to develop the *Interactive Intelligent Kitchen Helper* (Figure 3.1). 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 your boss's.

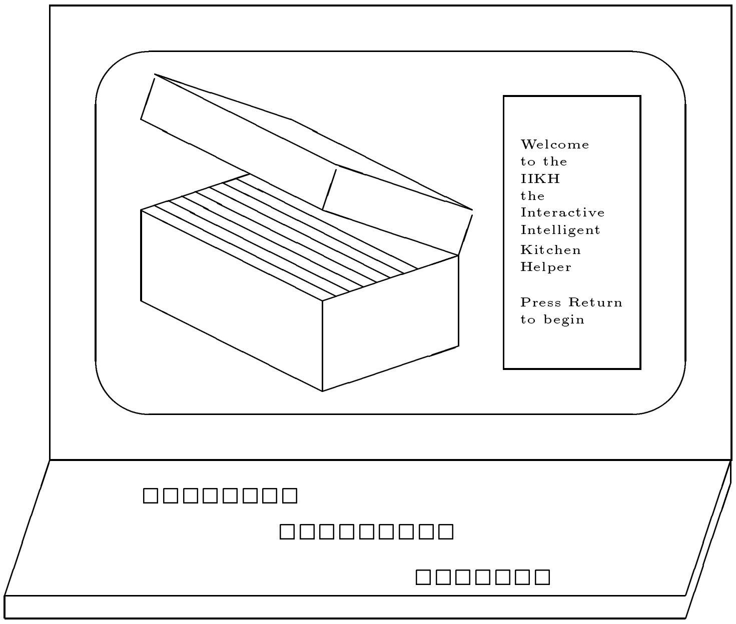


Figure 3.1: – View of the Interactive Intelligent Kitchen Helper.

3.4.1 The Interactive Intelligent Kitchen Helper

Briefly, the Interactive Intelligent Kitchen Helper (IIKH) is a PC-based application that will replace the index-card system of recipes found in the average kitchen. But more than simply maintaining a database of recipes, the kitchen helper assists in the planning of meals for an extended period, say 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 number of servings and will print out menus for the entire week, for a particular day, or for a particular meal. And it will print an integrated grocery list of all the items needed for the recipes for the entire period.

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 outline how the project can be divided into components to be assigned for development to individual team members.

The fundamental cornerstone of object-oriented programming is to characterize software in terms of *behavior*; that is, actions to be performed. We will see this repeated on many levels in the development of the IIKH. Initially, the team will try to characterize, at a very high level of abstraction, the behavior of the entire application. This then leads to 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 the software design team will perform in producing this application.

3.4.2 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 the 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 to easily accommodate change and that potential changes be noted as early as possible. (See Section 3.6.2; "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.

3.4.3 Identification of Components

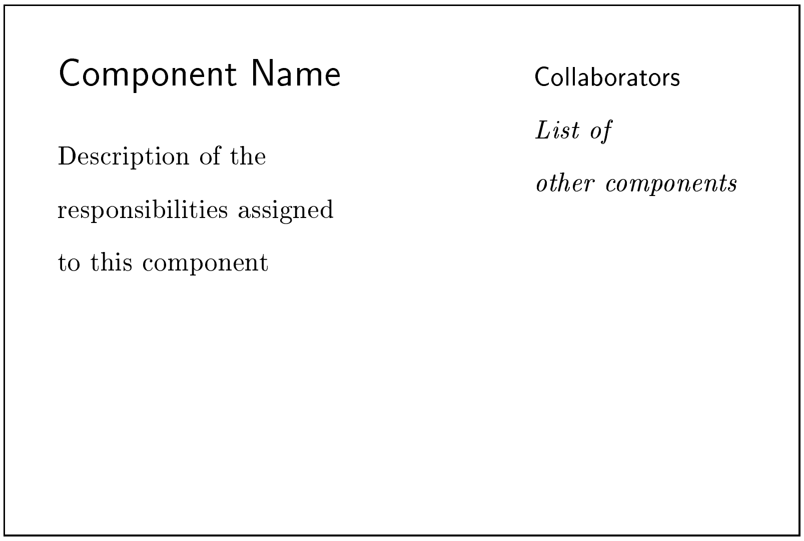
The engineering of a complex physical system, such as a building or an automobile engine, is simplified by dividing the design into smaller units. So, too, the engineering of software is simplified by the identification and development of software components. A *component* is simply an abstract entity that can perform tasks–that is, 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, a structure or class, or a collection of other components (a *pattern*). At this level of development there are just two important characteristics:

- A component must have a small well-defined set of responsibilities.
- 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 are simply concerned with the identification of component responsibilities.

3.5 CRC Cards–Recording Responsibility

In order to discover components and their responsibilities, the programming team walks through scenarios. That is, the team acts out the running of the 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 names of other components with which the component must interact. Such cards are sometimes known as CRC (Component, Responsibility, Collaborator) cards [Beck 1989, Bellin 97], and are associated with each software component. As responsibilities for the component are discovered, they are recorded on the face of the CRC card.

3.5.1 Give Components a Physical Representation

While working through scenarios, it is useful to assign CRC cards to different members of the design team. The member holding the card representing a component records the responsibilities of the associated software component, and acts as the "surrogate" for the software during the scenario simulation. He or she describes the activities of the software system, passing "control" to another member when the software system requires the services of another component.

An advantage of CRC 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 cohesion and coupling (which we will describe shortly). 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 the team should find a simpler solution, perhaps by moving some responsibilities elsewhere to divide a task between two or more new components.

3.5.2 The What/Who Cycle

As we noted at the beginning of this discussion, the identification of components takes place 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 of *who* 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.

We know, from real life, that if any action is to take place, there must be an agent assigned to perform it. 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 an agent for each action.

3.5.3 Documentation

At this point the development of documentation should begin. Two documents should be essential parts of any software system: 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; it is an excellent means of verifying that the development team's conception of the application matches the client's. 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.

Before any actual code has been written, 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 to which a novice user will need answers.

The second essential document is the design documentation. The design documentation records the major decisions made during software design, and should thus be produced when these decisions are fresh in the minds 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 will move to the level of individual components or modules. While it is also important to document the module level, too much concern with the details of each module will make it difficult for subsequent software maintainers to form an initial picture of the larger structure.

CRC cards are one aspect of the design documentation, but many other important decisions are not reflected in them. 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 way the software is refined and evolves.

3.6 Components and Behavior

To return to the IIKH, the team decides that when the system begins, the user will be presented with an attractive informative window (see Figure 3.1). The responsibility for displaying this window is assigned to a component called the Greeter. In some as yet unspecified manner (perhaps by pull-down menus, button or key presses, or use of a pressure-sensitive screen), the user can select one of several actions. Initially, the team identifies just five actions:

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 divide themselves naturally into two groups. The first three are associated with the recipe database; the latter two are associated with menu plans. As a result, the team next decides to create components corresponding to these two responsibilities. Continuing with the scenario, the team elects to ignore the meal plan management for the moment and move on to refine the activities of the Recipe Database component. Figure 3.2 shows the initial CRC card representation of the Greeter.

Broadly speaking, the responsibility of the recipe database component is simply to maintain a collection of recipes. We have already identified three elements of this task: The recipe component database must facilitate browsing the library of existing recipes, editing the recipes, and including new recipes in the database.

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

Figure 3.2: – CRC card for the Greeter.

3.6.1 Postponing Decisions

There are a number of decisions that must eventually be made concerning how best to let the user browse the database. For example, should the user first be presented 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, perhaps by providing a list of ingredients, and then see 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? These are fun to think about, but the important point is that such decisions do not need to be made at this point (see next section). Since they affect only a single component, and do not affect the functioning of any other system, all that is necessary to continue the scenario is to assert that by some means the user can select a specific recipe.

3.6.2 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 the software. Programmers and software designers need to anticipate this and plan accordingly.

- The primary objective is that changes should affect as few components as possible. Even major changes in the appearance or functioning of an application should be possible with alterations 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 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.

- In the design documentation maintain 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 at least partially different from the team producing the initial release. The design documentation will allow future teams to know the important factors behind a decision and help them avoid spending time discussing issues that have already been resolved.

3.6.3 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. Basically, it consists of a list of ingredients and the steps needed to transform the ingredients into the final product. In our scenario, the recipe component must also perform other activities. For example, it will display the recipe interactively on the terminal screen. The user may be given the ability to annotate or change 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 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 later return to a discussion of singular versus multiple components.)

Having outlined the actions that must take place to permit the user to browse the database, we return to the recipe database manager and pretend the user has indicated a desire to add a new recipe. The database manager somehow decides in which category to place the new recipe (again, the details of how this is done are unimportant for our development at this point), requests the name of the new recipe, and then creates a new recipe component, permitting the user to edit this new blank entry. Thus, the responsibilities of performing 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, which is the Plan Manager's 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 the plan. 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 is printing out the recipes for the planning period. Finally, the user can instruct the Plan Manager to produce a grocery list for the period.

The Date component maintains a collection of meals as well as any other annotations provided by the user (birthday celebrations, anniversaries, reminders, and so on). It 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 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 user's 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 is 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 he or she decides not to continue? Each possibility must be explored, and the responsibilities for handling the situation assigned to one or more components.

Having walked through the various scenarios, the software design team eventually decides that all activities can be adequately handled by six components (Figure 3.3). The Greeter needs to communicate only with the Plan Manager and the Recipe Database components. The Plan Manager needs to communicate only 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.

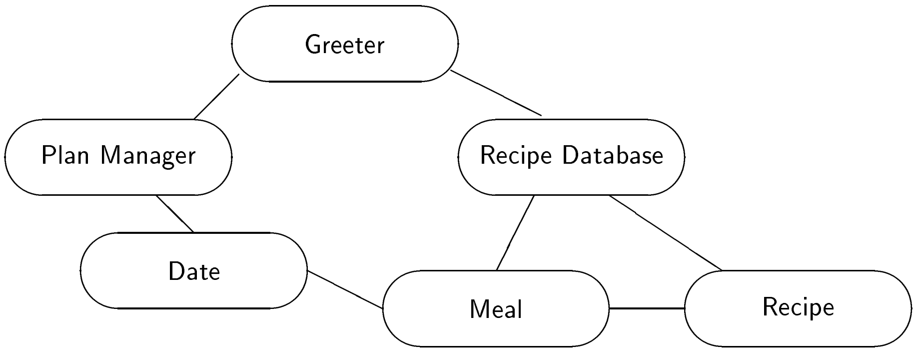


Figure 3.3: – Communication between the six components in the IIKH.

3.6.4 Interaction Diagrams

While a description such as that shown in Figure 3.3 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 an *interaction diagram.* Figure 3.4 shows the beginning of an interaction diagram for the interactive kitchen helper. In the diagram, time moves forward from the top to the bottom. Each component is represented by a labeled vertical line. A component sending a message to another component is represented by a horizontal 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 arrow forms, such as a solid line to represent message passing and a dashed line to represent returning control.) The commentary on the right side of the figure explains more fully the interaction taking place.

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

3.7 Software Components

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

3.7.1 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 certain information. Let us take as

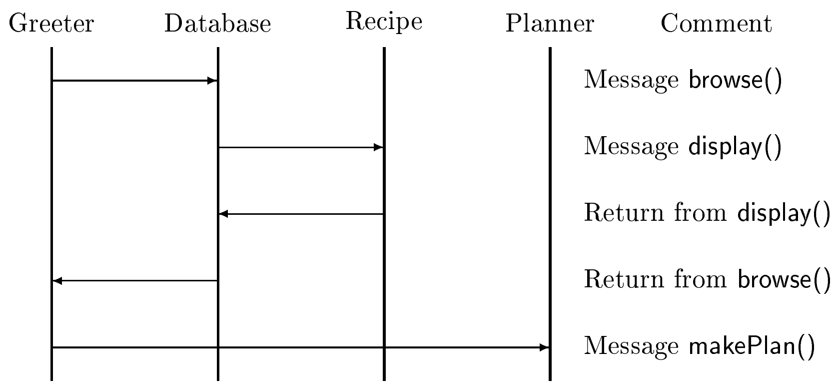


Figure 3.4: – An Example interaction diagram.

our prototypical component a Recipe structure from the IIKH. One way to view such a component is as a pair consisting of *behavior* and *state*.

- The *behavior* of a component is the set of actions it can perform. The complete description of all the behavior for a component is sometimes called the *protocol*. For the Recipe component this includes activities such as editing the preparation instructions, displaying the recipe on a terminal screen, or printing a copy of the recipe.
- The *state* of a component represents all the information held within it. For our Recipe component the state includes the ingredients and preparation instructions. Notice that the state is not static and can change over time. For example, by editing 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 that the Greeter component will not have any state since it does not need to remember any information during the course of execution. However, most components will consist of a combination of behavior and state.

3.7.2 Instances and Classes

The separation 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. However, all of these recipes will *perform* in the same manner. That is, the behavior of each recipe is the same; it is only the state–the individual lists of ingredients and instructions for preparation–that differs between individual recipes. In the early stages of development our interest is in characterizing the behavior common to all recipes; 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 a class is also used as a syntactic mechanism in Java. An individual representative of a class is known as an *instance*. Note that behavior is associated with a class, 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 instances of the class Recipe. They can all perform the same actions (editing, displaying, printing) but use different data values.

3.7.3 Coupling and Cohesion

Two important concepts in the design of software components are coupling and cohesion. Cohesion is the degree to which the responsibilities of a single component 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 inhibit 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 into the list of responsibilities of the component that holds the necessary data. For example, one might conceivably first assign responsibility for editing a recipe to the Recipe Database component, since it is while 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 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.

3.7.4 Interface and Implementation–Parnas's 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 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 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 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 difficult for students to understand, or to motivate. Information hiding is largely meaningful only in the context of multiperson programming projects. In such efforts, the limiting factor is often not the amount of coding involved, 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.

There is also an increasing emphasis on the reuse of general-purpose software components in multiple projects. For this to be successful, there must be minimal and well-understood interconnections between the various portions of the system. These ideas were captured by computer scientist David Parnas in a pair of rules, 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 that a programmer can experiment with several different implementations of the same structure without affecting other software components.

3.8 Formalize the Interface

We continue with the description of the IIKH development. In the next several steps the descriptions of the components will 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–for example, a component that simply takes a string of text and translates all capital letters to lowercase. 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, and 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 or global value, or maintained internally by the component, must be clearly identified.

3.8.1 Coming up with Names

Careful thought should be given to the names associated with various activities. Shakespeare has Juliet claiming that a name change does not alter the object being described,[1] but certainly not 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. The selection of useful names is extremely important, 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. Often a considerable amount of time is spent finding just the right set of terms to describe the tasks performed and the objects manipulated. Far from being a barren and useless exercise, proper naming early in the design process greatly simplifies and facilitates later steps.

The following general guidelines have been suggested [Keller 1990]:

- Use pronounceable names. As a rule of thumb, if you cannot read a name out loud, it is not a good one.
- Use capitalization (or underscores) to mark the beginning of a new word within a name, such as "CardReader" or "Card_reader," rather than the less readable "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. They are easy to misread as letters (0 as O, 1 as l, 2 as Z, 5 as S).

[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." *Romeo and Juliet*, Act II, Scene 2.

Date — **Collaborators**: Plan Manager, Meal

Maintain information about specific date

Date(year, month, day)–create new date

DisplayAndEdit()–display date information
in window allowing user to edit entries

BuildGroceryList(List &)–add items from
all means to grocery list

Figure 3.5: – Revised CRC card for the Date component.

- Name functions and variables that yield Boolean values so they 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.
- Take extra care 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 of a CRC card for the Date is shown in Figure 3.5. What is not yet specified is how each component will perform the associated tasks.

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

3.9 Designing the Representation

At this point, if not before, the design team can be divided into groups, each responsible for one or more software components. The task now is to transform the description of a component into a software system implementation. The major portion of this process is designing 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 of computer science come into play. The selection of data structures is an important task, central to the software design process. Once they 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 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, to ensure that expectations are fulfilled and necessary data items are available to carry out each process.

3.10 Implementing Components

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

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

As multiperson programming projects become the norm, it becomes increasingly rare that any one programmer will work on all aspects of a system. More often, the skills a programmer will need to master are understanding how one section of code fits into a larger framework and working 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 characterizing and documenting the necessary preconditions a software component requires to complete a task, and verifying that the software component will perform correctly when presented with legal input values. This is establishing the correctness aspect of the algorithms used in the implementation of a component.

3.11 Integration of Components

Once software subsystems have been individually designed and tested, they can be integrated into the final product. This is often 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 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 that the system is working 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 manifested in one software system, and yet to be caused by a coding mistake in another system. Thus, testing during integration can involve the discovery of errors, which then results in changes to some of the components. Following these changes the components should be once again tested in isolation before an attempt 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*.

3.12 Maintenance and Evolution

It is tempting to think that once a working version of an application has been delivered the task of the software development team is finished. Unfortunately, that is almost never true. The term *software maintenance* describes activities subsequent to the delivery of the initial working version of a software system. A wide variety of activities fall 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, perhaps as a result of government regulations or standardization among similar products.

- Hardware may change. For example, the system may be moved to different platforms, or input devices, such as a pen-based system or a pressure-sensitive touch screen, may become available. Output technology may change–for example, from a text-based system to a graphical window-based arrangement.
- User expectations may change. Users may expect greater functionality, lower cost, and easier use. This can occur as a result of competition with similar products.
- Better documentation may be requested by users.

A good design recognizes the inevitability of changes and plans an accommodation for them from the very beginning.

3.13 Chapter Summary

Object-oriented programming begins with object-oriented design. Object-oriented design is characterized by an emphasis on responsibility, rather than on structure. Responsibility and behavior are attributes that can be discovered for a software system well before any other features can be identified. By systematically tracing the behavior of a system, the design of the software elements flows naturally from the general specification.

A key tool in the characterization of behavior is the idea of scenarios. Developers trace through the execution of an imaginary system, identifying actions that need to be performed, and more importantly assigning the responsibilities for these actions to individual software components. A useful tool in this activity is the CRC card, which is an index card that records the responsibilities of a software system. As design evolves, the descriptions of the actions of each component can be rewritten in more precise formats.

Developing a working software systems involves many steps, frequently termed the software life cycle. Design and implementation are the first major steps. Implementation can be broken into the identification of components, development and testing of components in isolation, integration of components into larger units, and finally testing of the completed application. The life of a software system does not, however, halt with the first completed applications. Errors are uncovered, requirements change, and hardware modifications can all cause changes in the software system. The management of these changes that come after the first release is known as software maintenance.

Study Questions

1. What is the key idea driving object-oriented design?
2. How is the idea of responsibility tied to information hiding?
3. What are some of the characteristics of programming in the small?

4. How does programming in the large differ from programming in the small?

5. Why is information hiding an important aspect of programming in the large?

6. Why should the design of a software system begin with the characterization of behavior?

7. What is a scenario? How does a scenario help the identification of behaviors?

8. What do the three fields of a CRC card represent?

9. What are some of the advantages of using a physical index card to represent a CRC card?

10. What is the what-who cycle?

11. Why should the user manual be written before actual coding of an application is begun?

12. What are the most common sources of change in the requirements for an application over time? How can some of the difficulties inherent in change be mitigated?

13. What information is being conveyed by an interaction diagram?

14. Describe in your own words the following aspects of software components

    (a) Behavior and state

    (b) Instances and Classes

    (c) Coupling and Cohesion

    (d) Interface and Implementation

15. What are Parnas's principles of information hiding?

16. What are some guidelines to follow in the selection of names for components, arguments, behaviors, and so on?

17. After design, what are the later stages of the software life cycle?

18. What is software maintenance?

Exercises

1. Finish the development of CRC cards for the IIKH.

2. Having done Exercise 1, give a complete interaction diagram for one scenario use of the IIKH.

3. Describe the responsibilities of an organization that includes at least six types of members. 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 member type, describe the responsibilities and the collaborators.

4. Create a scenario for the organization you described in Exercise 1 using an interaction diagram.

5. For a common game such as solitaire or twenty-one, describe a software system that will interact with the user as an opposing player. Example components include the deck and the discard pile.

6. 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.

7. Consider a large program with which you are familiar (not necessarily object-oriented), and examine the names of variables and functions. Which names do you think are particularly apt? why? Which names do you think might have been badly selected?