INSTRUCTOR'S GUIDE

WITH EXERCISES TO ACCOMPANY

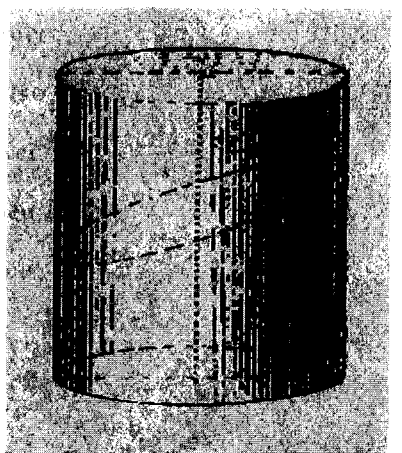
OBJECT-ORIENTED ANALYSIS AND DESIGN

WITH APPLICATIONS

GRADY BOOCH

SECOND EDITION

MARY BETH ROSSON



Instructor's Guide To Accompany

Grady Booch's

Object-Oriented Analysis and Design with Applications

Second Edition

Mary Beth Rosson
IBM T. J. Watson Research Center

ISBN 0-8053-5341-0

1 2 3 4 5 6 7 8 9 10—CRS—98 97 96 95 94 93

The Benjamin/Cummings Publishing Company, Inc.
390 Bridge Parkway
Redwood City, California 94065

Preface

This *Instructor's Guide* is a companion to *Object-Oriented Analysis and Design with Applications*, Second Edition, by Grady Booch. It is intended for use by instructors using the Booch book as the basis for a course on object-oriented design. It offers a range of questions, exercises, and projects that instructors might use to probe their students' understanding of and ability to apply the material in the book.

For the last five years, I have been working in the area of object-oriented instruction. My research has focused on educating professional programmers and on the use of interactive training environments. Thus, when I was approached by Dan Joraanstad of Benjamin/Cummings about developing an *Instructor's Guide* for Grady Booch's new edition, I was enthusiastic about experimenting with a different instructional format. I am also very interested in what others involved in object-oriented instruction may have developed. I would welcome a chance to review exercises or projects created by those of you teaching object-oriented programming or design. If you have electronic versions of such materials that you would be willing to share, please send them to rosson@watson.ibm.com.

My work on instruction has been guided by the principles of minimalism. This is a theory of instruction developed by Jack Carroll and his colleagues at the IBM T.J. Watson Research Center. It emphasizes learning by doing—learners will develop a richer set of skills and will be more able to apply them appropriately, if the skills are introduced and practiced in the context of realistic tasks. I have tried to apply this model as much as possible in the design of these materials, with an effort to include a variety of projects that should get the student out of the classroom and into a more realistic software development context.

Each chapter has four sections. The first is a set of Discussion Questions. These are intended to help the student review the conceptual material presented in the book. Instructors might choose to have students prepare written responses to these review questions, but their best use is likely to be as seeds for group discussion.

The Exercises are intended to give students a chance to apply the concepts and techniques presented in a chapter. Typically these exercises involve the generalization or extension of the material in the book. I have made an effort to reuse examples from the book, or from earlier exercises when possible, so as to encourage better integration of material from one chapter to another. The Sample Answers provide examples of the kind of exercise answers an instructor might expect from students who have studied the material in the chapter. As is normal in design situations, most of the exercises have no single "right" answer; instructors will need to use their own judgment assessing the adequacy of students' responses.

The Projects provide more open-ended tasks. Students might be asked to search out an expert and work with him or her to understand a new domain, or to research a range of tools or environments to analyze their underlying dimensions. These projects are opportunities for the student to tackle a problem of a more realistic size, to better prepare him or her for the activities of software development.

I would like to thank Grady Booch and Dan Joraanstad for giving me the opportunity to write this *Instructor's Guide*, and my management at IBM Research for giving me the flexibility to find the time to do it. But most of all, I want to thank Jack and Erin for their understanding and support as I single-mindedly applied myself to this writing project in an already overloaded summer.

Mary Beth Rosson

Contents

CHAPTER 1

Complexity

Discussion Questions

1. What makes software inherently complex? How might some of the problems associated with software design be mitigated by design tools or methods?
2. What are the defining characteristics of well-designed complex systems? How do these attributes apply to software?
3. What is the relationship between "part of" and "kind of" hierarchical structures in a complex system?
4. What general strategies have humans developed to make sense of complex systems? How does object-oriented decomposition reflect these general strategies?
5. How do algorithmic and object-oriented decomposition differ?
6. What is the meaning of design within the software engineering context? What is the role of model-building in the design process?

Exercises

1. Discuss the complexities apparent in the following software development situation:

   A group of developers in a small start-up located in Los Angeles have been contracted to build a chemical analysis system for an industrial chemical lab located in Santa Fe. The lab works with several thousand chemicals, and wants an exploratory tool for predicting the interactions of chemicals in novel combinations. The software house won the contract by under-bidding the competition, so they have little money available for travel. Only the team leader is able to make trips to Santa Fe and all contact between the chemists and the design team takes place through the team leader. She takes detailed notes on her discussions with the chemists who will use the tool, then briefs the team as a group when she returns. Requirements are established and a high-level design developed as a group. At this point, individual designers take on separate modules (e.g., the chemical database, the graph computation, the user interface). The developers are a close-knit group, and often discuss the detailed design of their modules with each other. This enables them to coordinate their designs from the beginning — for example, as the organization of the chemical database develops, the author of the graphing module directly incorporates the chemical grouping information embedded in the database and uses this information as an organizing rubric for his analysis options. Unfortunately, when the first prototype is shown to the client, the clients are unhappy with the chemical combination options. Both the database and the analysis modules must undergo substantial redesign.

2. The library at East-West University is a large building with over 4000 square feet of stacks, and a spacious reading/periodicals room overlooking the main quad. The library holds over 10K volumes, and subscribes to about 200 periodicals; most of these are archival and the library has bound journal volumes dating back as far as 1901. Books can be checked out for two weeks, periodicals for three days. A wide selection of reference aids are also available, but these must be used in the library. The material is heavily biased toward science, with engineering, life sciences, and mathematics the major topic areas. A smaller set of holdings in the liberal arts (literature, social sciences, and history) also exists. The staff consists of a head librarian, six students who take turns working the desk and shelving returns, andd two reference librarians.

   Characterize the East-West library system described above in terms of the five attributes of complex system discussed in the book. Elaborate the library description as necessary to support your analysis.

3. Fred is a Dodge dealer — his dealership maintains and sells a vehicle inventory typically numbering over 100 at any given time. It also services cars of all makes. While it stocks a variety of parts for Dodge models, it purchases parts for vehicles of other brands on an as-needed basis. What "kind of" hierarchies might be useful in organizing Fred's dealership? "Part of" hierarchies?

4. An important part of organizing a complex system is understanding how different abstractions are relevant to different situations. Imagine a salesman from Fred's dealership approaching a man who has just come in the door and begun browsing the floor models. Which of the abstractions you identified above are likely to be important to him in this situation? Next imagine the owner reviewing his business at the end of the year; what abstractions are relevant to his task? How are the two sets of abstractions related?

5. Figure 1-1 depicts a possible object structure for a simple blackjack game, listing some of the important elements connected in a "part of" hierarchy. In analyzing complex systems, understanding the *relationship(s)* between hierarchical components is just as important as identifying the components themselves. For each connection among nodes in the blackjack graph, specify the relationship(s) between the connected objects.

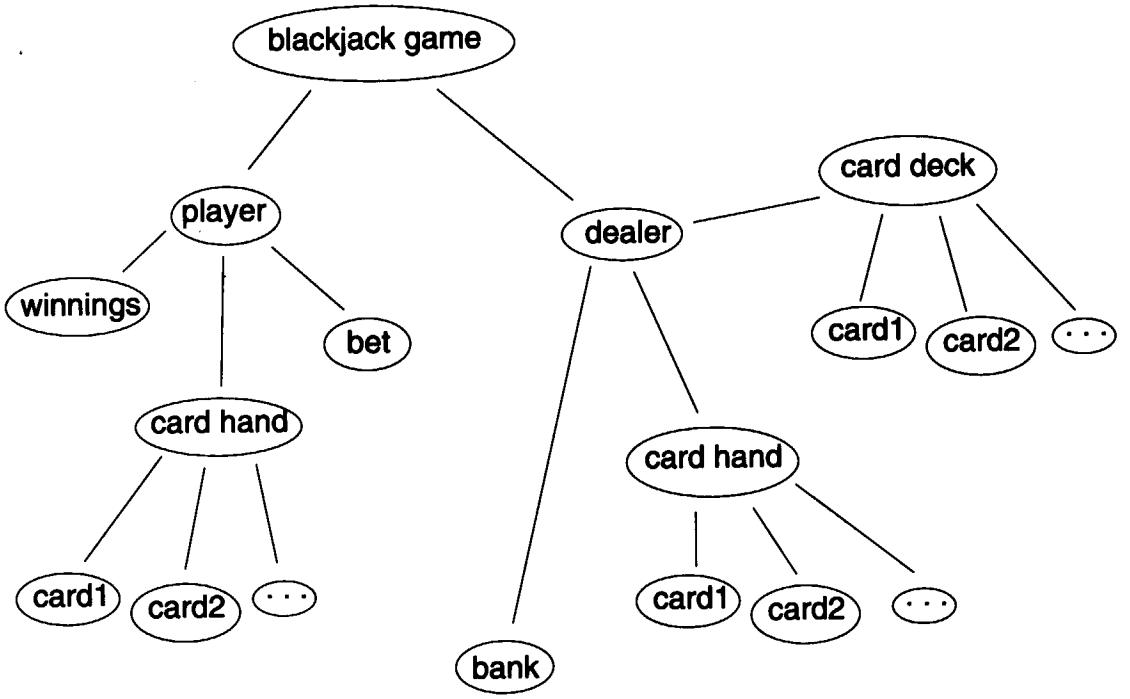


Figure 1-1
Object Structure for a Blackjack Game

6. Describe one or more extensions to the blackjack game depicted above that would have little impact on the complexity of the system. Then describe an extension that would make the system noticeably more complex. Why do the different extensions have differing effects on complexity?

Projects

1. Develop some measurements for assessing and contrasting the inherent organization in complex systems. As a start, use your measurements to distinguish between the different variants of the blackjack game you produced in Exercise 6. Then gradually add new systems with which you are familiar (e.g., your kitchen, the local hospital, a vegetable garden) and elaborate your scheme as necessary to position these systems.
2. Choose a complex system with which you are currently unfamiliar but for which you can locate one or more experts. Interview these experts and analyze the abstractions they use to organize their activities, developing both "kind of" and "part of" hierarchical views of the system. Example systems might be a restaurant, a course registration system, a volunteer agency, a research lab.

Sample Answers for Chapter 1 Exercises

The exercises represent design problems and have no single answer — any given situation can be analyzed in a variety of ways and to many different levels of detail. These sample answers reflect what we consider to be a reasonable analysis at a reasonable level of detail given the information available in the book.

1. The situation has several sources of complexity:

   a. The chemical inventory is large, and the goal is to produce an exploratory system for analyzing novel chemical combinations. This means that it will be impossible to know in advance all possible analysis scenarios. The problem is further complicated in that the set of chemicals is likely to evolve as the tool is used.

   b. The clients and the team are geographically separated. This reduces the bandwidth of communication between them. The development team's decision to minimize travel expense simply magnifies this problem.

   c. The team members' interactions are managed informally. This encourages mutual dependencies among their designs. Familiarity with the initial chemical groupings encouraged the author(s) of the analysis programs to solve a simpler, more concrete version of the analysis problem, developing a subsystem whose analysis options are explicitly based on the database categories. A more flexible system would have resulted if the analysis had made no assumptions about particular groups but rather had focussed on more generic characteristics of the chemicals (e.g., the factors that predict the groupings).

   d. The relevant problem abstractions changed, or at least became more apparent during the design process. A variety of software models can be used to characterize the same set of chemicals. It appears that the team leader either misunderstood the abstractions initially, or that the chemists' view of their domain evolved during the tool prototyping process. In the re-design, the database builder would be well-advised to create an abstract interface layer to the database itself, so that future changes to the database are less likely to be felt by its clients.

2. The five attributes can be seen as follows:

   a. *hierarchical organization of systems and subsystems*: The library is composed of several subsystems — the building, the reference material itself, the book loan system, and the staff. The building is composed of two main areas, the stacks and the reading room. Each of these contains their own structure (e.g., stacks, tables). Another subsystem is the library content, organized into books, periodicals and reference material. Yet another system is the mechanism by which materials are loaned, broken into books and periodicals. Finally, the staff can be seen as a department consisting of the head librarian and her student and professional staff.

   b. *relativity in level of abstraction*: The library system can be viewed at many different levels. For the president of the university, important abstractions will tend to be at a very high level: the overall size of the library holdings, its coverage with respect to the educational goals of the university, and so on. The professional staff are likely to be concerned with process abstractions — the flow of books, how many are checked out at any one time, how many are overdue and so on; the head librarian will also be concerned with the characteristics of the individuals she manages (i.e., their performance, what they are paid). The university students will be concerned with the organization of the reference material (e.g., Do we have books on social psychology? Where are the Psychological Abstracts?), as well as with individual volumes (is this the one I want?). They may also be concerned with expertise or personality of individual staff members, as they seek help in finding needed reference material.

   c. *intracomponent versus intercomponent communication*: The "communication" patterns are most easily imagined within the staff subsystem. Here, you would expect the reference librarians to be in close contact with one another (updating each other on new additions, on new reference techniques); similarly, the students staffing the desk would be in constant communication (alerting one another to intended breaks, managing lines as they develop and so on); those shelving books might be working together to divide up the returns. In contrast, those at the desk are likely to chat with the reference librarians and with those shelving books only during their breaks, or perhaps during slow times. At the other extreme,

the university president may talk with the head librarian, but is very unlikely to meet with the student helpers.

d. *commonality among subsystems*: Although the library's holdings are organized into books, periodicals and reference material, these subsystems probably share a great deal of internal structure. For example, all of them are likely to be organized by content (e.g., science and its subcategories). Each individual item (book, journal, etc.) has a title and date. Items also all have some sort of usage attribute (e.g., loan period), as well as a physical location. Similarly, although staff consists of subsystems for desk checkouts, re-shelving and reference aid, the individuals staffing these systems have many commonalities — they all have a particular work schedule, and have some set of library skills.

e. *evolution from successful simple systems*: The library almost certainly evolved from a simpler system established earlier in the university's history. Thus, the library may have begun with a small number of books, adding more content areas and different kinds of reference materials as the university's resources increased. In doing so, the organization would have been able to extend the system developed for loaning and returning books to other cases, for example the more restricted case of periodicals. Initially, they may also have had no specialized reference staff, relying on the head librarian to provide this function. As the library grew, this subsystem may have been added, with the head librarian generalizing and extending the skills developed in managing her checkout and re-shelving staff.

3. A car dealership embodies several important "kind of" hierarchies. The vehicles themselves can be organized this way, with different general vehicle categories (autos, trucks, vans), and subcategories within these (full-size, mid-size, compact, subcompact autos), down to the level of individual car models. The part inventory may also reflect such a hierarchy, organized by function at the highest level (e.g., body, tires, engine, suspension, accessories), subcategories within these (windows, mirrors, interior, exterior body parts), again down to the level of individual parts. Another "kind of" hierarchy may exist for the sales personnel or for the mechanics, with for example all salesmen having the basic persuasion skills, but some specializing their approach for particular vehicle types (e.g., those used for recreation).

   The "part of" hierarchical structure would be seen in the way the dealership organizes its activities. At the highest level, the organization might consist of sales, services, and customers. Sales might further decompose into inventory, personnel, and orders. The inventory would consist of the actual cars in stock. Note that the vehicle categories described above would not be part of this hierarchy, as these abstractions are captured by "kind of" relationships. However, vehicles themselves are composed of elements (color, price, model, year, etc.).

   If the dealership is large and the sales personnel have compartmentalized their responsibilities, this would be reflected in a personnel "part of" hierarchy (e.g., a sales manager and employees for RVs and so on). For most dealerships, though, this sort of organization is only very loose, with any salesperson ready and willing to sell any customer any vehicle!

   The orders abstraction captures the actual work-in-progress of the sales area. Its components include sales in progress and completed sales. Sales in progress might contain orders, where each order consists of customer, vehicle, price, promised delivery and so on. Completed sales might contain both final sales and financed sales, each also consisting of individual orders.

   Customers would probably have a fairly flat structure, as most customers are treated similarly by the dealership (depending on the dealership, you might want to distinguish between preferred and standard customers). Individual customers would have parts like name, address, sales history, service history, and so on.

   The service subsystem would have an analogous breakdown of inventory, personnel, and services, but the abstractions used in breaking down these components would be those relevant to carrying out service requests. An additional component of the services area might be an appointment book, as in this part of the business, scheduling appointments is a key task. The appointment book of course would break down successively into months, weeks, days, and hours.

4. A salesman approaching a prospective customer is concerned with customer characteristics. He may be thinking that this individual seems similar to some other customer to whom he sold a van several months ago. The salesman is also thinking about vehicle characteristics. He is sensitive to what particular models are on

the floor at this time, what options they represent, how they are or are not related to other vehicles currently in stock. Finally, of course, he will be anticipating an order, thinking about what financing options, etc., he might use to entice the customer into buying.

In contrast, the owner reviewing the year is not concerned with individual customer profiles. He may be interested in how much the customer database has grown (or shrunk!), but probably doesn't care about any one sale. He is not concerned with the options available in a particular vehicle, but rather may want to know how vehicles of a certain type (e.g., RVs) did over the year, how many were in inventory, how many figured in actual orders, and so on. In general, the owner will be working with high level abstractions — aggregates and categories of customers, vehicles and orders — whereas the salesman approaching a customer would be concerned with particular instances and their features.

5. Some of the important object relationships in blackjack might be characterized as follows:
   a. The blackjack game is related to the player and dealer by a *participant* relationship. The dealer and game may also be related by an *initiates* relationship.
   b. Both the player and the dealer are related to their card hands by several relationships — *holds, adds-card, evaluates, discards.*
   c. The relationship between the card hands and the individual cards is simply one of *contains*; that between the card deck and its cards is similar, but more constrained, *contains-at-position.*
   d. The relationship between the dealer and the card deck include *shuffles, removes-first.*
   e. The relationships between the player and his winnings include *adds-money, removes-money.* The same relationships hold between the dealer and the bank.
   f. The relationships between the player and his bet include *calculates, declares.*
6. Extensions having little impact on the complexity would involve addition of components that simply duplicate existing problem elements. Examples include adding a second player, or using a larger card deck. Greater impacts on complexity would come in extensions changing the structure of a subsystem, for example, allowing players the option of playing more than one hand, or providing for "team" play. Another source of additional complexity would be elaboration of the game's "kind of" hierarchies, for example, admitting different kinds of players (e.g., handicapping for experience), or using exotic card decks with additional card suits and/or values. Note however that the complexities introduced by these latter sorts of extensions would not be evident in the blackjack object structure graphed here; a combined graph showing both "part of" and "kind of" hierarchies would be required (see Figure 1-1 in the book).

CHAPTER 2

The Object Model

Discussion Questions

1. What have been the important trends in the evolution of programming languages? What problems have been addressed by different language generations? What are some of the important remaining challenges for future generations?
2. What makes a language object-based rather than object-oriented? What aspects of an object-oriented design would be impossible to implement in an object-based language? What would you do instead?
3. What are the four major elements of the object model? In what ways is each element crucial to successful object-oriented design?
4. The Smalltalk language allows no access by clients to the internal representation of an object, while C++ allows the designer to decide whether a member object should be publicly available. With respect to member operations, Smalltalk permits clients to request any operation defined for an object, whereas C++ developers specifically designate those operations available for public use. What are some of the pros and cons of these two language approaches?
5. For what kinds of situations is strong typing likely to be most helpful? When might it get in the way?
6. What are the major contributions of the object model to dealing with concurrency issues?

Exercises

1. One way to characterize an object and its behavior is through its invariant properties. What invariants can be used to characterize a house? A window? A skylight? An ATM? A bank account?
2. Recall the blackjack game whose object structure was graphed in the Chapter 1 exercises. One blackjack scenario involves the player requesting a "hit" (a new card) from the dealer. What role is being played by the player in this scenario? By the dealer? What precondition(s) and postcondition(s) are associated with the request?
3. Consider an appointment book object that might be used by the car dealership described in the Chapter 1 exercises. Let's say that our initial design decision was to render it as a "passive" object — it can be queried for available times, and can record appointments. What sort of extension to this object's functionality would make it an "active" object? What design issues would be associated with such an extension?
4. The book discusses some of the abstractions that might be developed to automate a hydroponics farm. Another object participating in this automatic gardening system might be a nutrient dispenser, used by a growing plan to control the relative amounts of different chemicals mixed together in a particular plant's nutrient solution. Given the following description of the role of nutrient dispensers in the farm activities, what might the abstract interface consist of? What should be encapsulated within the object?

   A nutrient dispenser is created for every vat of nutrient solution, and initialized with the locations of the chemical reservoirs over which it has control. Periodically, the growing plan checks the state of the nutrient solutions for the various plants. If a solution has moved beyond the acceptable range, the appropriate nutrient dispensor is asked to correct the problem, receiving information concerning the current solution profile and the target concentrations. The dispensor analyzes the difference between the two readings and injects the chemicals needed to reach the desired level. The dispensor can also report the current reservoir level of any chemical it controls.

5. The library at East-West University is developing a new browsing and request system for its holdings. Based on the project description below, what high-level decomposition of the work into modules would you propose? Why?

   The user interface to the system will run on two platforms, the 10 Mac workstations available in the library itself, and the PC-clone machines that the undergraduates use as home terminals; all of the database programs will run on the university mainframe. The students will be provided with a simple interface that allows them to browse the library database much like a card catalog; as they browse, they can request any item, and if it is available it will be forwarded to them. The PC interface will be menu-based; the menu design (wording, sequencing, etc.) is being developed as a class project in an undergraduate course on human-computer interaction. The Mac interface will be a direct manipulation system used by students working in the library. Librarians will be using an extended version of this Mac system to maintain the library database, to track usage, and to send out overdue notices. The project will build on an already existing database of library holdings, but will enrich the current classification system to support more flexible browsing.

6. Characterize the different hierarchical relationship(s) implied by the following descriptions.

   a. A baseball team consists of a manager and 25 players. Each player normally plays a single defensive position (e.g., pitcher, catcher), but some players are utility players who can serve in a range of positions.

   b. The main window for a library book browser has a title bar and three subpanes, one for selecting book categories, another for choosing particular books within a category, and a third for displaying an abstract of the book. Each subpane has its own scrollbar.

   c. A heating control system has three separate thermostats. Two of the thermostats are basic thermometers that must be adjusted manually, but the third can be adjusted either manually or by setting a timer.

   d. The Acme ATM provides two types of transaction, withdrawals and deposits. For every transaction, the user specifies a reference account (for deposit to or withdrawal from) and a dollar amount. Withdrawals can either be made in cash to the user or as a transfer to another account. For cash withdrawals, the user chooses the denominations of the payment; for transfers, the user specifies a target account. Recently, the ATM has begun offering a transfer+cash option, in which the user can carry out both sorts of withdrawals in a single transaction.

7. As part of a graphical editor, a designer has created an inheritance hierarchy of C++ classes to represent different kinds of nodes in the graph. The `Node` base class includes in its public interface the virtual member modifier functions `draw`, `position(int x, int y)`, `move(int x, int y)`, `select`, and `deselect`, as well as the selector function `isSelected`. The subclasses `RectangularNode` and `EllipticalNode` each define their own `draw` and `move()`. functions. The `EllipticalNode` also includes in its interface the `rotate(int a)` function. Given this node hierarchy, and the variables `n`, `rn`, and `en` declared as instances of `Node`, `RectangularNode`, and `EllipticalNode` respectively, which of the following statements are legal or illegal? Why?

   a. `rn.rotate(90);`

   b. `en = n;`

   c. `en.isSelected;`

   d. `rn.draw;`

   e. `n.move(50, 50);`

   f. `rn = en;`

   g. `n.rotate(180);`

8. As part of a military video game, a designer has created a vehicle class hierarchy. The base class is `Vehicle`, and it has `AirVehicle`, `LandVehicle` and `SeaVehicle` as derived classes. The class `SeaPlane` inherits from both `AirVehicle` and `SeaVehicle`. What design issues had to be considered in developing the `SeaPlane` class?

Projects

1. This chapter has focussed on applying the object model to the design of software. But another rich application area is in the design of user interfaces. Choose one or more user interfaces with which you are familiar, and analyze how the elements of the object model either do or do not apply to them. Note that the goal is to analyze the *functionality* of the user interface — how it is used — rather than the structure of the software that implements it.

2. At the end of Chapter 2, Booch describes five practical benefits derived from application of the object model: exploiting the expressiveness of OOPLs, reuse of design, resiliancy through stable intermediate forms, reduced risk in system integration, and naturalness. Choose a problem domain and try to generate examples of these benefits.

Sample Answers for Chapter 2 Exercises

The exercises represent design problems and have no single answer — any given situation can be analyzed in a variety of ways and to many different levels of detail. These sample answers reflect what we consider to be a reasonable analysis at a reasonable level of detail given the information available in the book.

1. Invariant properties for each of these objects might include:
   a. house — has walls, roof, entrance; sits on a solid surface
   b. window — installed as part of an enclosure; allows light to pass from one side to another
   c. skylight — installed in the roof of a building; allows light to pass through the ceiling of a building
   d. ATM — has a screen, keypad, card reader, receipt printer, money dispenser, money holder; accepts requests for money; dispenses money from valid accounts
   e. bank account — has account number, owner, amount; is held by a banking institution; increases when deposits are made; decreases when withdrawals are made
2. In this scenario, the player is the *client* and the dealer is the *server*. The request from the player-client has a precondition that the player has someplace to put the new card (e.g., a hand). Depending on the contract in force, there may also be a precondition that the player-client is a valid participant of the game; however it may be that assessing valid participation is a responsibility of the dealer. A postcondition for the hit operation is that a card object (i.e., with value and suit attributes) is returned.
3. One simple extension might be to give it a "reminder" responsibility — rather than simply answering questions about time available and accepting or removing appointments, the object might perform certain duties triggered by the passage of time. Thus, it might initiate a review of inventory at pre-specified times, or alert the service manager prior to appointments which have been noted to be resource-heavy or dependent on the availability of certain personnel.

   One design issue associated with such a decision would be the increased complexity of the appointment book object. It now would have a greater range of responsibilities, and the designer would need to consider whether a new abstraction (e.g., a "clock-watcher") is warranted. Whether or not the reminding function was given directly to the appointment book or to a service object, the designer would also now have to consider how to handle concurrency in the system, analyzing the priority of the possible reminder(s) and assessing and selecting among the facilities provided by the implementation language and platform for handling active objects.
4. The abstract interface to the software dispenser objects consists of the protocol for creating and (assuming a C++ implementation) destroying instances. The creation message would include arguments identifying the names and physical locations of the chemicals it will use to adjust the vat's solution. Also in the interface would be an `adjust()` operation that takes the two arguments concerning the current and desired solution chemical profiles, as well as a `currentLevel()` operation that takes as an argument the name of a chemical.

   Encapsulated within the object abstraction would be the representation it uses to encode the associations between chemical names and their physical dispensor valves. Also encapsulated would be the algorithm it uses to carry out its adjustment responsibility. For example, it will use a particular level of granularity in comparing one level to another, and in the current description this is hidden from its clients; it may also have the flexibility of using different chemical sources to achieve the same overall solution result. The mechanism by which it causes the injection to take place is also hidden, allowing for many variations in how the physical dispenser is turned on and off. Finally, the means by which it detects the current level of chemicals should also be hidden, again to allow for flexibility in the communication between the dispenser and the physical reservoirs.
5. Because the user interface runs on two platforms, it would probably be broken into two modules. The PC interface module would include a submodule for its constituent menus, so that these could be developed and iterated independently in the class project. The database itself would be another module — it might consist of a new "shell" providing the enhanced indexing capability, with the original database as a submodule, allowing for a clean separation between the original database code and its new interface to the other modules

in the system. The application programs would most likely be split into two modules — the student tasks (browsing and requesting) and the librarian tasks (library management), because they are serving different audiences, have rather different requirements, and may well be developed at different times by different individuals. Another module implied by the description would be one handling the network traffic between the user workstations and the database programs.

6. Each situation reflects a combination of "part of" and "kind of" relationships:

   a. The manager and the 25 players are "part of" the team; each player is "part of" the set of players. A defensive position is "part of" each player. Some of the players have a utility defensive position, which is a "kind of" two or more single positions.

   b. There are "part of" relationships between the window and its menu bar and three subpanes, as well as between each subpane and its scrollbar. The category selection, title selection, and abstract display are each a "kind of" subpane.

   c. The three thermostats are "part of" the control system; the thermostat allowing two means of adjustment is a "kind of" thermometer. The timer control is "part of" the specialized thermometer.

   d. A reference account and a dollar amount are both a "part of" the basic ATM transaction. Deposit and withdrawal are a "kind of" ATM transaction; and cash and transfer are a "kind of" withdrawal; cash+transfer is a "kind of" both cash and transfer withdrawal. A denomination specification is "part of" a cash withdrawal; a target account is "part of" a transfer withdrawal.

7. Three of the statements are legal, four are illegal:

   a. illegal — `rotate` is part of the interface to a peer class

   b. illegal — you cannot assign an instance of a superclass to an instance of one of its subclasses

   c. legal — this selector function is declared for the base class and can be used by any of its subclasses

   d. legal — this modifier function is declared in the base class and overridden in the subclass

   e. legal — this modifier function is declared in the base class

   f. illegal — these two variables are instances of peer classes

   g. illegal — this modifier function is not declared in the base class; it was added by the one of its subclasses

8. The designer has created a multiple inheritance situation where *repeated inheritance* will be an issue. Because the two superclasses are part of the same subhierarchy, one or more base classes will be inherited redundantly. In C++ the standard approach to this problem would be to use "virtual" inheritance (e.g., `class SeaPlane : virtual public AirVehicle, virtual public SeaVehicle`).

   Another design issue will be possible name clashes — for example, a sea plane will need to move both as an air vehicle when it is in the air and as a sea vehicle when in the water. It is quite likely that the both superclasses have declared `move()` functions, and the designer will need to distinguish between them.

Classes and Objects

Discussion Questions

1. What is an object? A class? How do you distinguish between these two constructs?
2. What dimensions would you use in describing an object's state? Its behavior? How are state and behavior related?
3. What factors should you consider in deciding whether or not to provide special operations for copying, assignment and equality?
4. Some languages (e.g., Smalltalk) provide garbage collection to clean up objects that are no longer being used, whereas other languages (e.g., C++) require object destruction to be managed by the programmer. What are the pros and cons of these two approaches?
5. What domain characteristics lend themselves to modeling via rich inheritance hierarchies? What are examples of such domains?
6. How does polymorphism contribute to the design of object-oriented systems? In what situations will it be most useful?

Exercises

1. Objects have properties which are usually static, and values, which are usually dynamic. What are the properties of a quarter in your pocket? What values do these properties have — are they dynamic? What about a literature review you are editing in a word processor — what are its properties and values?
2. Consider a Toyota Tercel. What are some of its behaviors that are dependent on its state? How about a computer chess game?
3. Draw a picture capturing the relationships among the four objects participating in the following situation. Within the context of this situation, which of the three participant roles (actor, server, agent) is each object playing? Back up your claims by showing the requests sent among the objects.

   At Fred's Dodge dealership, the customer records can initiate regular scheduled services as a function of their vehicle's maintenance schedule and its last scheduled service. The `customerSRJ` *object (representing Dodge customer Sue Jones) begins such a process: it checks with the* `theServiceCatalog` *for the appropriate service given Sue's 1988 Shadow service history, and sends a request for an appointment to* `theApptBook`*, sending along information concerning the service needed, the customer name, telephone, and scheduling preference. The appointment book finds the next available time and sends a notification to the service manager's* `anInBox`*, so that the manager can confirm or modify the proposed appointment.*

4. What is the cardinality of the association between the classes in each of the following pairs?
   a. the classes `Student` and `Course` in a university registration system
   b. the classes `Dessert` and `Recipe` in a cooking tutorial
   c. the classes `Vegetable` and `Nutrient` in a hydroponics farm management system
   d. the classes `Room` and `Window` in an architectural design system
5. An important subsystem within the chemical analysis system introduced in Exercise 1 of Chapter 1 are the graphical objects used in displaying the results of the chemical combinations. These are likely to include graphical primitives such as lines, splines, circles, ellipses, rectangles and polygons. These primitives would

be constructed into more complex configurations — filled or shaded surfaces, element distributions and a variety of graphs. Because of the complexity of the chemical interactions, it will be important to support 3-D rendering and manipulation of the experimental results.

Consider some of the classes and operations that might be developed in support of this subsystem. How might polymorphism contribute to the design of such classes?

6. In what way(s) do the subclasses below extend or restrict their superclasses?
   a. `RightTriangle` as a subclass of `Triangle`
   b. `Blackjack` as a subclass of `CardGame`
   c. `WaterTank` as a subclass of `StorageTank` (as in Chapter 2).
   d. `Periodical` as a subclass of `LoanableVolume`

7. Figure 3-1 depicts a class structure developed as part of the blackjack game described in earlier exercises. The labels on the arcs indicate whether public or private inheritance has been declared by the class developer, and the annotation next to each class indicates some of the public, protected and private parts of each class. Given the following declarations, judge legality of the statements a-g, and for legal method calls, indicate which method would be invoked.

   The declarations:

```
Bank stash;
BjHand myHand;
CardCollection theDeck;
```

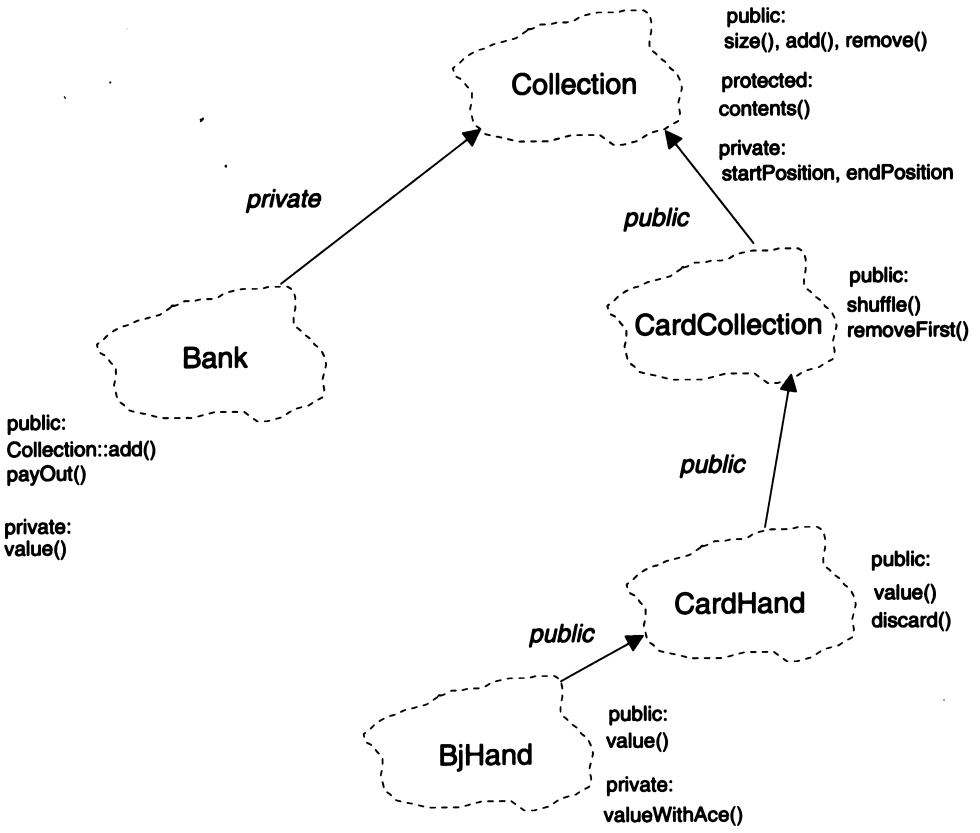


Figure 3-1
Partial class structure for the blackjack game

The statements to be judged:

a. `stash.add(100);`

b. `theDeck.endPosition;`

c. `theDeck.value();`

d. `BjHand = theDeck;`

e. `stash.contents();`

f. `myHand.size();`

g. `myHand.valueWithAce();`

Projects

1. Coupling and cohesion are important indicators of the quality of a class hierarchy — a good design strives to achieve loose coupling across classes and modules, but good cohesion within these design units. How might you go about assessing the coupling or cohesion of a class or a module? Try to develop a scheme that distinguishes between coupling due to inheritance structures and that due to collaboration.

2. A metaclass is a class whose instances are themselves classes; a class-instance is responsible for creating instances (of itself). Some languages (e.g., Smalltalk, CLOS) permit the manipulation of metaclass functionality, thus allowing the developer to change the way that class objects behave (e.g., what it means to create an instance). See if you can sketch out some possible variants on a class object's behavior (drawing from and expanding on the discussion at the end of Section 3.4), and develop corresponding problem scenarios for which the different variants might be useful.

Sample Answers for Chapter 3 Exercises

The exercises represent design problems and have no single answer — any given situation can be analyzed in a variety of ways and to many different levels of detail. These sample answers reflect what we consider to be a reasonable analysis at a reasonable level of detail given the information available in the book.

1. A quarter is a tangible object. It has properties such as shape, size, color, location, value, owner. Values of these properties would be circle, about 1 inch in diameter, silver, your pocket, 25 cents and you. Most of this object's physical characteristics are static; only the location and the owner are dynamic (although you could possibly paint a quarter and change its color).

   A document is a "soft" object. It has properties such as name, size, date last modified, location, font, spacing, document style. All of its values are dynamic — "OODreport", 1200 lines, today, your word processor, and so on.

2. The Tercel will start only if an acceptable key is in the ignition; it will transport people only if there is sufficient gasoline in the tank, and if it has four inflated tires. Its engine will run more or less smoothly depending on the adjustment of its timing, carburetor and so on.

   The game will allow the player to move a piece only if the new location is a legal move from its current position; its choice of a move is based on its analysis of the game's current state; it will remove a piece from the board only if another piece is on top of it; it will declare a "check" only if the pieces of one player are threatening the king of the other; it will declare a "checkmate" only if the king of one player can make no move to save itself.

3.

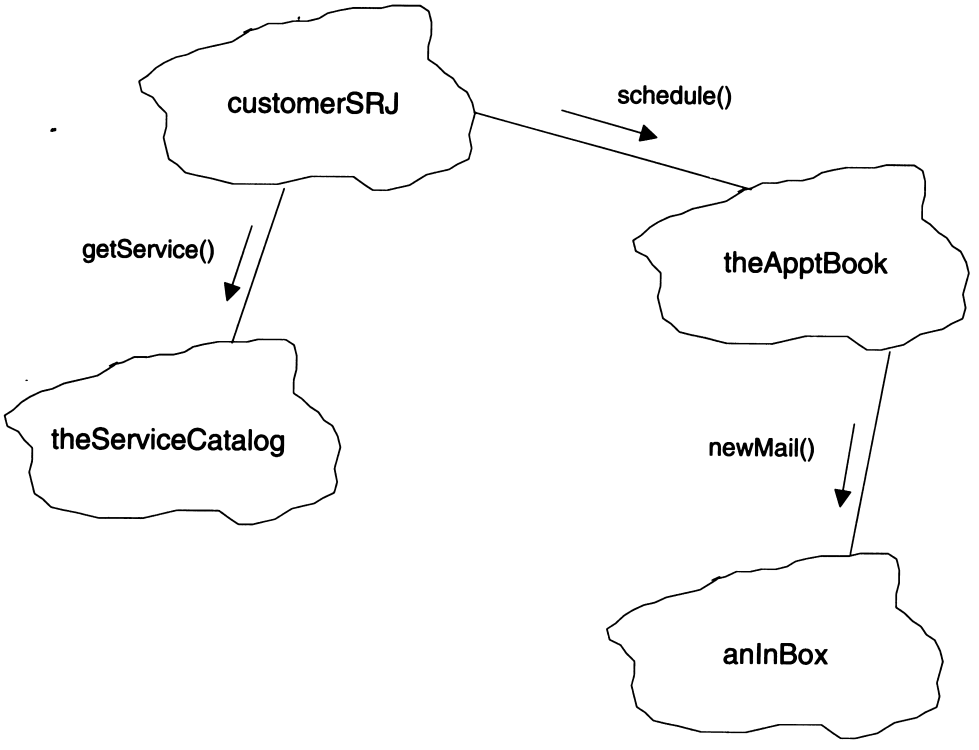


Figure 3-2
Objects in a car service scheduling scenario

In this scenario, the customer record is serving as an actor, initiating the process. The service catalog is simply a server, responding to a request from the customer record. The appointment book acts as an agent, in that it both responds to a request from the customer record and operates on the in-box to set up a confirmation process. In this description, the in-box is acting as a server — it enables the confirmation process, but the confirmation sent back to the appointment book will probably come from some other object (e.g., via the user interface to the service manager).

4. The cardinality of the associations is as follows:

   a. many to many: any given student may be taking from 0 to N classes; any given course will have from 0 to N students registered

   b. one to one: any given dessert will have one recipe associated with it; any given recipe will produce one particular dessert. (If the tutorial models dessert results as well as dessert specifications, this situation would be seen as many to one, in that a given recipe would produce many physical dessert instances).

   c. many to many: any given vegetable will require from 0 to N nutrients; each nutrient may be needed by 0 to N vegetables.

   d. one to many: a room will have from 0 to N window instances associated with it; a particular window will always be associated with a single room instance.

5. The graphical classes would probably be developed with a shared abstract protocol for operations such as `draw`, `rotate()`, `scale()`, `translate()` and so on. These operations would then be defined for each concrete class, leading to a wide distribution of same-named operations throughout the class hierarchy. The users of these primitive elements (e.g., the graphs and surfaces) could then make use of a single abstract protocol and not worry about how their individual elements carry out these operations.

6. Each case involves both extension and restriction, but they vary in the relative amount of the two forms of inheritance.

   a. The inheritance in this case is mostly restrictive — one of the three angles in the subclass is constrained to be 90 degrees; the sides are related by the formula $a^2 + b^2 = c^2$. However, because of the new constraint some new behaviors may also be possible, for example perpendicular alignment makes sense for a right triangle but not for triangles in general, and thus might be an extension provided by this subclass.

   b. In this case, the inheritance is mostly extension. The `CardGame` superclass would serve as an abstract class and would have little in the way of specific behavior. Its job would be to set up the common characteristics of card games (perhaps providing for the deck, for some number of players, a dealer and so on). The `Blackjack` class then adds specific behaviors needed to play this particular kind of card game (e.g., the rules for playing the game). In some cases, the definition of this concrete behavior may involve a restriction as well — for example, all we can assume about a card game is that it uses cards, whereas for blackjack card usage is restricted to cards from a single shuffled deck.

   c. The inheritance here is pretty much an even mixture of extension and restriction. Although the superclass will define some general behaviors for filling and draining the tank, the `WaterTank` will need to specialize these operations for the case of water (e.g., using a flow appropriate for water). It may also introduce new behavior that is relevant only to water (e.g., assessing and manipulating the water temperature).

   d. Unlike the blackjack example, most of the behavior and structure of loanable items would be common across different types of items and so could be defined in the superclass — all have a loan period, all have physical characteristics (title, age, library location), and all have the behavior needed for recording loans and returns. Nonetheless, this case would still be largely inheritance by extension: a periodical can be viewed as a cumulating set of issues which eventually are bound as a volume, and it would need the state and behavior to track where it is within the current volume and to initiate binding when appropriate.

7. Two of the statements are legal, five illegal:

   a. legal — the `add()` operator is explicitly declared as public in the Bank subclass, meaning that clients of `Bank` can invoke this superclass operation, even though the inheritance is private.

   b. illegal — this is private in the superclass and not available even in subclasses with public inheritance.

c. illegal — this public operation is an extension made by a subclass and not available to the superclass

d. illegal — you cannot assign an object to an instance of a subtype.

e. illegal — although the `contents()` operation is protected, it is available only to subclasses with public inheritance.

f. legal — the `size()` operation is declared in the base class and inherited through a line of public inheritance.

g. illegal — the operation is part of the private interface to `BjHand`, not available for client use.

CHAPTER 4

Classification

Discussion Questions

1. What is the relationship between analysis and design in object-oriented development? What are examples of abstractions that might be realized during analysis? During design?
2. Why must classification be seen as an iterative process? What sorts of changes to a classification scheme are likely to occur as it evolves?
3. How would you describe the classical approach to object-oriented analysis? How does this compare to a more behavioral approach?
4. Who should participate in domain analysis? What kinds of activities are you likely to carry out during domain analysis?
5. How can traditional structured analysis techniques contribute to object-oriented design? What are some potential pitfalls of this approach?
6. Why are class-based abstractions insufficient in analyzing a problem domain? What does the notion of *mechanism* add to problem analysis? What are examples of mechanisms?

Exercises

1. An inherent difficulty in object classification is that most objects can be classified in a variety of ways, and the choice of classification depends on the goals of the designer. Consider the holdings of the East-West University library first described in the Chapter 1 exercises. What are some of the ways you might classify these holdings? For what purpose(s) might you use the different classifications?
2. Consider the customer database for Fred's Dodge dealership. Describe how a set of customer categories would be derived using the classical property-based approach, and compare this to how categories might be derived via the conceptual clustering and prototype approaches.
3. Suppose that you are involved in the design of East-West University's library browsing and loan system. What are some of the important use cases you might develop in the analysis phase (simply enumerating them is sufficient)? Try to generate at least 4-5 cases.
4. Read the following problem statement and use the "informal English description" technique to identify candidate objects and operations. Defend your analysis.

   We wish to build a university course registration system. We want students to be able to browse and select among all the undergraduate courses offered in a semester. A student's selections are entered into his or her permanent record (transcript); each course is updated with an outcome at the close of the semester.

   In registering for courses, each student fills out a schedule, with minimum and maximum number of courses determined by student category (e.g., part-time, honors program, standard). Students considering a course can browse information about its teacher and a course abstract, as well as determine what prerequisites it has, when it meets, and how many credits it is worth. A course selection is accepted if its prerequisites have been met, if it doesn't exceed the total allowed hours, and if its meeting time does not conflict with other courses on the student's schedule. Teachers can use the system to monitor the registration process for the courses they have been assigned, to see how many students are currently registered for a course and who they are.

5. Suppose that the following class hierarchies have been developed as part of the chemical analysis system proposed in the Chapter 1 exercises. How might the class structure be refined? Why?

   Two different developers have been working on the two main views of the results of the chemical combination experiments — line drawings and shaded complex surfaces — and have created two rich sets of functionality. The important characteristics of the line drawings are line width, and orientation, as well as line segment color. The complex surfaces also have an orientation attribute, but differ from the line drawings in that they represent information in terms of element density and shape. Either sort of representation can be created and manipulated (e.g., rotated in 3-D) within the currenntly active analysis window. Because the developer working with the complex surfaces has discovered that 3-D manipulation of some surfaces is very expensive, he has provided the option to translate back and forth to a 2-D representation. Thus his hierarchy provides both 3-D and 2-D versions of its drawing and manipulation operations.

Projects

1. Select your favorite end-user application (e.g., word processor, spreadsheet, email). Try several of the analysis techniques described in the book (e.g., use case analysis, CRC cards, informal English descriptions), and see what kinds of objects and responsibilities you come up with. How do the techniques compare? Did you generate similar sorts of objects using the different techniques or do they differ on one or more dimensions (e.g., concreteness, activity, complexity)?
2. Choose an object-oriented software library for which you have documentation (and preferably code; examples would be Interviews, Borland's ObjectWindows library). See how many idioms, design patterns, or mechanisms you can identify across the classes in the library.

Sample Answers for Chapter 4 Exercises

The exercises represent design problems and have no single answer — any given situation can be analyzed in a variety of ways and to many different levels of detail. These sample answers reflect what we consider to be a reasonable analysis at a reasonable level of detail given the information available in the book.

1. A curriculum development system might classify the holdings according to their pedagogical characteristics. Thus, one might distinguish among textbooks, supplementary reading, reference books, current affairs, and so on.

   A library management system would classify the holdings according to usage characteristics, distinguishing among books, periodicals, references. Note that while these categories might overlap somewhat with those listed above, the details of the abstractions are likely to be quite different (e.g., a course designer would want to know the level of difficulty associated with a text, whereas this would not matter to library management; in contrast, a course designer would have no interest in the current physical location of a volume).

   A typographic design system would classify the holdings according to the physical characteristics of each volume's content — font types and sizes, page layouts, paper and ink quality, and so on.

2. Although the three approaches might well end up with similar or overlapping categories, the process of developing categories would be quite different.

   a. The classical approach would base its categories on discrete characteristics of customers. So, for example, we might have male vs. female customers, or customers owning 1 vehicle vs. those owning more than 1 vehicle. Any of these properties in combination could be used to construct more complex categores, for example, female customers who bought a car last year without financing.

   b. The conceptual clustering approach would begin with conceptual descriptions and then find customers fitting the description. So we might have "preferred customers" and "standard customers", and someone(s) involved in the design of the database would make a judgement as to which customers fit into which category.

   c. The prototype approach would begin with a "typical" customer representing some business-relevant interaction(s). Perhaps, for example, Sue Jones is a typical "conservative" customer — she follows the book on her services, buys a new car when her warranty expires, always asks the manager to call her after her car has been serviced, and so on. Other customers might then be categorized as "conservative" if they are seen as similar to Sue. (Note that a possible downside of this approach is that irrelevant characteristics of Sue might lead to inappropriate similarity judgements).

3. The original problem description included both student users and staff:

   a. One student use case might be opportunistic browsing. The user has no particular content goals in mind, but has some spare time and is just "looking around" electronically. One can think of the analogy of walking up to the card catalog and riffling through the cards in one drawer and then moving on to another.

   b. Another student use case would be direct query. A user knows the item he or she wants (i.e., can identify it uniquely), and simply wants to borrow it. A variant of this would be where the user knows the item exists but needs help in identifying it uniquely to the system.

   c. A third student case would be reference assistance. Here, the user doesn't know what if anything might suit her needs (this is perhaps the most typical case), and would like to see things that are "in the neighborhood". Thus, the student can specify content areas, perhaps the type of reference material (book, periodical, etc.), maybe even a rough time of publication. The possible matches can then be browsed to make specific selections.

   d. From the perspective of the desk staff, one important use case would be the book return scenario. The staff member picks up a book, identifies it somehow to the system (e.g., through a magnetic strip in its spine), and records it as returned.

   e. Another staff use case might be the generation of overdue notices. If the campus is on a network, this might be done electronically — when a book has not been returned on time, a notice is sent automatically

to the borrower. If the borrower is not reachable electronically, a backup system using standard mail might be used.

f. Another staff use case might be the review of current holdings. In this case, they would want to see the number (and perhaps title/date/author) of holdings in various categories. During this review, they might want to select holdings for discard, or make annotations of current needs, perhaps generating a report summarizing their analysis when finished.

4. In the following, we indicate things that are unlikely to be pulled out as objects, as well as the likely candidates.

   a. *course registration system* is unlikely to be a single object, rather it is the system we are building.

   b. *student* is a candidate object. It has attributes (name, degree program, transcript), and operations (check schedule, check prerequisites, add course). Note that at least part of the student object will be a persistent object.

   c. *course* will be an object. It has structure to it (e.g., teacher, registered students, descriptive material), and it admits operations (e.g., add a student, browse, see total students).

   d. *semester* is a candidate object, as a container of courses. However, its behavior seems not to be very rich.

   e. *transcript* is a candidate object. It has state (courses organized by semester) and behavior (add course, add course outcome).

   f. *course outcome* is unlikely to serve as an object. It is a simple value (e.g., a symbol like an 'A' or a number like a 1.0).

   g. *course schedule* is a candidate object. It contains the current set of proposed courses, and has behavior to add or remove courses, as well as possibly to tally up total hours registered, and to detect schedule conflicts.

   h. *degree program, credit hours, meeting time, course abstract, prerequisites* are unlikely to serve as objects. As for course outcome, these seem to be simple values with no interesting structure or behavior.

   i. *teacher* is a candidate object. It has internal structure (name, courses taught, other biographical information), and operations it admits (e.g., assign to course).

5. One refinement of the class structure would be the development of an abstract superclasses of the two view hierarchies. The line drawings and surfaces have a fair amount in common (e.g., their interface to the containing window, their protocol for scaling and rotation) that could be represented as shared state and behavior in a superclass.

   A second refinement would involve splitting the surface functionality into two subhierarchies, one for 2-D and one for 3-D variants. The operations involved in drawing and manipulating in different dimensions are likely to be different enough that they should be separated in the class structure.

CHAPTER 5

The Notation

Discussion Questions

1. Design notations can be created at many different levels of detail. In what situations would very rough notations be appropriate? When would you want to produce very detailed notations?
2. What are the different models of system development? How are these different models represented in the Booch notation?
3. For which parts of the design model is it most important to develop specifications in addition to graphical notation diagrams? Why might you want to develop a full set of specifications?
4. Class category diagrams and top-level module diagrams (subsystems) typically have a very similar structure. Why is this? What is the source of differences in structure?
5. The chapter describes six main diagram types (class, module, state transition, object, event trace, and processor). What are examples of activities supported by the different diagram types?
6. The notation presented here is largely language-independent. Nonetheless, not all aspects of the model and its notation are equally relevant to all object-oriented languages. Which diagrams seem most useful to the object-oriented languages with which you are familiar? Why?

Exercises

1. Part of the Mac interface to the East-West University library system is a graphical network depicting topic areas and their relations to one another; students can navigate through the network, selecting nodes and arcs and reading summary material, in the course of identifying reference material of interest. Below is a scenario depicting the node selection in this system; prepare an object diagram that corresponds to this scenario.

   The episode begins when an unnamed instance of `NetworkWindow` *sends the* `mouseDown()` *message to an unnamed instance of* `NetworkPane`. *The subpane iterates through its node network, sending the* `underMouse()` *message to each constituent* `Node` *instance; the pane identifies* `N` *as the node under the mouse. It then sends the message* `selectNode()` *to the window, including the identified node* `N` *as a parameter. The window responds by sending* `N` *the* `select()` *message. The node completes the episode by invoking the* `select()` *operation on* `Net`, *its associated* `NetworkNode`, *and then invoking the* `drawReversed()` *operation on itself.*

2. Using the same node selection episode, prepare an event trace diagram.
3. Read the following description of some of classes developed for Fred's Dodge dealership sales subsystem and develop a class diagram depicting class characteristics and relationships.

   In Fred's sales subsystem, the vehicle hierarchy is a lattice. So, for example, two subclasses of the abstract class `Vehicle` *are* `FamilyCar` *and* `RecreationalVehicle`; *the class* `Mini-van` *inherits from both of these subclasses. To avoid multiple copies of the* `Vehicle` *information, its subclasses' inheritance is virtual.*

   A private attribute of the `Vehicle` *class is* `stock`; *a public operation of interest is* `cost()`. *The actual information about vehicles is contained in an instance of* `VehicleDatabase`, *a class variable of* `Vehicle`.

   The `InventoryController` *has the responsibility of tracking sales of various vehicles and maintaining an appropriate supply. It has been declared a "friend" of* `Vehicle` *so that it can ac-*

cess the details of any particular kind of car (e.g., its current stock). Nested within `InventoryController` *is a supporting class* `MarketAnalysis`.

The dealership is not large, and has room for at most five salespeople on its roster. The `SalesPerson` *class is a client of the* `InventoryController`*, using its analytic information to set up sales goals. Each* `SalesPerson` *physically contains by value exactly one instance of the* `SalesPlan` *class.*

4. Figure 5-1) depicts a class diagram for part of the blackjack game introduced in earlier exercises. Study the diagram and write a textual description of all the information it contains.

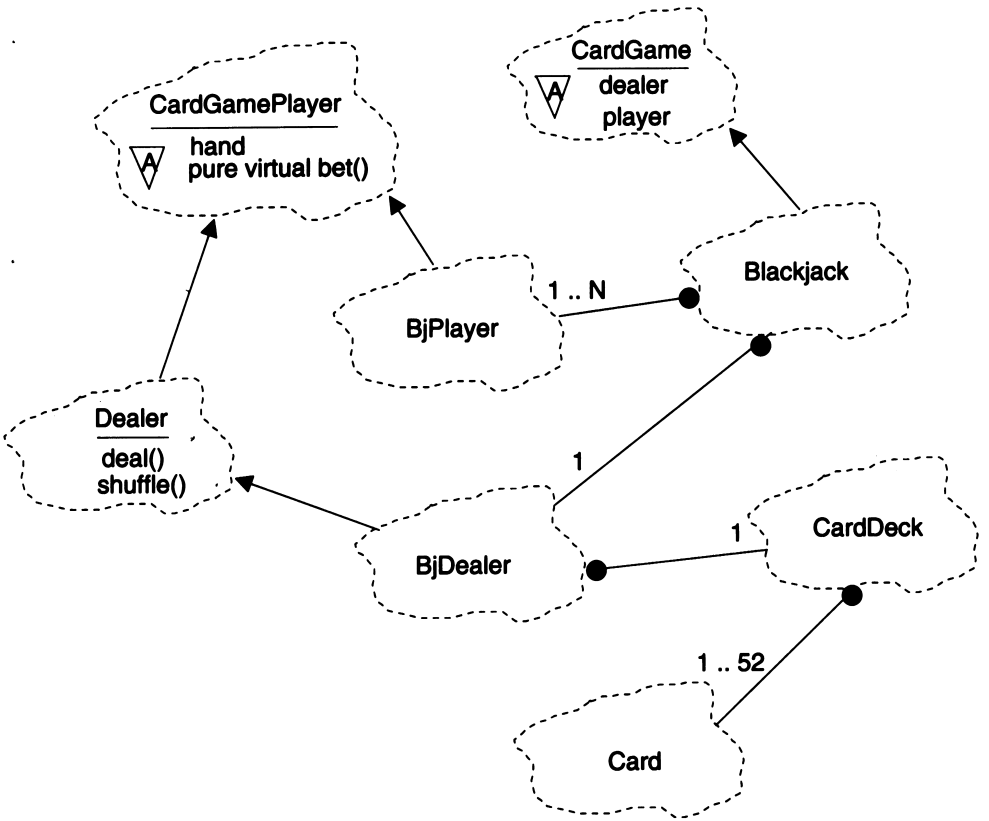


Figure 5-1
Blackjack Class Diagram

5. In Exercise 3, you worked from a textual representation to generate a graphical representation of a class structure; in Exercise 4 you did the reverse. Looking across these two exercises, what do you see as the relative advantages of graphical and textual represention?

6. Study the state transition diagram in Figure 5-2 (on the next page) depicting the main states and transitions in the class `TextEdit`. What information does the diagram convey?

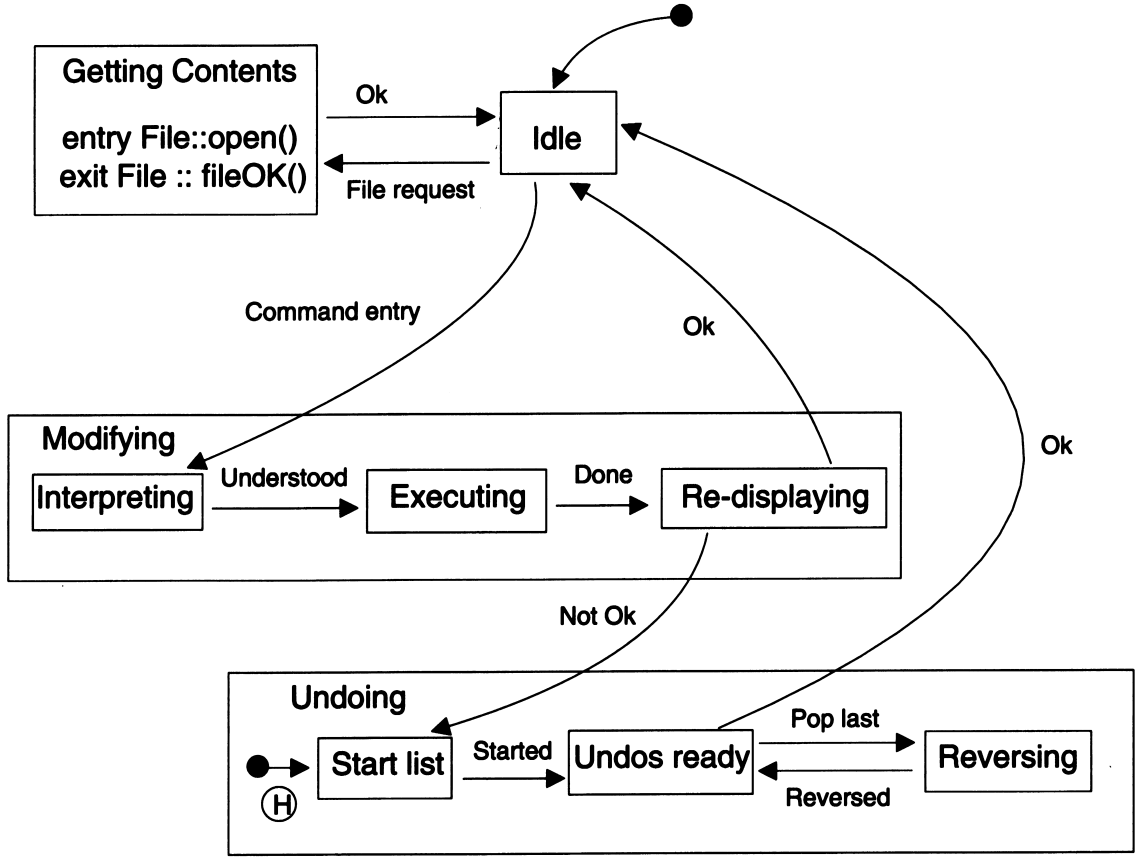


Figure 5-2
TextEdit State Transition Diagram

7. Examine the module diagram in Figure 5-3, depicting some of the physical architecture of the chemical analysis system discussed in earlier exercises. What does the diagram tell you?

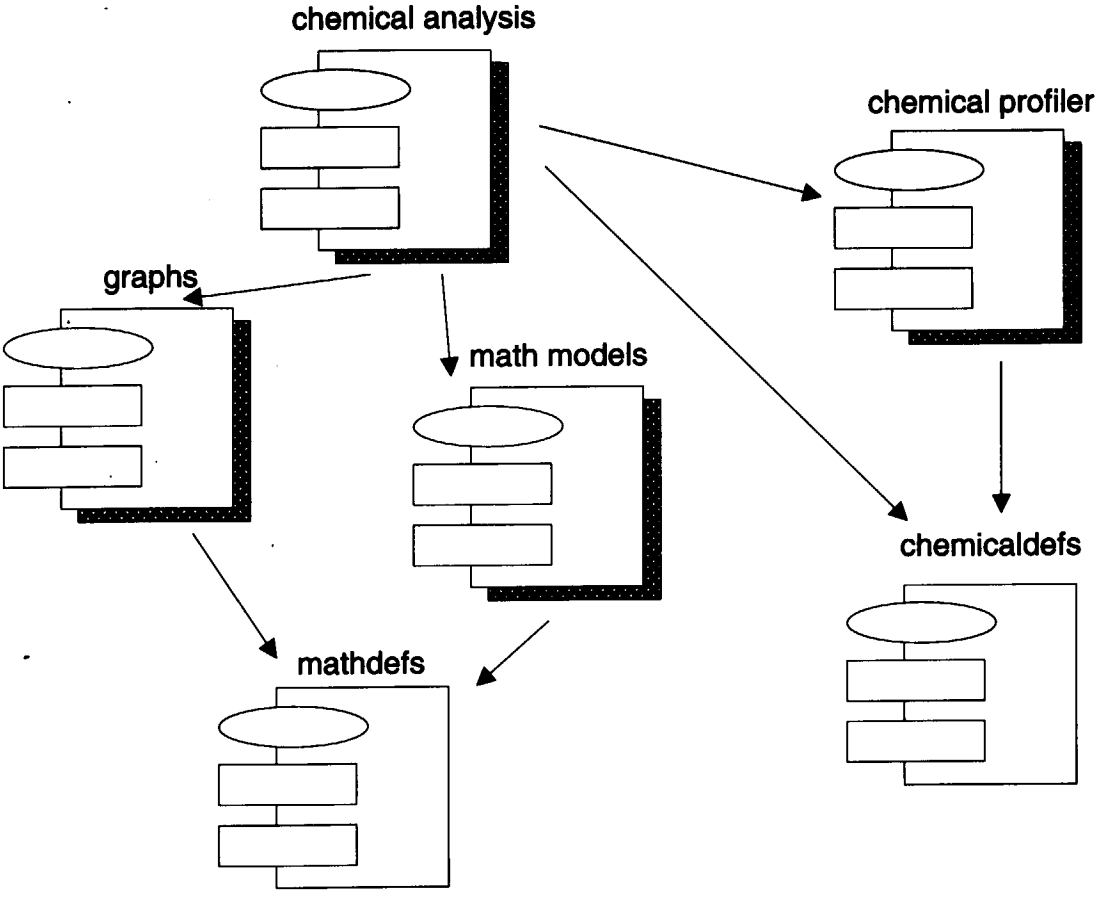


Figure 5-3
Chemical Analysis System Module Diagram

Projects

1. Find another book on object-oriented design that includes a graphical notation system (candidates would be Jacobson, Christerson, Jonsson & Overgarrd, *Object-Oriented Software Engineering*, Addison Wesley, 1992; Rumbaugh, Blaha, Premerlani, Eddy & Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991), and compare its notational system to the Booch system. How are they the same? How different? What seem to be the strengths and weaknesses of each system? If you were to design your own system, what changes would you make?
2. Visit the Computing Center of a university or large business. Talk to the staff who run the center and see if you can develop a processor diagram for the services it provides. Be sure to include workstation network(s) if they exist. Use nesting liberally!

3. The book suggests that one advantage of the graphical notation is that it can be created and supported by tools. These tools can then carry out some of the tedious work of managing the multiple representations — consistency checking, constraint checking, completeness checking, and analysis. Take each of these services that might be provided by a tool and elaborate — set up some development situations and describe what such a tool would be doing, how it would be helping the developer.

Sample Answers for Chapter 5 Exercises

The exercises represent design problems and have no single answer — any given situation can be analyzed in a variety of ways and to many different levels of detail. These sample answers reflect what we consider to be a reasonable analysis at a reasonable level of detail given the information available in the book.

1. The episode includes the 6 ordered steps depicted in Figure 5-4. The object labels indicate the class and/or object name information available in the scenario; the adornment on the link to the `Node` instance indicates visibility as a parameter.

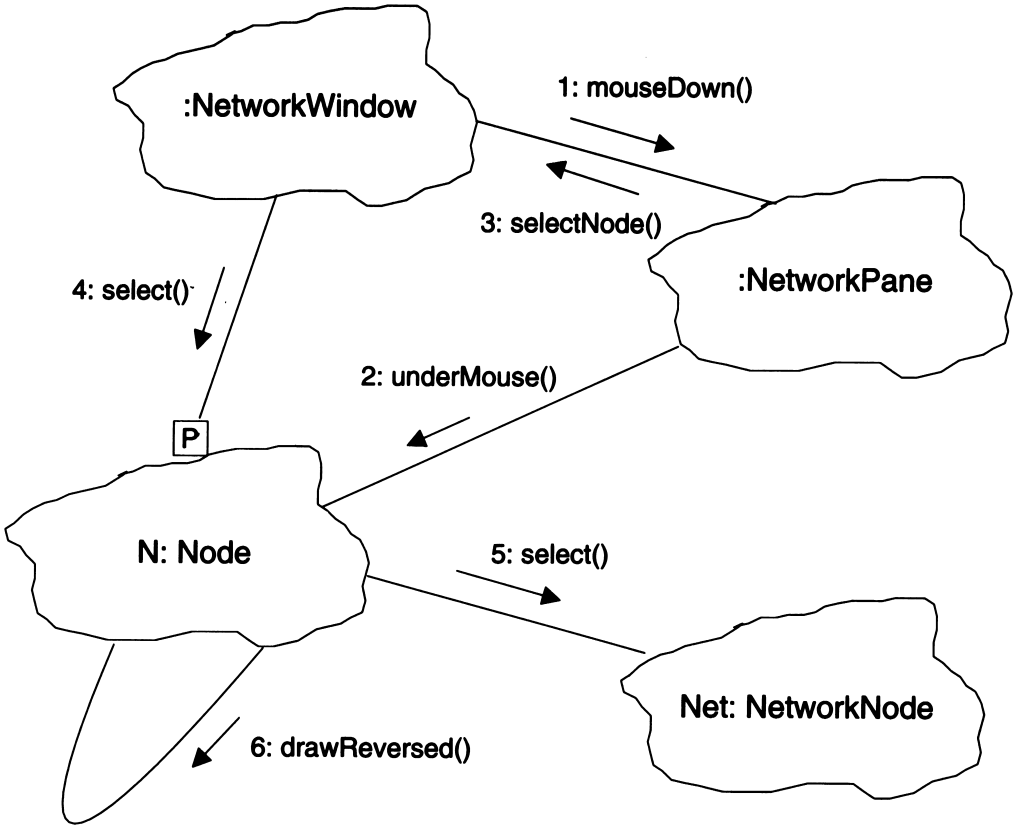


Figure 5-4
Node Selection Object Diagram

2. The event trace in Figure 5-5 (on the next page) captures the same information as the object diagram. The messages aren't numbered, but their vertical ordering conveys order of invocation. The only other difference is that because the object links aren't shown explicitly, they cannot be adorned with additional information, e.g., that `N:Node` was accessed by the pane as an argument in one of its operations.

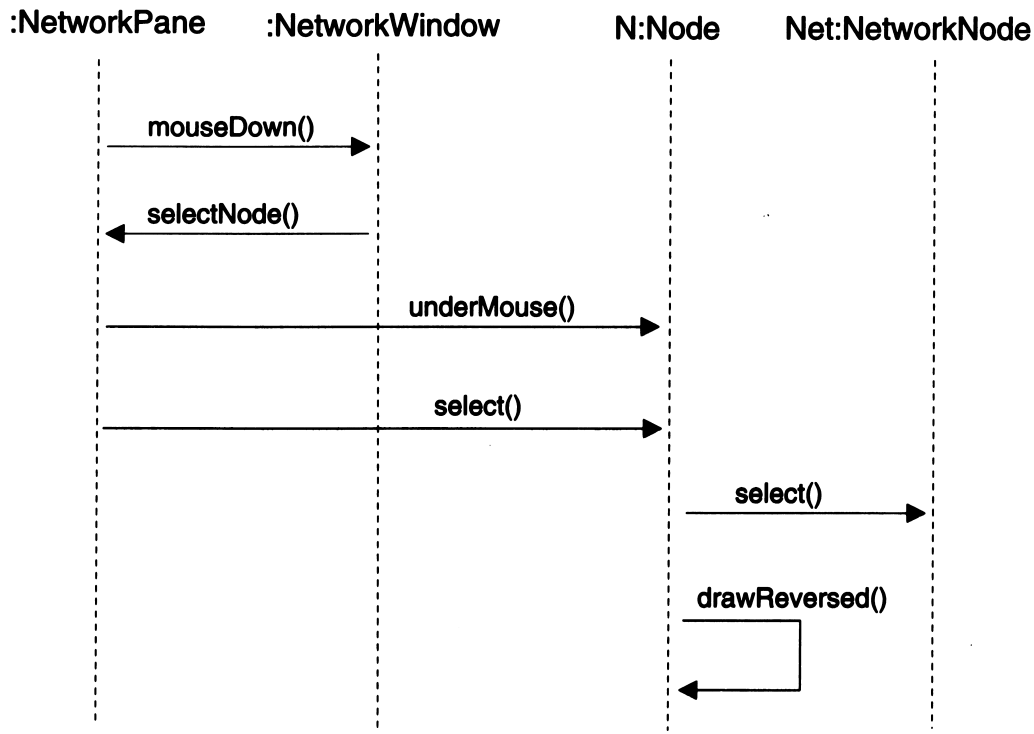


Figure 5-5
Node Selection Event Trace Diagram

3. The figure below captures *abstract class, virtual, friend, static* properties; it also shows *export controls* on the `Vehicle` class. It depicts *multiple inheritance*. It shows an *aggregation* relationship, where the sales goal component has *cardinality* of one and is *physically contained by value*. It shows a *using* relationship, *nesting* and a *constraint*.

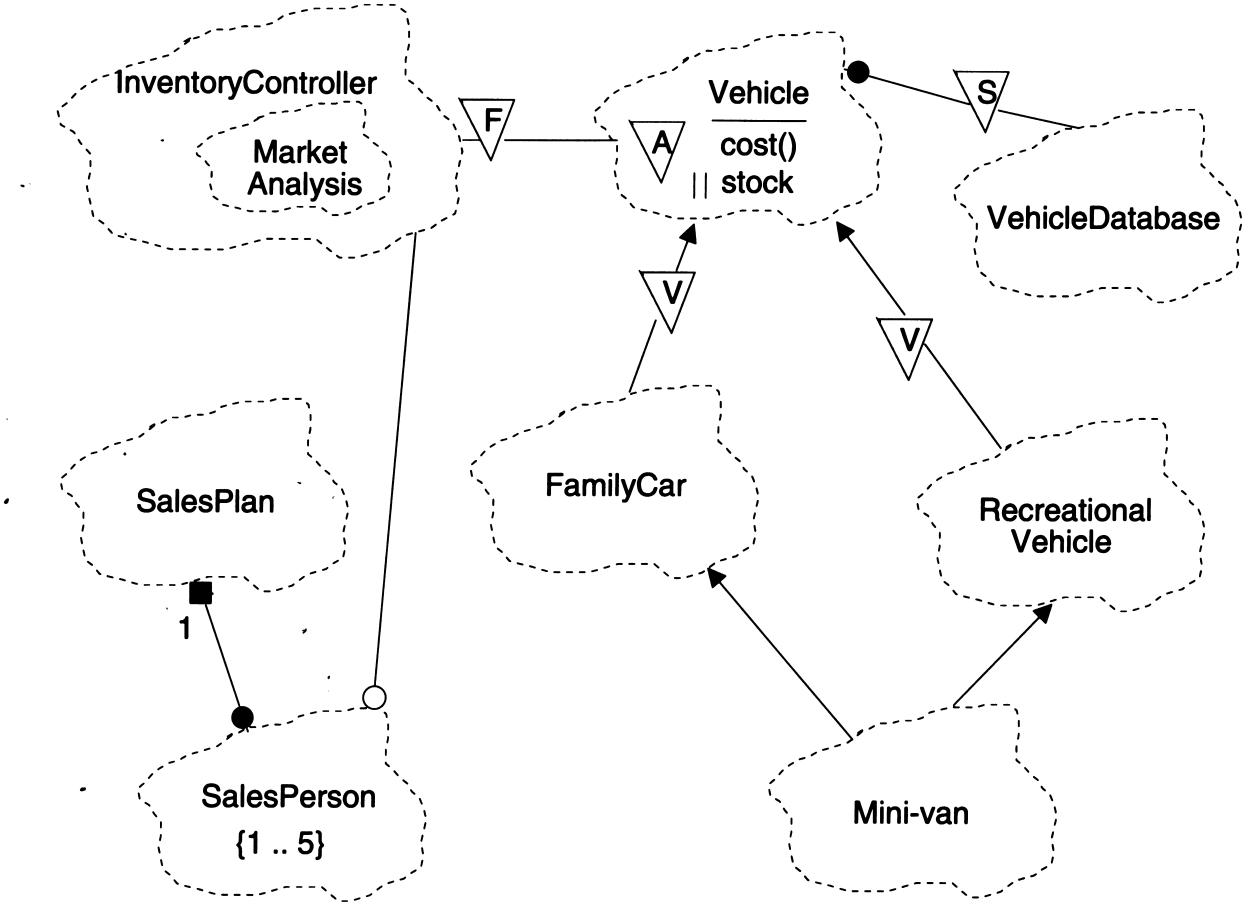


Figure 5-6
Car Dealership Class Diagram

4. The diagram conveys information about the concrete classes used in the system and their inheritance and aggregation relationships.

   a. *concrete classes* — the game uses instances of `Blackjack`, `BjPlayer`, `BjDealer`, `CardDeck`, `Card`, and `BjHand`.

   b. *inheritance* — `CardGame` is an abstract superclass of `Blackjack`, `CardGamePlayer` an abstract superclass of `BjPlayer` and `Dealer` in is a (non-abstract) superclass of `BjDealer`.

      The diagram also shows an elided view of some of the classes' attributes and operations. It has been specialized for the C++ implementation environment, using the `pure virtual` terminology to signify that `CardGamePlayer::bet()` is an operation that must be implemented by subclasses. In contrast, the `Dealer::deal()` and `Dealer::shuffle()` may or may not be redefined by subclasses.

   c. *aggregation* — We can see that instances of `Blackjack` contain exactly one instance of `BjDealer` and one or more instances of `BjPlayer`; instances of `BjDealer` hold exactly one instance of `CardDeck`, and instances of `CardDeck` hold from 1 to 52 instances of `Card`.

5. The graphical representations are relatively compact, as they use spatial relationships to convey some of what must be said in text. Overall, the diagram also conveys a more wholistic sense of the interacting classes, a global view which must otherwise be created mentally by working out the relationships in the text. Because distinct graphical characteristics are associated with different aspects of the classes and their structure, a viewer can easily focus on one dimension (e.g., aggregation, export control, inheritance) at a time. These dimensions are distinguished in text only via terminology — thus even though the graph and text are both physically two-dimensional, the graph uses notational conventions to add dimensionality.

   The main advantage of the text is that it is familiar — we are used to reading abstract descriptions of situations. It uses the terminology that we are trying to learn (e.g., "virtual inheritance"), whereas for the graph we must learn the mapping from notation to concept to make sense of it. And although the notation has been carefully developed and iterated, some aspects of it will seem arbitrary, and as a complete notation it is quite complex.

6. The state transition diagram depicts four top-level states: Getting Contents, Idle, Modifying and Undoing. Modifying has three substates, Interpreting, Executing, and Re-displaying; Undoing has the three substates Start list, Undos ready, and Reversing.

   The TextEdit object begins in Idle. When a file request event occurs, it moves to Getting Contents and a file opening operation is invoked on entry into this new state. If the file is successfully opened, a fileOk operation is evoked on exit, causing an Ok event and a return to Idle. An editing command moves the object to the Interpreting substate of Modifying; the command's successful resolution moves it to Executing, and the completion of execution moves it to Re-displaying. At this point, an Ok event returns the object to Idle, but a not Ok event moves it to Undoing.

   The first time the Undoing state is entered, an undo list is constructed. Once the list of available undos has been constructed, the object is in the Undos ready state; popping an undo item then moves the object to the Reversing state, and the reversed event moves it back to Undos ready. From there, an Ok event moves the object back to Idle. The history adornment of the Undoing state assures that the original undo list will be constructed just once, and that subsequent undo efforts will begin in the Undos ready state.

7. Figure 5-3 shows four modules which include both specifications and body (chemical analysis, graphs, math models, chemical profiler), and two modules which merely provide needed shared definitions and so have no code body (chemicaldefs and mathdefs).

   According to the diagram, the chemical analysis body depends on the specifications of the chemical profiler, the graphs and the math models, as well as on the chemical definitions. The bodies of both the graphs and math models depend in turn on the math definitions.

The Process

Discussion Questions

1. What are the five levels of process maturity? What are examples of activities or structures in a development process that might be used to assess process maturity?
2. Most development organizations are not very good at following a prescribed development process. Why is it important to have an acknowledged process anyway?
3. What constitutes the macro development process? Micro development? What are the points of contact between the two processes?
4. What is the role of prototyping in object-oriented development? When if at all would it be reasonable to "productize" a prototype? Why?
5. Many of the macro process recommendations are general in nature. Give examples of recommendations that apply equally to traditional and object-oriented development situations. Give examples of other recommendations that seem to be specially relevant to the needs of object-oriented development.
6. An important tenet of object-oriented design is that a system's behavior must be defined first. How do the macro and micro development processes described here reflect this approach to design?

Exercises

1. Consider the following object-oriented development situation. What changes would you make to increase the likelihood of a good result?

   A small software house has been awarded a contract to build a multimedia home care system for a large community hospital. The designers were selected because of their demonstrated expertise in multimedia technology — they have recently begun to market a comprehensive library of C++ classes for storing, retrieving, transmitting and manipulating multimedia documents. They have little experience in building applications, but see this as an opportunity to expand their business. They hire two new designers, one an applications expert from the health field (in the area of patient tracking), another a user interface designer. Neither of these new members has experience with multimedia, but the pre-existing team members are already confident of their expertise with this technology. The applications expert is sent out to meet with hospital staff and patients, to scope out what the application will provide. The user interface designer is given some sample multimedia documents and begins working on interaction techniques for retrieval, browsing and navigation. Meanwhile, the original designers use their existing library to experiment with different document formats.

2. One of the important scenarios for the services subsystem at Fred's Dodge dealership is service scheduling, for example, setting up a 40K service for Sue Jones for next Monday. The developers of the system have identified six main abstractions as participants in this scenario — `WorkOrder`, `AppointmentBook`, `WorkDay`, `Customer`, `Vehicle`, and `ServiceRequest`. Walk through a scheduling scenario, and create an initial distribution of responsibilities among these abstractions.
3. In the blackjack game discussed in earlier exercises, the dealer has several responsibilities: he manages the card deck, dealing new cards as needed, he determines the winner, and he collects and pays off bets made by players. Suppose that as part of the game a `CardDealer` superclass has been created with the following abstract interface:

   `prepareDeck()` — obtains new shuffled deck of cards

   `dealHands()` — existing card hands are replaced with fresh ones

`dealDown()` — a single card is dealt face down

`dealUp()` — a single card is dealt face up

`payOut()` — a particular player's payoff is calculated and distributed

`collect()` all losing bets are collected

As a client of this class, the `BjDealer` subclass is able to use these operations as-is, simply using its own state (e.g., number of players, how many cards in a hand) to contextualize the operations for blackjack. It also adds protocol for managing turn-taking and for determining a winner.

How would you change the protocol for the `Dealer` class to improve its reuse potential? Why?

4. Suppose that you are designing the architecture for the graphing subsystem of the chemical analysis system discussed in earlier examples. Give examples of some of the high-level policy decisions that might be critical to the success of this subsystem, explaining for each what its ramifications might be as the project evolves.
5. In exercise 4 from Chapter 4, we described a university course registration system. Imagine that you were part of a development team building an object-oriented version of such a system. Prepare an overview of the macro development process you would follow. Describe example artifacts you might produce as well as the measures you might employ to track the progress of your development work.

Projects

1. Throughout the chapter (and elsewhere in the book), Booch provides a number of heuristics to guide the different phases of the development process. Create a representation of the macro and micro development processes that summarizes the recommended steps and the measures of goodness that developers might use in evaluating their progress. Refer to other texts on object-oriented design and see if and how other methodologists' process recommendations can fit within the same scheme.
2. Consider one of your own system development projects (or if you like, interview another developer about a recent experience). How was the overall organization of that development project similar or different to the macro development process described here? Was there activity in that project that you would characterize as micro development? How was it similar or different to the micro development of objects and classes described here?
3. Return to the description of the macro development process for a university course registration system you created for exercise 5 above. Work through the process again, but try to develop some concrete artifacts as you would if you were actually building the system, e.g., CRC cards, a data dictionary in various stages, use cases, object and class diagrams, class specifications, module diagrams, and so on.

Sample Answers for Chapter 6 Exercises

The exercises represent design problems and have no single answer — any given situation can be analyzed in a variety of ways and to many different levels of detail. These sample answers reflect what we consider to be a reasonable analysis at a reasonable level of detail given the information available in the book.

1. The original team acknowledges the novelty of the problem they have taken on by hiring new personnel. However, rather than choosing new team members with broad experience who might be able to educate them as needed to appreciate issues concerning application structure and user interface (in particular, designers who have worked with multimedia applications in this area), they have chosen designers with narrow expertise, and show no intention of expanding or sharing their own expertise in multimedia. The team should have hired new members of the "short fat men" variety and set up a mutual education process whereby all team members could develop a broad view on the project.

   The narrowness of the members' perspectives is compounded by the process they have begun, in which the initial work on the problem has in a sense been "subcontracted" to the different problem areas (user interface, application, multimedia). It is too soon in the process to modularize the activities — especially early in the design's conceptualization, it will be important to work together as a group, to prevent the development of design conflicts later on.

   Overall, there is no sign of any particular vision, nor of any controls on the process. The software house seems to have taken strategies which may have worked well during their last few years of refining a class library (letting individuals or small groups set up and manage their technical contributions), and generalized them to this very different, more open-ended situation. The team needs to develop a shared vision of all parts of the system and to agree on a process whereby the vision will be achieved (e.g., what functionality will be in an initial prototype, how that will be evaluated, how such evaluation results will be incorporated into future work).

2. The responsibilities for the scheduling scenario might be distributed as follows:

   a. `WorkOrder`: obtains needed information about the customer requesting the service, the vehicle being serviced, and the service requested. Works with customer and service request objects to acquire this information.

   b. `AppointmentBook`: holds a collection of work days; knows the current day and can access its days in various ways (e.g., next Monday, a week from today); works with its work day(s) to schedule individual service requests

   c. `WorkDay`: holds time slots; assesses free time; adds and removes appointments

   d. `Customer`: provides detailed customer information (address, contact info, vehicles) given customer name and phone

   e. `Vehicle`: provides detailed information about a vehicle (model, year, identification number, license plate)

   f. `ServiceRequest`: provides detailed information about a car service (e.g., activities, time required, estimated cost)

3. The protocol for `Dealer` might be OK for the blackjack usage situation, but the operations are not primitive enough to be more generally useful. In particular:

   a. `prepareDeck()` should be broken into two more primitive operations, getting the new deck and shuffling it. Not all card games may need a shuffled deck, and those that do may want the shuffling to be done multiple times.

   b. `dealHands()` should be broken into two more primitive operations, discarding existing hands and dealing new hands. Different card games may have different rules about whether and how old hands are discarded.

   c. `payOut()` should be broken into two more primitive operations, one the calculation of the appropriate amount of money and the second the actual distribution of it. Card games may vary on when the actual money distribution occurs.

The other operations seem reasonable. However, the class seems to be missing a primitive operation that might be expected by many clients, that of cutting the deck (i.e., after a shuffle).

4. There are several general architectural concerns which should be addressed via common policies. An important example is the structure of the user interface. One approach would have that the user interface components (the different graphical representations) are largely complex views onto model data, and that the interactions with that data will be managed by generic interaction techniques (like buttons, menus, and so on). The advantage of this would be to simplify the architecture, essentially getting rid of the "controller" part of the MVC paradigm. The cost would be reduced flexibility as the system evolved — the users and designers might decide, for example, to use the graphical representations as a means for controlling their analyses as well as viewing them, and it would be difficult to return and re-engineer this interactive flexibility.

   Another important concern will be how best to manage the tradeoffs between computation and space. The users will be manipulating complex visual representations (e.g., rotating 3-D surfaces), and the architects will need to develop policies concerning the generation and buffering of the graphical primitives, taking into account the likely physical processors and devices that will be used to run the system.

   A third area of concern will be synchronization. Many chemical analyses will involve a dynamic process over time. The architecture must decide how to interface to these dynamic processes — one option is to "run off" the analysis and then let the analyst explore a stored version of the analysis. This simplifies the architecture, in that the analysis can simply be carried out and the results stored for viewing. But another option would be to allow the dynamic process to be viewed in "real time" as it is carried out. This option would necessitate control policies regarding when analysis updates were allowed, where the initiative for updates comes from, whether the process itself can receive input and thus can be manipulated in real time. However, by developing such policies, considerably more flexibility in the analysis-graphing interface would be achieved.

5. Following Booch's recommendations, the macro process would have five phases:

   a. *establish core requirements*: Here the goal would be to develop a shared vision of the basic functionality to be offered by the registration system. This vision would probably be documented via one or more prototypes, with each prototype given its own set of goals and schedule. First, for example, we might spend a week creating a simple abstract wrapper around an existing database of courses, just to ascertain that we would be able to work with the earlier code. In parallel we might prototype a course browsing scenario, say in Hypercard on the Mac. Measures at this point are simply that we are able to create the prototype (i.e., interface to the existing database), and that team members are able to understand and agree on the ideas embedded in the prototypes. We might also use these prototypes to gain approval and/or resources from the university administration for the project.

   b. *model system behavior*: Here we seek to create a model of registration system behavior. We want to first identify problem abstractions and their relationships to one another — entities like students, teaches, courses, and so on. Later we will want to invent additional entities to provide the mechanisms for the problem objects to work together — abstractions like a "validation" object that identifies a particular user as a registered student. During this phase we would be primarily concerned with enumerating, elaborating, and documenting use scenarios (e.g., via object diagrams and state transition diagrams).

      An example scenario might be the browsing scenario mentioned above. We would identify participating objects (e.g., the student, some set of courses, course instructor, a current schedule and so on). We would then begin to assign responsibilities to these objects (e.g., the student tracks prior course history as well as maintaining a current set of educational goals; course objects describe their contents, requirements and so on). Given these responsibilities, we would work out the detailed collaboration for this scenario (e.g., a course object receives the browse request, followed by a check prerequisites request, at which point it checks with the student object to see if its requirements have been satisfied; the teacher object then receives a browse request). We might also consider variants of this scenario (e.g., one in which the course receiving the initial browse request already has a full course roster).

      Scenarios such as these would be developed and presented to team members, including potential users of the system, so that all can understand and agree on the proposed behavior of the system. A successful

presentation will be one where the scenarios are simple and well-understood, and where there are no glaring holes in coverage.

c. *create an architecture*: Once we have a complete analysis of system behavior, we will design an architecture that supports this behavior, documenting the architecture via class category diagrams and module diagrams. At the same time, we will be developing high-level policies that will guide subsequent implementation activities. So, for example, we might organize the logical architecture into class categories such as Persons, with Students and Teachers nested within it; Course database; Graphical Browsing Interface; and Scheduling Requests. The module organization might have a similar structure, but in addition reflect a substructure in the Course database subsystem reflecting the object-oriented wrapper around the existing course database; it might also decompose the browsing interface into submodules reflecting the database queries versus the presentation and manipulation of course descriptions. We would assess the goodness of this architecture by developing a restricted prototype that instantiates the models.

   At this time we would also be developing high-level policies to be shared across the implementation. So, for example, we would develop a general protocol for accessing the existing course database, including a specification of how retrieval failures would be handled. We would also plan a series of releases based on this architecture, perhaps initially building a system that only has browsing facilities, followed later by one with the actual scheduling functionality; we might also implement the browsable information incrementally, beginning with just course descriptions, and then adding access to student and professor records.

d. *evolve the implementation*: Here we would begin the iterative process of actually building the system code. We would follow the releases planned during the design stage, and implement the architecture documented during that stage. At the same time that we are building these releases, we may be producing more limited prototypes to test issues that have been developed (e.g., working with the users to understand the implications of course database access delays). After each release, we will get user and performance feedback, and use this information to adjust our release plan as necessary (e.g., delaying addition of student and teacher information until we have worked out the bugs in accessing the course database). We will assess our success by how well each release meets its goals, as well as by noting changes in our defect discovery rate.

e. *manage post-delivery evolution*: once we have carried out our release plan, and have a stable system in place, we will maintain the system to address lingering bugs or to enhance it with new function. Thus, we might at some point add to the system a facility for generating a "hot courses" list, so that students can use course popularity (for the current or for prior semesters) as a factor in making course selection. We will track the success of these efforts by noting the ease with which the new function or bug correction is admitted into the existing architecture.

CHAPTER 7

Pragmatics

Discussion Questions

1. What is the biggest problem in task planning for the macro process? How can a development organization deal with this problem?
2. An incremental and iterative approach can be applied to any development project. What makes this approach especially appropriate for object-oriented development? What aspects of the object-oriented paradigm facilitate this style of development?
3. How is staff management affected by adoption of an object-oriented development process?
4. What changes in the nature and timescale of system releases are to be expected when an organization moves to object-oriented development? How do these changes impact testing activities?
5. Should a development organization use traditional productivity metrics in evaluating the progress of an object-oriented project? Why or why not? What alternatives exist?
6. What performance issues are associated with object-oriented development? What are some approaches to dealing with these potential problems?

Exercises

1. Suppose that you are the manager of a group of 10-12 developers, and that you have decided to "take the plunge" into object-oriented development for your next project. Describe the steps you would take to manage this transistion.
2. You are a developer involved in maintenance activities for the hydroponics farm system used as an example in the book. The accounting data indicate that more money than expected is being spent on growing tomatoes, and your job is to track down the cause of this mismatch and correct it. What documentation would you search for to help you in this analysis, and how would you use it?
3. Consider the following description of an organization considering the adoption of object-oriented technology. What organizational changes are they likely to need to be successful in this? How might such changes be implemented?

   WorldView Applications, Inc. has been in business for 15 years. It has two main divisions, Business Applications and Scientific Applications. The managers of these two subdivisions joined the company together, and over the years have become great competitors. They keep up with each other's hiring and firing activities and try to match or better the other's latest recruiting successes. The main productivity measure has always been lines of code produced, and each division is evaluated against this metric. This has led to an implicit demand for proposing and building large development projects — the managers of large successful projects producing hundreds of KLOC are rewarded by being allowed to grow even more.
4. Many artifacts of software development can be reused, and one of the contributions of object-oriented development is in fostering reuse at many different levels. Within the context of the chemical analysis system discussed in earlier exercises, what elements of reuse might you expect to find? Give examples of each type.
5. As the "toolsmith" of a small software house just beginning to experiment with object-oriented technology, you have not been given the resources to purchase a sophisticated development environment. However, you do have a text-editor that you built for general use, and are considering enhancing it to better support object-oriented development. What are some of the enhancements you might consider?

Projects

1. Supporting the development and incorporation of reusable components is an important element of successful object-oriented development. Develop a set of techniques that an organization might use to encourage the development of reusable code and describe how these would fit into (or change) existing practices and methods. Then do the same for encouraging the reuse of others' code. Finally, develop a specification for one or more tools that might aid an organization in applying the techniques you have proposed.
2. Choose an object-oriented system for which you have access to the class structure (one you wrote yourself would be best, but if not, try to find a colleague willing to provide access to a system; or perhaps you can get access to a publicly-available class library). Apply the six metrics described in the chapter to three or four classes from different subhierarchies in the class structure. How do your results compare across classes? Across classes in different subhierarchies? Do your results seem related to the kind of functionality being provided by a class?
3. Investigate two or three different object-oriented programming environments. Contrast the tools provided by the different environments, using as a framework the tool requirements for object-oriented development discussed in the chapter. What does this suggest about the development situations for which the different environments would most appropriate?

Sample Answers for Chapter 7 Exercises

The exercises represent design problems and have no single answer — any given situation can be analyzed in a variety of ways and to many different levels of detail. These sample answers reflect what we consider to be a reasonable analysis at a reasonable level of detail given the information available in the book.

1. There are a number of things you might do to both facilitate start up and promote an effective development process. For example:

   - Finding a source of object-oriented expertise will be critical. Best of course would be to hire a new group member who has actually worked on (hopefully successful!) object-oriented projects in the past. If this is not possible, hiring a consultant might work, or finding a knowledgeable colleague somewhere else in the organization.
   - Set up a training program for your current staff. If you are able to bring in a new staff member or a consultant, take advantage of this expertise in training other staff. In learning about object-oriented design, the mentoring approach is likely to work best. Be sure to allow plenty of time for training.
   - Analyze the expertise of your current staff and consider what roles they should play in the project. Pay special attention to the roles that are new or are highlighted by this move to a new paradigm — e.g., the reuse engineer and the toolsmith. For the more senior developers (those with more architectural experience) focus their training on class design; for more junior developers focus their training toward the reuse of existing components.
   - Use your imported expert (new group member, consultant) to help you choose among development platforms (e.g., C++, Ada, Smalltalk). If others in your organization are already using an object-oriented platform, find out what they are doing and whether you might be able to reuse any of their work.
   - Once you have selected a platform, spend some time investigating tools and libraries that are available. Involve your expert in this, but as soon as possible bring in the existing team members in their new roles as toolsmith or reuse engineer, so that they will begin to feel ownership of these responsibilities.
   - As part of the training program, provide a range of examples (from simple to complex) for your developers to learn from. These might be developed by your expert for this purpose, borrowed from others in the organization working on object-oriented projects, or purchased from commercial sources. Note that these examples need not all be implemented in the development platform that you will be using, as much of what needs to be learned is at the analysis and design level and relatively language/environment independent.

2. We would hope that the developers had produced a variety of products during the development of the original system that would allow us to trace back to the decisions and rationale associated with the design. Tracking down and fixing the problem might run something like this:

   a. Look for a requirements analysis document that describes important system scenarios. Hopefully, one or more scenarios will involve the calculation of expected crop-growing expenses.

   b. Once a source scenario has been identified, look for object diagrams (or event traces) that document how that scenario is supported by the system. These diagrams will indicate the mechanism in play that needs more detailed analysis. In particular, we should be able to determine the kinds of objects that have the responsibilities for calculating a crop's predicted growing costs, and the messages that they exchange to do so.

   c. To investigate in more detail, look for class diagrams to understand the relationships among the key abstractions. These diagrams will point to issues of inheritance and aggregation which may not have been apparent in the object diagrams. So, for example, you might discover that cost-prediction is handled by an abstract superclass, and that it needs further specialization for the problem case you are considering.

   d. When the problem has been analyzed, use the processor and module diagrams to determine where the relevant specifications and implementations are located within the physical organization of the current system.

3. The description of this organization depicts a company caught up in "empire building" — the extended competion of the two high-level managers has produced a climate in which a primary goal is to build large sub-organizations so as to merit greater resource. This approach to team-building will not transfer well to object-oriented development situations, where you typically need fewer people even on complex projects. The fact that KLOC are used in assessing productivity also flies in the face of object-oriented development, in which the quality of the abstractions and architecture developed are much more important than the amount of code produced.

   The management climate must be modified. An obvious possibility would be to wrest control from the two high-level managers who have created the current situation. A "flatter" organization might reduce the competition and give individual managers more flexibility in creating groups that are not large but instead have a range of personnel who can fulfill the various staff roles required by object-oriented development.

   The other major modification must be to the way in which projects are evaluated and resources assigned. The new management team must investigate the use of quality measures such as defect discovery rate and defect density, and more appropriate quantity measures such as the number of working classes and modules. To be truly successful with the new paradigm, they must also develop techniques for rewarding reuse, for example, measuring the frequency with which a developer takes advantage of others' code, as well as the frequency with which code he or she develops is used by others.

4. One measure of an effective object-oriented development project is that reuse can be observed throughout the design. For the chemical analysis system, one might find the following sorts of examples:

   - The reuse of individual components — this is perhaps the most salient form of reuse, when one abstraction (class) is directly usable in more than one situation. A simple example would be graphical elements (e.g., a polyline) that is used in many different graphing situations. Note that this form of reuse also implies some reuse among related documentation, as a class or object diagram involving the reusable component(s) might be nested within a higher-level diagram of the client.
   - Reuse through inheritance — in this case, multiple classes are able to share operations and attributes defined in their common superclasses. An example might be within a `ChemicalReaction` hierarchy, in which the superclasses defined common characteristics and behavior (e.g., `addElement()`, `pause()`), while subclasses added new behavior of their own (e.g., `increaseHeat()`).
   - Reuse of protocol — individual elements of an abstraction's interface may be reused, to take advantage of polymorphism in the system. Thus, an abstract class specification may include an operation that is deferred for definition by subclasses. Here, even though no real code is being reused, a shared abstract protocol is structuring the code that is developed. Again returning to the `ChemicalReaction` hierarchy, the abstract class might include a pure virtual function `mixElements()` which is needed by all subclasses but assumed to depend on the sorts of chemicals and conditions present for a given type of reaction.
   - Framework reuse — this is similar to reuse through inheritance, but occurs at a higher level, with the framework's clients sharing behavior defined across a group of collaborating classes, not a single class. So, for example, our system is likely to include a framework for the rendering of complex surfaces that involves classes from the mathematical modeling domain as well as graphics, and any scenario in the system that includes surface rendering can incorporate this framework as a high-level design, specializing it as necessary.

     Related to the reuse of a framework is that of *mechanisms*. Here, the focus is on the shared patterns of collaboration used to carry out different scenarios, as documented via object and class diagrams. So, for example, the system may use a single shared mechanism for building and submitting a query to the chemical database.
   - Scenario reuse — the complete set of scenarios will exhibit considerable commonality. One way to see this is in the relation of secondary and primary scenarios, where the former can be seen as variants of the latter. So, for example, one might have a scenario in which a heated solution is observed over time as a primary scenario, and then a variant which is just the same except that at some point a critical

temperature is reached and an explosion takes place. These two scenarios share much in object structure and responsibility, but they vary in the outcome.

Scenarios may also have a nesting structure in which an overarching scenario is expanded into some of its subscenarios. For example, the analysis scenario above might be a "parent" to two subscenarios, one of solution construction and a second of results analysis.

- Reuse of policies — there are likely to be a number of policies which are shared throughout the system, policies which may or may not translate into class frameworks or some other form of shared code. So, for example, there may a policy about how to manage the synchronization of chemical analysis results with their display, or how to handle failures in retrieval from the chemical database.

5. In general, your goal would be to incorporate some form of intelligence concerning class and object abstractions into the editor. For example:

   - Adding a simple hypertext capability could help immensely in navigation from one abstraction to another. So, for example, you might set up different kinds of links — perhaps "superclasses" and "subclasses" which would prompt you to choose among any of possibly many classes connected via these links; a "client" link which would help you navigate from an interface declaration to a user of this interface component; a "part of" link which would move you from an attribute to the abstraction it represents; an "implementors" link to travel from a specification to its implementation (perhaps distinguishing between implementation within this class and implementations provided by other classes within a subhierarchy or even anywhere in the system).

     A cost in this is the creation of the links, which in the simplest version would be up to the developer. Note, however, that the links could be also be created automatically, if you were able to parse the structure of the specifications as they were created. Developing a tag language or set of templates as described below would facilitate the automatic generation of such links.

   - Developing templates or tags corresponding to the language with which you are experimenting would reduce the tedium of setting up declarations — e.g., for C++ you might set up automatically the specification of constructor and destructor functions, breaking up a class specification into public/protected/private, and so on.

   - In addition to allowing navigation from one abstraction to another, you might support simple information queries. Thus, rather than moving physically to a related class, you might simply want to see some summary information it (e.g., its public interface, the comments provided with it), or you might want to simply check a method implementation without leaving your current editing context. This could be handled via a simple dialog that presents a read-only summary of abstractions based on their structural characteristics.

Data Acquisition: Weather Monitoring System

Discussion Questions

1. What design rationale is associated with the `Sensor` hierarchy? Could `HistoricalSensor` have been set up as a superclass of `CalibratingSensor` instead of vice versa? Why or why not?
2. How are the keypad and display device abstractions similar? What implications do they have for the rest of the design?
3. The case study elaborates the primary scenarios for the system. What other scenarios would the system support? What implications do these secondary scenarios have for the system's design?
4. The `Timer` abstraction is an active object that employs the C++ callback idiom to initiate sampling. What are other mechanisms that could accomplish the same function (e.g., for other implementation platforms)?
5. What is the role of the `Sensors` abstraction in the system? Why does it have `Collection` as a protected superclass?
6. The first system release provided monitoring facilities for just a single sensor. Why? Can you imagine situational factors that might have led you to produce a different release plan?
7. Why were input states not modeled as objects? What additional design work would have been required to include these abstractions as objects in the system?

Exercises

1. The book describes a simple `TimeDate` abstraction used to provide an abstract interface to an embedded timing mechanism. Figure 8-2 in the book is a state transition diagram depicting the lifecycle of this simple object. Suppose that in addition to the timing mechanism the hardware included an "internationalization" switch that could be adjusted to indicate whether the system was being used in the U.S. (e.g., where dates are formatted with the month value first; where temperature is calculated in degrees Fahrenheit) or in Europe (where dates have the month value second and temperature is in degrees Celsius). Prepare a new version of the `TimeDate` state transition diagram to show the impact of this new feature.
2. The scenarios for displaying the highest and lowest value of a selected measurement and for setting time and date overlap, in that both begin with pressing the SELECT key. Expand the Selecting state in Figure 8-11 to capture the states and events relevant to these two scenarios.
3. Consider addition of a "comfort index" (another measure derived from temperature and humidity measurements) to the weather monitoring system. How would Figure 8-9 (showing the associations between the derived measures and the sensors themselves) be changed to reflect the addition of this measure to the system?

   What other consequences would this addition have to the design of the system? Include notation diagrams as relevant to describe and document the system modifications.
4. Using the system structure and interface information provided throughout the chapter, develop an object diagram documenting the user's display of the high temperature for the day.
5. The comfort index addition analyzed in Exercise 3 is a system enhancement easily accommodated by the system architecture (as are the rainfall and report downloading enhancements described in the book). Describe one or more enhancements that would require re-engineering the architecture. Include in your description the nature of the re-design required.

Projects

1. The design of the weather monitoring system presented in the book employs the basic mechanism of time-frame-based processing to manage the sampling of the various sensors. An alternative architecture could have employed an active sensor mechanism. Work back through the case study described in the book, re-engineering the weather system to employ this alternative mechanism, documenting your changes with diagrams or class specifications as necessary. What new scenarios are suggested or enabled by your new design?
2. Choose another example of a data acquisition system (examples include patient monitoring in the hospital or at home, the sensor subsystem of the hydroponics farm, the dashboard of a car). Develop a case study at a level of detail similar to that provided for the weather monitoring system. Be sure to analyze the boundaries of your system before you begin, and to use release planning to organize the more detailed design.

Sample Answers for Chapter 8 Exercises

The exercises represent design problems and have no single answer — any given situation can be analyzed in a variety of ways and to many different levels of detail. These sample answers reflect what we consider to be a reasonable analysis at a reasonable level of detail given the information available in the book.

1. We would need to build another wrapper around this new piece of hardware and introduce it as a collaborator in the `TimeDate` initialization process. The consequence for the state diagram is that the Initializing state now has a more complex structure as in Figure 8-1 below.

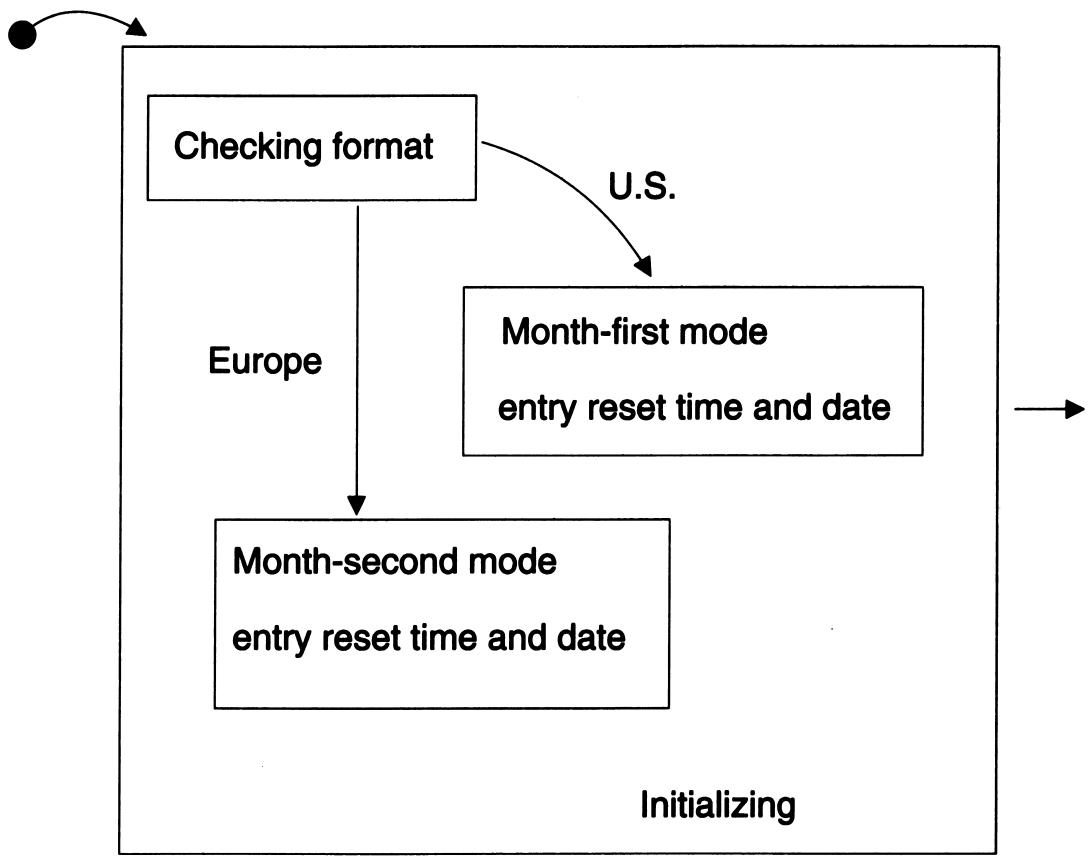


Figure 8-1
Partial TimeDate State Transition Diagram

2. The Selecting state has two major substates within in it, Displaying and Setting. These two substates break down further as follows:

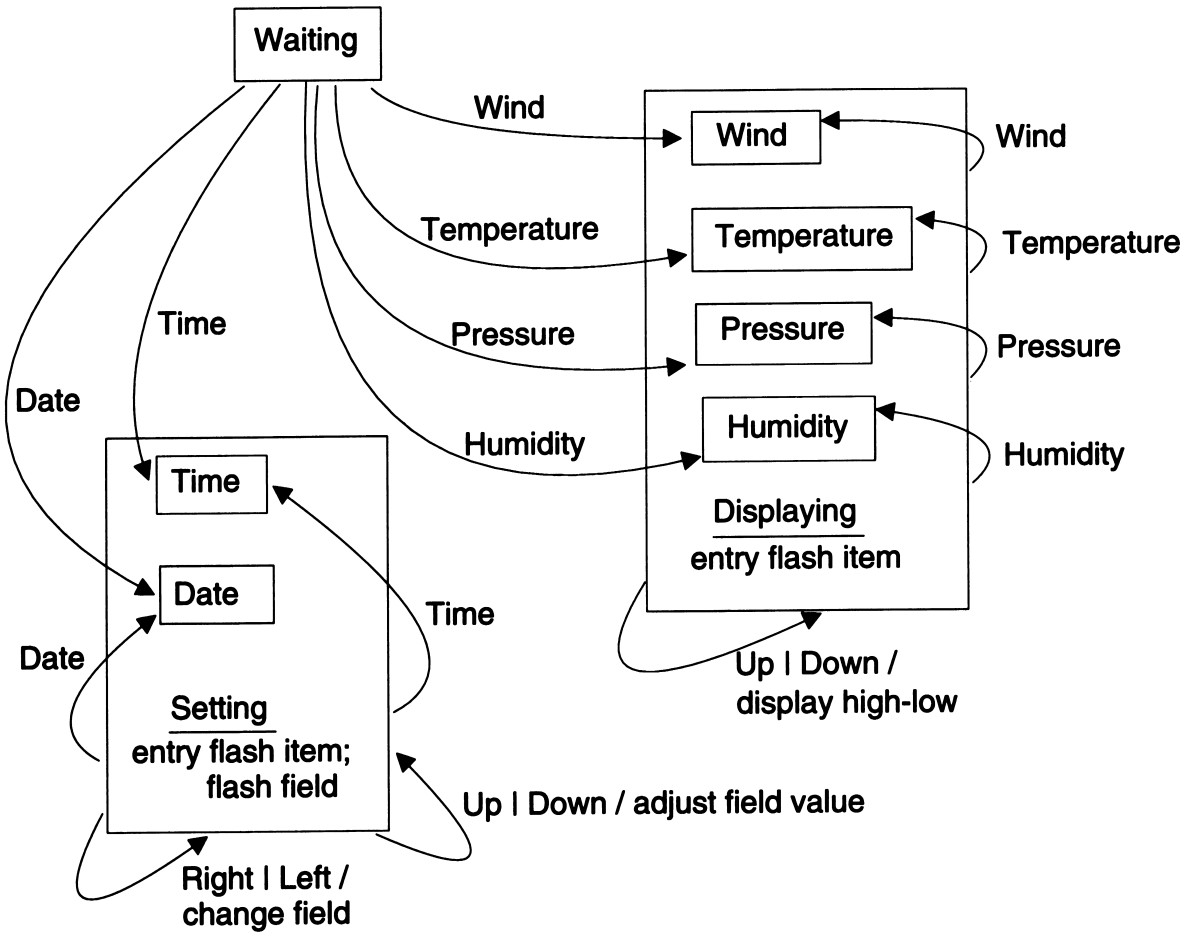


Figure 8-2
Expansion of Selecting State

3. The new measure would have exactly the same usage links as the dew point measurement, causing the diagram to look as follows:

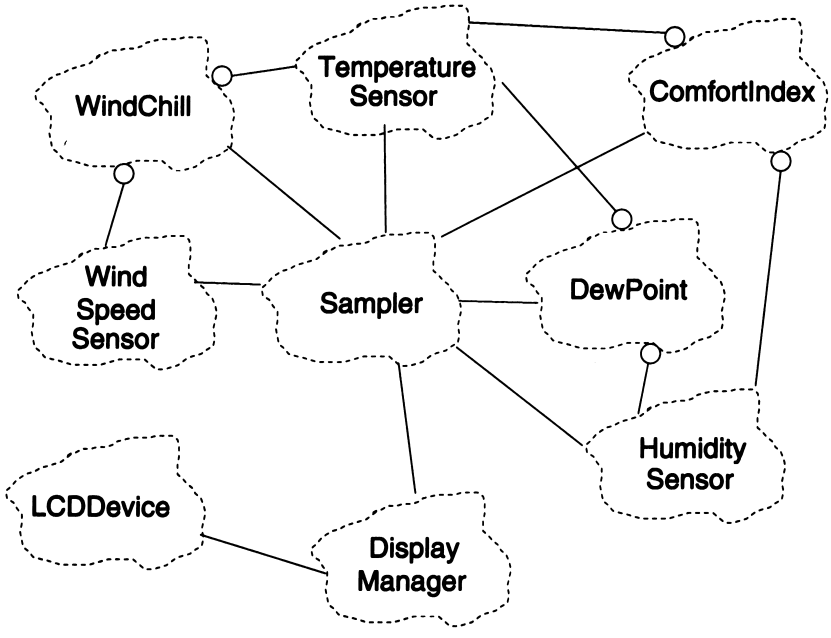


Figure 8-3
Derived Measures Class Diagram

The new measure qualifies for abstraction as a new kind of object in the same sense as dew point and wind chill — it has its own behavior (calculation of the value) and encapsulates state (the two sensors it uses). Because of the overlap between the new `ComfortIndex` and `DewPoint`, we would probably want to introduce an abstract superclass to unify the abstractions. The new structure might appear as follows:

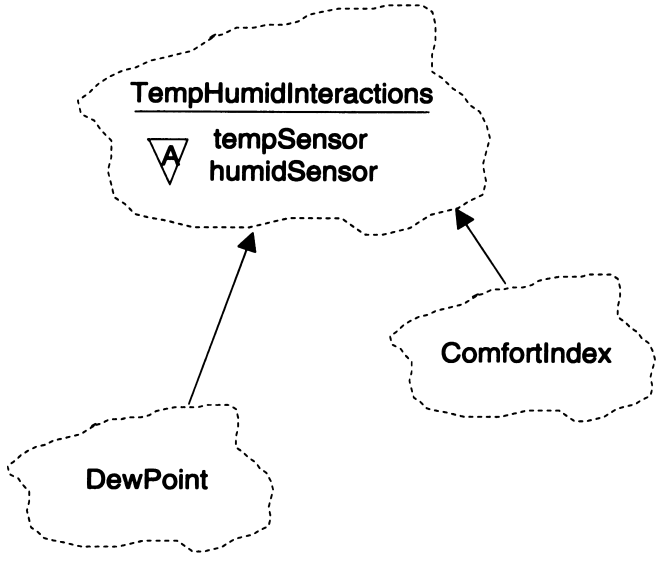


Figure 8-4
Temperature-Humidity Interactions Class Diagram

Note that this introduction of a new superclass could be expensive if this was a situation of enhancement maintenance. A less-than-satisfactory compromise would be to simply add a new operation to the `DewPoint` class, `calculateComfort()`, as this class already has access the relevant state information. We might also decide to simplify the problem explicitly, and eliminate both the `DewPoint` and `ComfortIndex` classes, providing both derived measures via the higher-level `TemperatureHumidityInteractions` class.

Other changes to the design would include expanding the scenarios to include selection of this new derived measure, adding an instance of `ComfortIndex` to the collection held by the `Sensors` instance, and adding a new display function to the `DisplayManager` class (e.g., `displayComfort()`).

4. This is a Selecting scenario. The scenario is controlled by the unnamed `Sampler` instance, whose `processKeyPress()` is evoked whenever it receives an interrupt. It then collaborates with its member objects to interpret key input and to display the result.

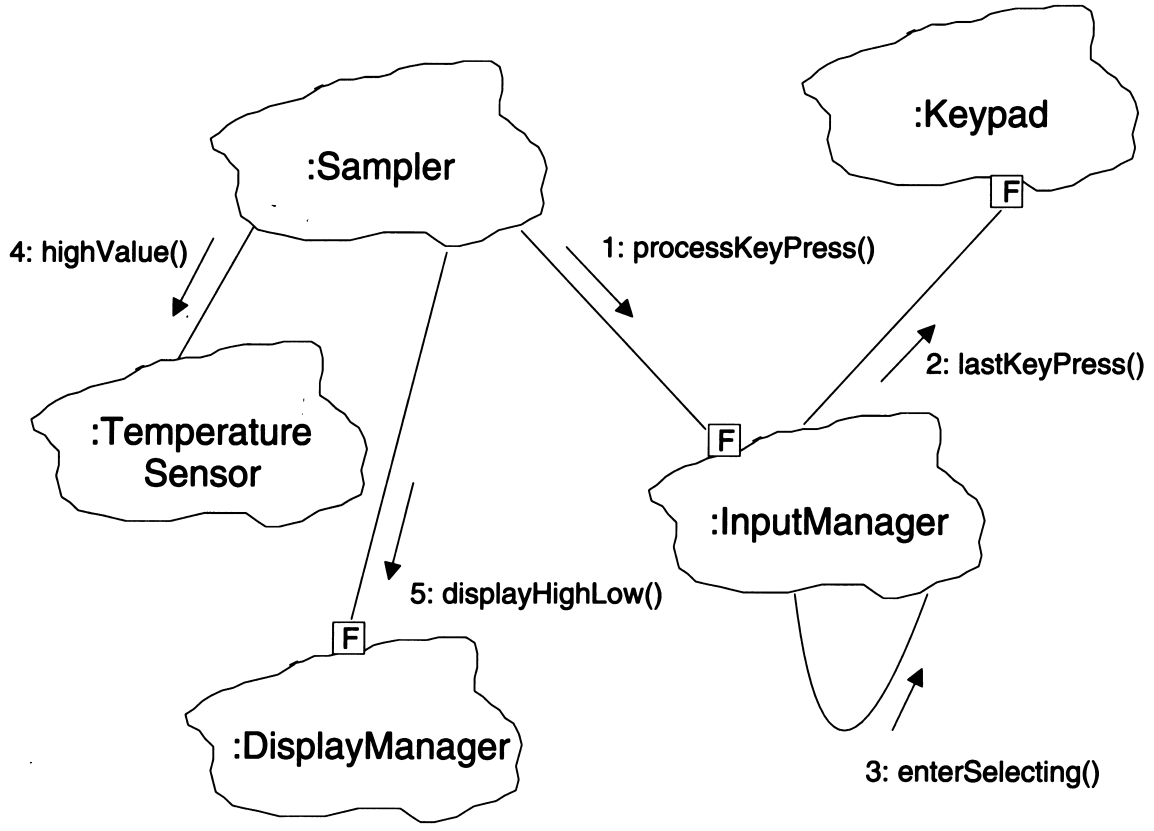


Figure 8-5
Object Diagram for Displaying High Temperature

5. One example of an enhancement requiring major architectural changes is that suggested by the book (and raised as a project below) — a version of the system whose sensors are active objects. As noted in the book, such an architecture would be needed if the system were enhanced to support distributed processing, with sensors deployed at different physical locations. Changing the system to work in this way would involve a change to the main processing loop, where instead of being time-interrupt-driven, it would be sensor-interrupt-driven. The `Sampler` abstraction would probably disappear altogether, as the relevant sampling knowledge would now be encapsulated within the individual sensors.

Other enhancements having major implications for the system architecture would be any involving modifications to the class abstractions developed for the sensor objects. Suppose, for example, that new technology

became available such that the temperature and humidity sensors were self-calibrating, but could report on their calibration history. This would introduce a new high-level abstraction (e.g., `SelfCalibratingSensor`) that must be positioned correctly within the existing hierarchy, and would require the re-positioning of two of the concrete classes to inherit from this line. Multiple inheritance concerns would arise as well, in the course of making the history and trend abstractions available to both calabrating and self-calibrating sensors.

CHAPTER 9

Frameworks: Foundation Class Library

Discussion Questions

1. What is a framework? How would you characterize the essence of the framework embodied in the foundation class library described in this chapter?
2. The book describes two phases to the initial requirements analysis — surveying the theoretical literature, and analyzing production systems. The former is a top-down approach which can take advantage of existing well-known abstractions, but the latter is a more bottom-up process involving the discovery of possibly new abstractions. How would you carry out this piece of analysis?
3. The foundation classes are organized into class families, with several concrete classes inheriting from a single abstract base class. What are the benefits of this organizing principle? What downside(s) are associated with this design choice?
4. Why is the class parameterization facility of C++ so crucial to the development of this library? How would the library have been different if template classes could not have been built?
5. What issues must a user of the foundation library take into account when choosing among bounded versus unbounded forms of an abstraction? Among sequential, guarded and synchronized forms?
6. All of the structure base classes in this library have several common "helper" functions. Why is this? What are examples of how this has facilitated design and evolution of the class library?
7. What is the difference between the active and the passive iterator mechanisms? When would you choose one form over the other?

Exercises

1. Using the style outlined for foundation library classes, write a definition for the abstract `Collection` class. How does it compare to the example given for `Queue`? What will the subclasses of this abstract class be?
2. Developers who use the container classes in the foundation library must accept certain responsibilities. Take as an example a developer who wishes to make use of the `UnboundedQueue` structure and describe what he or she must do to use the class. What are the benefits of taking on these responsibilities?
3. The book provides a list of exception conditions addressed by the foundation class library, conditions identified through domain analysis. How might this domain analysis have been carried out?
4. Developers at times may need to track the space demands of the structures that they are using, perhaps to create a profile of a system's overall space requirements, or to analyze the needs of one or more specific types of structures (e.g., so as to make a more informed decision about storage management policies). How would you extend the foundation class library to include such a facility?
5. Each structure in the foundation class library provide guarded and synchronized forms for dealing with concurrency issues. Prepare a class diagram depicting the class relationships relevant to the use of the concrete class `SynchronizedBoundedQueue`.

Projects

1. This chapter is replete with nuggets of design rationale, from high level concerns about functionality and architectural policies, to very detailed arguments concerning specific mechanisms. Study the chapter carefully

and extract as many of these design arguments as you can, including the one or more examples to which they were applied (use textual descriptions, diagrams or actual class specifications to record the example depending on what seems most useful). Organize the arguments and examples into a "design heuristics" document. Then try it out on colleagues already familiar with object-oriented programming (and preferably with C++). Are they able to make sense of the arguments and examples you have extracted? Would they find it useful in their design work? Can you see ways to reorganize the information that would make it more useful?

2. Compare the foundation classes described here to those available for some other object-oriented platform (e.g., Smalltalk). Are there substantial differences in functionality, or are the differences largely in how the abstractions are organized? Can you see any implications for use of the two sets of classes? What situations would lead you to choose one over the other?

Sample Answers for Chapter 9 Exercises

The exercises represent design problems and have no single answer — any given situation can be analyzed in a variety of ways and to many different levels of detail. These sample answers reflect what we consider to be a reasonable analysis at a reasonable level of detail given the information available in the book.

1. The class definition might look as follows:

```
template<class Item>
class Collection {

public:
    // constructors
    Collection();
    Collection(const Collection<Item>&);
    // virtual destructor
    virtual  Collection();

    // operators
    virtual Collection<Item>& operator=(const Collection<Item>&);
    virtual int operator==(const Collection<Item>&) const;
    int operator!=(const Collection<Item>&) const;

    // modifiers
    virtual void clear() = 0;
    virtual void add(const Item&) = 0;
    virtual void remove(const Item&) = 0;
    virtual void replace(unsigned int at, const Item&) = 0;

    // selectors
    virtual unsigned int extent() const = 0;
    virtual int isEmpty() const = 0;
    virtual const Item& itemAt(unsigned int) const = 0;
    virtual const Item& first() const = 0;
    virtual const Item& last() const = 0;
    virtual unsigned int location(const Item&) = 0;
    virtual int includes(const Item&) = 0;

protected:
    // helper functions
    virtual void purge() = 0;
    virtual unsigned int cardinality() const = 0;
    virtual void lock();
    virtual void unlock();

    // friends
    friend class CollectionActiveIterator<Item&>;
    friend class CollectionPassiveIterator<Item&>;

};
```

This abstract class overlaps considerably with `Queue` — both have functionality for adding and removing elements from some collection. The main differences are in the abstract interface for using the elements, with `Collection` providing more general access to its elements.

Like all foundation abstract classes, `Collection` would have a two subclasses reflecting the boundedness dimension; each of these would then have two subclasses reflecting the synchronization dimension.

2. Because `UnboundedQueue` is a template class, users must instantiate it before they can create instances in their application. In this case there are two formal arguments which must be instantiated.

a. The user must declare an "Item" class. This determines the type of objects that can be placed in the queue. The use of the template form means that the actual determination of item type is deferred until run-time, which gives the user greater flexibility while still maintaining the benefits of runtime type checking.

   In declaring the type of item in this queue, it is the responsibility of the user to ensure that the item provides certain operations. In particular, it must have at least a default constructor, a copy constructor and an assignment operator, as well as a destructor that prevents the item from being destroyed immediately on removal from the queue. By fulfilling these responsibilities, the user can take advantage of the design decision to store item values not references in the queue, which permits the same class to be used safely across a variety of item types (e.g., built-in and user-defined).

b. The user must also instantiate the queue template with a storage manager class. In doing so, he or she must ensure that the storage management abstraction has the minimum protocol needed for participating in the storage management mechanism (i.e., it must provide an `allocate()` and a `deallocate()` operation). In return, the user is given the flexibility to use whatever management policy is appropriate to the problem and/or platform, and indeed can easily experiment with different policies.

3. As in the case of identifying the initial abstractions in the library, the analysis of exception conditions might have begun by searching the available literature for existing abstractions developed for handling exceptions; similarly, one might also examine some set of existing systems covering a wide range of applications, to extract commonalities among concrete cases of exception handling.

   With these externally-derived abstractions as a starting point, one could then engage in some scenario analysis on the class library, essentially concocting "what if" situations, and determining whether and how the existing set of exception conditions could handle the scenarios, extending or modifying the set as necessary,

4. There are two general design options which have varying impact on the existing library and provide differing levels of service.

   a. The simplest option would be to develop an abstraction serving as an "observer". The object would rely on some external device (e.g., a timer interrupt) to initiate a time-stamped size record for one or more structure instances with which it had been associated. Thus you might create a parameterized class (say `SizeProfiler`) with formal arguments for the Item and Structure classes. Instances of the tool might be initialized with a particular sampling rate, and perhaps a "significance" level (i.e., the minimum delta worth recording). This design implies addition of this class as a "friend" to all of the structure base classes, so that it would have access to the common `cardinality()` operation. It has the advantage of offering different levels of granularity in the profile constructed (e.g., sampling rapidly or slowly, depending on the character of the system being tracked), but because the sampling is initiated by an external device, the profiler might miss large changes that took place in-between samples.

   b. A more "invasive" option would be to develop an agent who works in collaboration with the structure instances to track space needs. The parameterized class would again have arguments for Item and Structure class, and would be added as a friend to the structure classes. Also as above, instances might be given a "significance" value to use in deciding whether to record a change. However, the initiation of the recording in this case would be the responsibility of the individual structures; a particular structure would notify the profiler at the start of any size-modifying operation (much like the existing exception handling mechanism). The cost of this alternative would be that individual modifying operations would need to be updated. The advantage would be that every change would be considered for possible reporting.

5. This queue variant inherits some functionality from its abstract `Queue` base class and its direct superclass `BoundedQueue`, with additional functionality provided through member objects defined for its superclass and for itself. This can be seen in Figure 9-11.

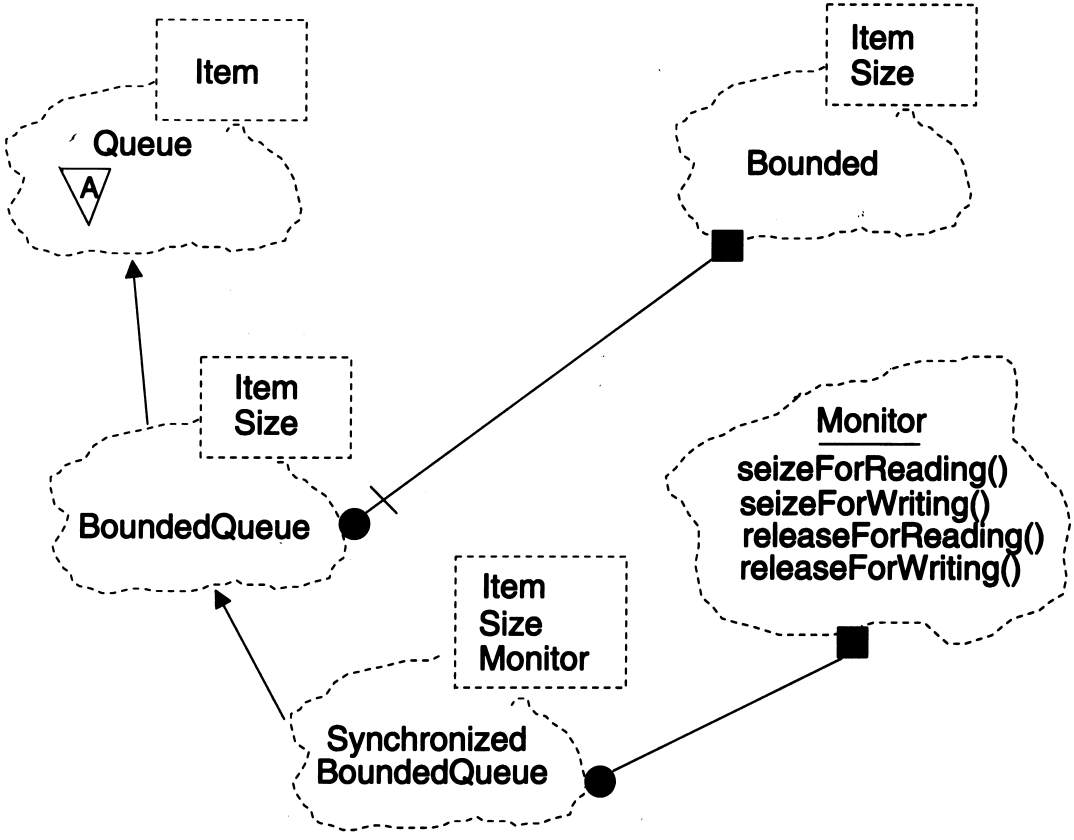


Figure 9-1
Class Relationships for SynchronizedBoundedQueue

Client/Server Computing: Inventory Tracking

Discussion Questions

1. A client/server architecture with multiple kinds of clients will normally use a variety of controls to ensure that clients have access to and can operate on only data for which they are authorized. What are examples of such controls relevant to an inventory tracking system?
2. Why is planning for evolution so important in the design of business systems? How do object-oriented systems address this concern? Can you give an example?
3. What are the main components of a client/server application? How does the architect decide how to distribute these components? What rules of thumb can be used as guidance?
4. If you are building an object-oriented system, why incorporate relational database technology? If you do so, what are design issues that must be considered?
5. What is a database schema? How should it be generated? What is the role of object-oriented analysis in this process?
6. What transaction processing problems arise for systems with distributed databases? What design solutions have been proposed?

Exercises

1. Prepare an event trace diagram for a scenario in which a customer calls up to add an item to her current order, but when the agent checks the inventory, he discovers that the item is not available and so initiates a replacement order with the appropriate supplier, informing the customer of the revised date of shipping.
2. The analysis of the inventory tracking problem includes scenarios involving planning and analysis (i.e., the generation of various reports), but the design developed does not elaborate this aspect of the system beyond the proposal of an abstract `Report` class. Use the example scenarios proposed during problem analysis to further develop report generation facilities: expand the scenarios enough to identify the key report classes and their responsibilities, documenting the resulting abstractions and their associations in a class diagram.
3. Consider the following description of an inventory tracking system in use over a period of several years. What seems to have happened? How should it be corrected?

   Fred's Dodge dealership began using an inventory tracking system several years ago. Initially, the analytic services in the system were quite slim — Fred occasionally reviewed sales performance of different car models, but most of the planning was based on the intuitions of Fred and his sales personnel. However, as the sales data built up, the sales personnel realized they were missing a golden opportunity to gain a competitive edge, and began commissioning various sorts of planning systems that allowed them to work with their sales histories — this enabled them to ask a variety of open-ended questions about customer characteristics related to sales of various sorts, as well as to make very informed guesses as to the kinds of cars to have in stock at different points during the year. However, as the sales personnel have become more fluent at spinning these "what if" scenarios, they also have become more frustrated at the system's apparent lack of responsivity. It seems that every simple little question and variation thereof takes a lifetime to hear about. They have been finding themselves repressing some questions that come to them, so as not to find themselves hanging around for an answer.

4. Using the SQL mechanism described in the chapter, create an object diagram illustrating a client asking for the current quantity of some item in the inventory.
5. Suppose that the inventory tracking system described in this chapter had been built from scratch — i.e., a new object-oriented database was built rather than an interface to a existing relational database. How would the design process have been different?

Projects

1. At the end of the chapter, several examples are given of possible enhancements to the inventory tracking system. Elaborate the enhancement involving the development of personalized catalogs describing items in the inventory, tailored to individual customers; assume electronic distribution of the catalogs. Develop scenarios to analyze the functionality that will be provided. Then identify new abstractions and their attributes and operations. Document your work with class and object diagrams. Be sure to indicate how the new service fits within the overall client/server architecture.
2. The East-West University's library system that has figured in various exercises in earlier chapters is another example of a system requiring a client/server architecture. Using the inventory tracking system described here as a model, work through the analysis and design of the library system. Try to find analogs of the analysis concerns addressed in this chapter in the context of the library system. Develop documentation for your design of the same sort and at approximately the same level as presented in the inventory tracking case study. Discuss the differences and similarities between the two problem domains. Are you able to reuse design information (at any level) from the case study presented in the book?
3. An important component of any client/server application is its GUI model, in that it sets up an architecture for the kinds of user interactions that are available, which in turn has implications for the kinds of tasks the user can carry out. Within a GUI, a crucial mechanism is its response to events. Gather information on a number of GUI architectures (e.g., those mentioned in the book — X Windows, Open Look, MS Windows, MacApp, NextStep, Presentation Manager), and compare and contrast their basic event handling mechanisms. See if you can find any differences among the event-handling architectures that appear to have implications for the kinds of user interactions (and application tasks) that will be handled gracefully by the architecture.

Sample Answers for Chapter 10 Exercises

The exercises represent design problems and have no single answer — any given situation can be analyzed in a variety of ways and to many different levels of detail. These sample answers reflect what we consider to be a reasonable analysis at a reasonable level of detail given the information available in the book.

1. The scenario would involve interactions with three databases, one in which the existing order is stored, a second with information about the current inventory, and a third with information about product suppliers. These interactions can be seen in the diagram below:

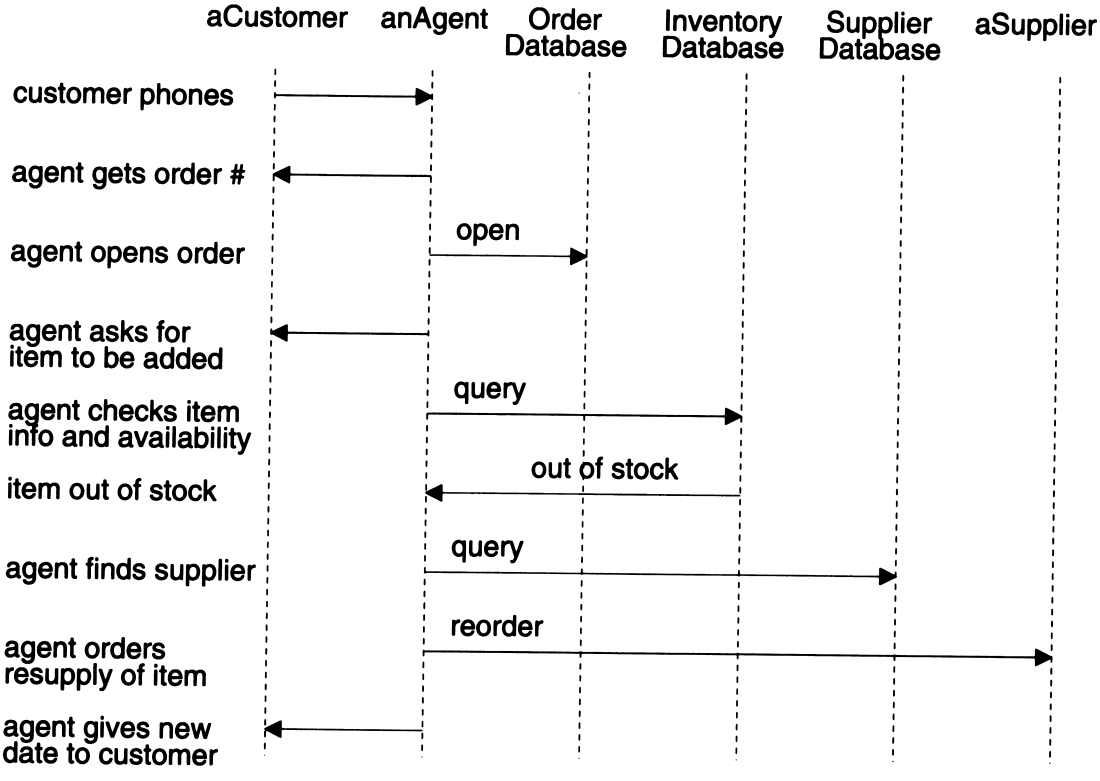


Figure 10-1
Order Addition and Resupply Scenario

2. There are three "reporting" scenarios suggested in the analysis: trend analysis on sales activity, summary of inventory levels, and revision of inventory summaries. In all of these cases, the "client" would be the planning application (i.e., the end-user is a planner in the organization). We shall consider each of these scenarios in turn.

   a. trend analysis — here, report generation would involve the application of various queries against the order database (i.e., creating and invoking instances of a `QueryTransaction`) For example, the planner may wonder about the sales for a particular type of product, over some particular time range, and broken up into a particular set of time intervals. The planner may be interested only in completed orders (in the `OrderTable`), or he may wish to see both completed orders and orders in progress (e.g., in the `PurchaseOrderTable`). Thus the scenario suggests a subclass of `Report` which is a `TrendReport` and which has behavior for setting and modifying time ranges and intervals and for setting and modifying product lists.

This trend report abstraction may be a simple summary of the raw data, or it may have additional behavior to create mathematical projections based on the raw data. If the latter, we would need to provide to the `TrendReport` abstraction the services of a `Projection` object whose responsibilities are to apply regression techniques to model the sales data and to project sales in the future. These projections might be based simply on sales over time or on more complex models (e.g., ones based on interactions among product types, or based on the customer information associated with the order records).

b. inventory summary — here, the report will also involve the application of queries, but this time to items in the inventory (in `InventoryTable`). The queries in this case are much simpler, involving just the identification of a product. As in the case of trend analysis, quantity information will be returned, but in this case the quantity in stock rather than quantity sold. The important behavior of the `SummaryReport` abstraction will be the organization of the report created. As suggested in the book, one summary might be produced for purposes of tax analysis. This implies that the report object has the ability to organize its queries as a function of tax-relevant product characteristics. The abstraction might either be a general one that requires the user to specify the model for the summary (giving it a behavior for setting summary dimensions), or the organization might decide to create a number of specialized summary classes, each with a built-in summary model (e.g., a `TaxSummaryReport` that knows about product characteristics relevant to tax planning). The latter is preferable, as it documents important pieces of the business model, ensures that this model is shared among different planner-users, and removes the need for these users to specify the information themselves.

c. summary revision — this reflects a secondary version of the scenario above, and its analysis depends on the decision made about specializations of the general summary report abstraction. If summary organization is specified by the user, then new summaries would be created simply by re-specifying the product dimensions in the original query. If a `TaxSummaryReport` abstraction has been created, then this implies that its knowledge about tax rules would need updating, giving it the additional behavior of setting its task-relevent product knowledge.

A class diagram capturing these abstractions and their associations with one another appears in Figure 10-2 (on the next page).

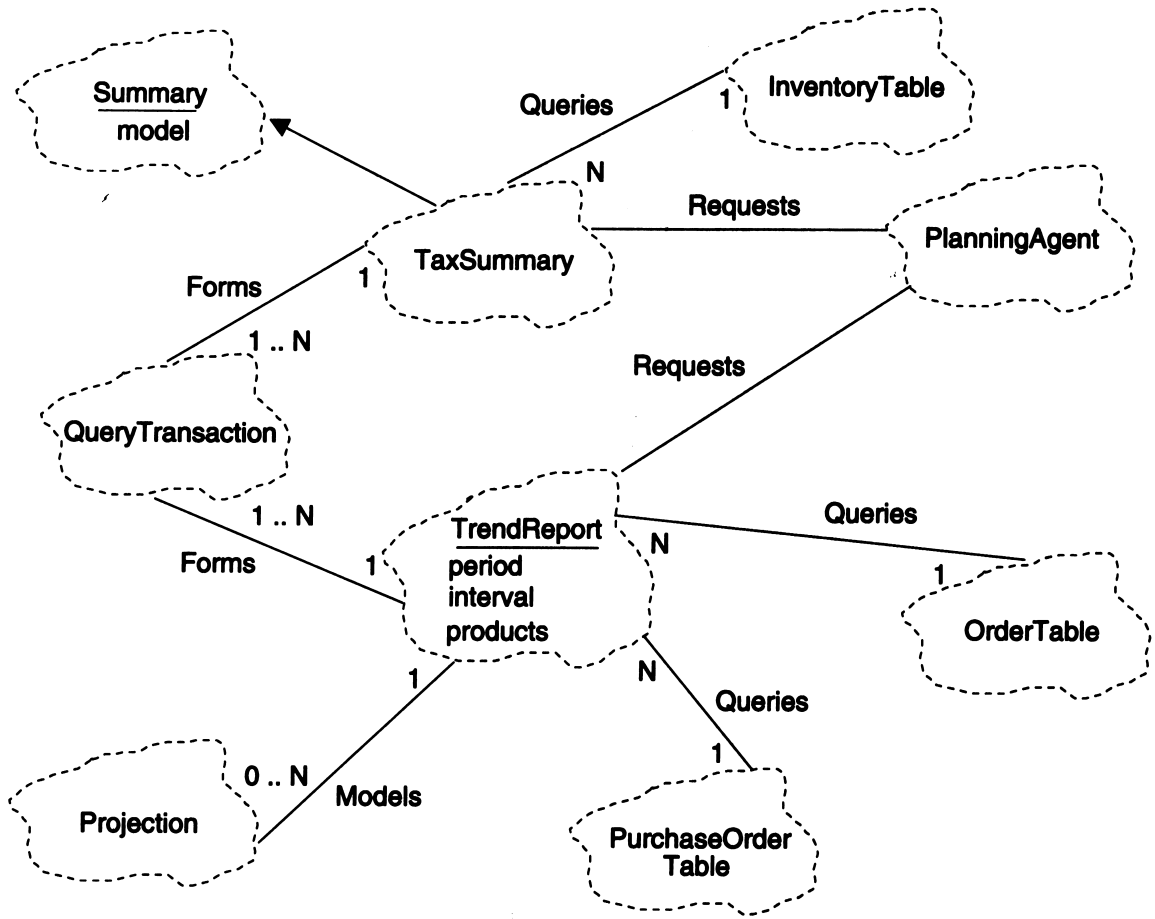


Figure 10-2
Key Classes for Generating Reports

3. It appears that based on initial requirements a decision was made to distribute the planning/reporting services to a server. Thus, whenever an individual user is attempting to ask questions concerning the sales database, this ultimately is turned into a query to some shared database. This architecture may have worked well for a business situation in which the reports and analysis is done in a sort of "batch" mode, e.g., at regular intervals and with regular patterns of queries. However, the business climate has evolved since then. The availability of rich data has spawned new tasks, tasks that are more ad hoc and open-ended in nature. Because these tasks have a dynamic character to them (e.g., the response to one query may evoke a chain of other queries), they have dynamic processing requirements. It appears that the server communication link is bogging down these tasks. A solution would be to move all of the analysis facilities into the client application. This might mean, for example, creating a shadow version of the relevant databases and downloading them prior to analysis, enabling a flexible and fluent analysis session. If the system has been designed in an object-oriented fashion, with abstractions serving as an interface between the actual data and the application programs, this re-engineering should not be difficult.

4. In this scenario, a client requests quantity information. In order to respond to this request, the query object obtains from the product its product identification to incorporate into an SQL selection request.

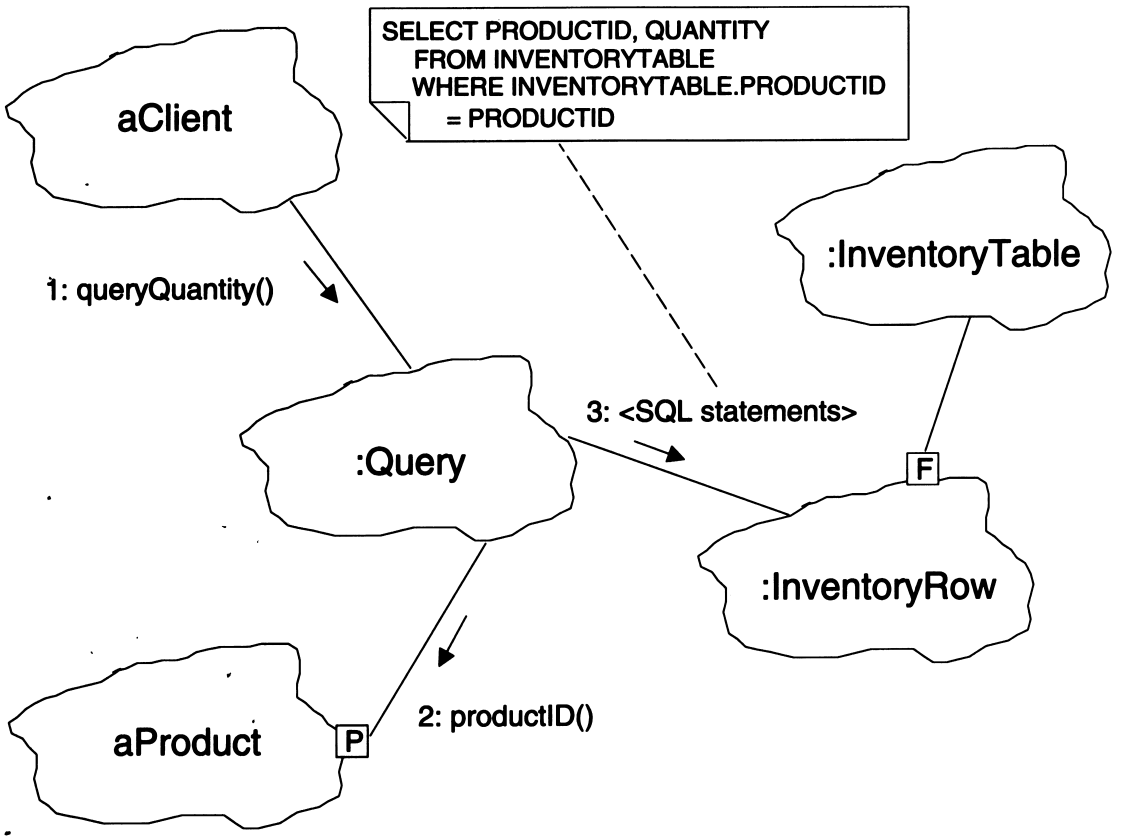


Figure 10-3
Item Quantity Query

5. The general impact of inluding database design as part of the process would have been to remove constraints from the process, making it much more open-ended and flexible. This of course would have made the problem much more difficult (but also more fun!). The impact of this flexibility would have been felt throughout the development process. For example,

   a. analysis — considerably more effort would have been spent in advance, working out a model of the products being tracked. Given an existing database, it is sufficient to understand what abstractions are appropriate for accessing what is already there. But given the opportunity to create a database, you must begin by analyzing the content it should contain. This would be a major classification effort.

      Scenario analysis would be much more open-ended, as what counts as a reasonable scenario clearly interacts with the nature of the information in the database. Further, an object-oriented classification system might encourage system users to conceptualize products differently than a relational system (e.g., more wholistically, or in terms of the hierarchical structure they reflect), and this might inspire very different kinds of usage scenarios. An initial exploratory period using a prototype database would almost certainly be required.

   b. design — the database would require considerable architectural work, for example, determining what communication protocol it would have with the rest of the system, how its components would be distributed if at all, the kinds of access controls it would support and how these would be enforced, and so on. A major issue would be the "query" language developed for accessing the data. In the chapter, the standard language for querying RDB's was adopted; here, the "language" might simply consist of the

abstract protocol developed for a query abstraction. The whole notion of a transaction as the atomic unit of communication might need rethinking — perhaps, for example, the database would admit some objects of the active sort, who periodically inform interested clients of changes in their state.

c. implementation — the evolution of the actual system would probably have an increased amount of iteration in it, as the representation of the information would be evolving in concert with the techniques developed for accessing and analyzing it.

d. maintenance — now, instead of maintaining an abstract interface to an existing database, tuning and enhancing the database body itself would become part of the maintainers' job. This means that if the business model for the inventoried products changes, it is up to the maintainers to determine whether such changes can fit within the object-oriented architecture originally designed, or whether serious re-engineering of the database will be required. The benefit is that if the business model does evolve, the presence of object-oriented abstractions all the way "down" into the data will almost certainly simplify the introduction of corresponding representational changes.

CHAPTER 11

Artificial Intelligence: Cryptanalysis

Discussion Questions

1. What constitutes the blackboard framework? What alternatives exist for the design of intelligent systems? Would they have worked well for the problem analyzed here? Why or why not?
2. What is the relationship between the objects placed on the blackboard and the available knowledge sources? How are the different knowledge sources in the cryptanalysis problem related?
3. Would there ever be more than one instance of a given knowledge source in the cryptanalysis system? What does this suggest about designing the system in a language where classes behave as objects (e.g., Smalltalk or CLOS)?
4. Why was dependency introduced as a mixin class rather than via introduction of an intermediate superclass? Why were affirmations introduced via aggregation rather than inheritance?
5. What is the role of assumptions and assertions in the cryptanalysis design? What are examples of these?
6. What is the relationship of the `Blackboard` class to the foundation class `DynamicCollection`? Is this likely to be a common usage situation for the foundation library classes? Can you think of other similar examples?

Exercises

1. In what way(s) does the cryptanalysis system exemplify the attributes of intelligent systems described by Erman, Lark and Hayes-Roth? For any attribute(s) it doesn't already exemplify, can you describe extensions that would qualify it?
2. Consider an expert system developed to critique the compositions produced by students in a Freshman Literature course. What might be an example of forward chaining? Of backward chaining? What sorts of objects might be involved in these reasoning activities?
3. Recall the hydroponics farm that has figured in many examples in the book. How might this have been conceptualized as an intelligent system? Could the blackboard framework have been applied? If so, how?
4. Figure 11-8 in the chapter diagrams the basic assumption mechanism designed for the cryptanalysis system. How would this diagram be altered to depict a retraction mechanism in which the retraction of a given assumption caused assumptions related to it to be simultaneously retracted?
5. A key responsibility in the cryptanalysis process has been given to the controller object, which must adjudicate among competing knowledge sources, given the current state of the blackboard. The case study presented here provides a few suggestions as to how this prioritization might be carried out. Work through the example scenario given in the book (the nine steps applied in solving `Q AZWS DSSC KAS DXZNN DASNN`). For each step, describe the proposals which were likely to have been sent to the controller, and then indicate which proposal was accepted. What priorities does the controller managing the example scenario seem to have in play? Can you see any relationships in how the knowledge sources are treated? What does your analysis imply about the information used by the controller to select among candidate assumptions?

Projects

1. A critical component of designing an intelligent system is the analysis of the knowledge that must be represented and applied. Choose an AI application for which you can locate a domain expert — examples might include a course planning advisor, a chess game, an algebra tutor, or an investment advisor. Interview the expert to extract the knowledge they apply in solving problems in their domain. A useful technique might be to first ask them to enumerate common problem situations, and then ask them to describe the knowledge they would bring to bear on each problem. Take the information they provide you and organize it into a set of classes representing knowledge abstractions. Check back with them to evaluate your abstractions. How well did you do? Can you think of ways you might have improved upon your knowledge extraction and/or modeling process?

2. The last section of the chapter proposes a simplistic learning mechanism that might be used to extend the cryptanalysis system, whereby users are asked for hints and the hints recorded and re-applied when relevant. Consider instead a more "intelligent" learning mechanism in which the system learns from its own successes and failures. For example, it might keep records about the priorities scheme(s) it has used, so as to tune them, or perhaps it stores solution efforts that turn out to be particularly effective. Develop some scenarios of "learning" in such an extension. Use these to analyze the new abstractions that would be required by the system and refine these into classes and their interfaces. Be sure to show how your learning subsystem would fit into the design of the existing system.

Sample Answers for Chapter 11 Exercises

The exercises represent design problems and have no single answer — any given situation can be analyzed in a variety of ways and to many different levels of detail. These sample answers reflect what we consider to be a reasonable analysis at a reasonable level of detail given the information available in the book.

1. Erman, Lark and Hayes-Roth point to five attributes that distinguish intelligent systems from conventional systems.

   a. goals vary over time — The cryptanalysis system reflects this in that it always holds a current set of assumptions about the possible mappings from the target to the solution, and tries to add new assumptions as they qualify. The evolution of the assumption base can be seen as an evolution in the specific subgoals addressed by the inference engine.

   b. use of knowledge — In some sense, any object-oriented system reflects this characteristics, as all object encapsulate knowledge about themselves and their responsibilities. But from the perspective of "knowledge-based systems" the cryptanalysis program has several important kinds of knowledge. The most obvious is the knowledge the source objects encapsulate concerning the system's beliefs about sentence, word, string and letter characteristics of the English language. The controller further holds knowledge about the relative priority of these characteristics in solving problems of this sort. Finally, the blackboard can be seen as holding temporary knowledge, as in the current knowledge about mappings from target to solution.

   c. diverse, ad hoc subsystems — Each of the knowledge sources can be seen as a subsystem, and they use quite different rules for evaluating their applicability. On the other hand, the actual mechanism for making the inferences (i.e., the underlying inference engine) is a uniform mechanism, meaning that all of the variation is embedded in the declarative knowledge of the different sources. A more "genuinely" intelligent system might have a more complex knowledge processing mechanism where different knowledge sources rely on different mechanisms to evaluate relevance of knowledge (e.g., parallel activation for pattern-matching vs. production rules for grammar and syntax).

   d. intelligent interaction — This particular system is not interactive so does not exhibit this characteristic. However, one might easily imagine extensions which would make it so. For example, the system might be given the ability to provide a solution account. If this solution were simply "spit out" at the end of the process, the interaction would not be one with much intelligence. But if the solution account was constructed in a way to allow open-ended queries and analysis by interested users (say students of cryptanalysis), the system would exemplify this attribute of intelligent systems.

   e. self-allocation of resource — The program has at its heart a controller object whose responsibility is to manage resource and attention. Although not fully elaborated, it clearly has been given the responsibility to choose among competing suggestions made by the knowledge sources. Even simple management of priorities in this fashion is a sign of intelligence.

2. Forward chaining refers to reasoning from specific assertions to a general assertion. An example from a document-critiquing system might be an inference that "The author tried to gradually insert clues concerning the killer's identity" contains a split infinitive, based on a set of prior judgments concerning the sentential role of each of the constituents (e.g., that "The author" is a noun phrase, that "tried to insert" is a verb phrase).

   Backward chaining refers to reasoning from a conclusion (or hypothesis) back to the assertions that would have been necessary to support it, and verifying their existence. So, for example, the critiquer might use its knowledge of sentence structure to predict that "John looked all over trying to find ..." would be followed by a noun phrase, and then verify that the following sequence of words fit into this category (e.g., could be parsed into one of the possible noun phrase structures).

   The objects participating in these reasoning processes will be the knowledge sources (e.g., a sentence structure source, a noun phrase source, a lexical item source), the data objects (e.g., the letters, words, sentences, punctuation making up the document), as well as some set of objects which manage the application of the former to the latter. This would certainly involve an inference engine as in the chapter, and might well involve objects participating in a blackboard framework as well.

3. As described throughout the book, the hydroponics farm used the abstraction of an "automatic gardener", an object that relied on a set of growing plans to control the growing conditions for the plants in the garden. The plans were administered with the aid of sensors and other devices.

   The growing plans were provided as a given in this conceptualization of the farm, and the gardener simply administered them. However, another model would have been to give the gardener more responsibility, such that it constructed plans dynamically based on intelligent analysis of the existing conditions. This more futuristic vision of the farm would involve a different architecture, one that represented explicitly the knowledge needed to create and adjust growing plans, as well as a mechanism for applying that knowledge.

   Reuse of the blackboard architecture would be one way to provide this intelligence. Thus, the gardener would take on the role of the controller. New objects would need to be designed to take on the roles of the blackboard and the knowledge sources. The knowledge source objects would replace the original gardening plan objects, as now there would be no static "plan of record" but rather a dynamically engineered sequence of caretaking decisions. The system might choose to save a sequence of such decisions for subsequent review, but this would not be part of the central control mechanism.

   The relevant knowledge sources would be analyzed in collaboration with human gardeners. So, for example, gardeners might be seen to have knowledge about managing different plant characteristics: root quality, fruit production, leaf growth, and so on. These knowledge sources would be elaborated to include relationships among a plant's growth situation and implied actions. Whenever a plant's growth situation suggested a particular action, that knowledge source would post a suggestion to the gardener. Accepted suggestions (they would be prioritized according to the gardener's scheduling heuristics) would be applied, and the growth state updated on the blackboard. The growth state information itself would be provided by the farm's sensor system. The result would be a plant care environment that dynamically adjusted itself to particular plant (or sets of plants, depending on the sensor and control arrangements) successes and problems.

   (Clearly, the weak link in this conceptualization of the farm is in the sensor technology. The examples in the book discussed very basic sensors — for temperature, humidity, and so on. The demand for state information here would be much increased, as the knowledge sources would need to know about things like leaf density and color, root density and strength, and so on. In other words, the sensors would need to provide the information that a human gardener uses in making his or decisions about plant care.)

4. The diagram would have the same participants. However, there would be no creation of an assumption, as the assumption in question already exists. Instead the mechanism begins with the `KnowledgeSource` instance sending a `retract()` message to the `Blackboard` instance. The `Blackboard` instance would then invoke a `dependents` operation on the offending `Assumption` instance (made visible as a parameter to `retract()`), to construct a collection of related assumptions for removal. It would send this list in a `removeAll()` operation to its `Alphabet` instance as well as to affected `BlackboardObject` instances.

5. The example scenario reflects nine steps resulting in blackboard updates:

   a. The direct substitution source proposes switching V for W. It wins over likely proposals from the small word source (switch A or I for Q) and over the double letters source (switch TT, SS, LL, etc. for NN).

   b. A small word proposal (A for Q) wins over likely proposals for double letters (TT, SS, etc for NN) and word structure (final E after V).

   c. A word structure proposal (internal vowel) wins over likely proposals for double letters, and another word structure (final E after V).

   d. A small word proposal (three letter word is THE) wins over a likely double letter proposal. Note that the small word source might collaborate with the pattern-match source to select among its three-letter words ending in E.

   e. A word structure proposal (internal vowel) combined with a vowel set proposal (non-consonant letters) combined with a pattern-match proposal (only I leads to a word) win over (again!) a double letters proposal.

f. A pattern-match proposal (4-letter words with double EE) combined with sentence structure proposal (likely verb position) wins over double letter proposal and legal string proposal (S, T, C, R, P, W before H in DHENN).

g. A sentence structure proposal (hives cannot see -- this is actually more of a word meaning proposal, but that knowledge source is not one of the thirteen) combined with a small word proposal (I for Q) win over a double letter proposal.

h. A double letter proposal (TT, SS, etc. for NN) combined with a pattern-match proposal (LL would make SMALL) win over sentence structure (THE likely starts a noun phrase in I HAVE SEEN THE).

i. A sentence structure proposal (word following THE probably an adjective) combined with pattern-matching (could be STALL or SMALL) wins over letter frequency (ST more frequent than SM).

The controller appears to have a strong preference for small word proposals, perhaps because these are a well-restricted set and at least for humans with limited processing capacity, retrieval from a small set will be easier. It seems to treat letter-level knowledge (e.g., double letters, legal strings) with relatively low priority. This might be because this information is not as well encoded in the knowledge source. It might also be because in this particular case, the double letter proposal that kept coming up would have had few side-effects.

The analysis shows a number of interactions among knowledge sources. For example, the vowel source would never have anything to contribute unless some other knowledge source had proposed a vowel position. The pattern-match source seems unlikely to make suggestions without some filter on the set of possible matches (e.g., from the 3-letter word set, or believed to be an adjective). These sorts of interactions are easily handled by the blackboard architecture, because as soon as one source puts up their proposal, the secondary proposal would naturally follow. It might mean, however, that the controller would favor sequences of proposals exhibiting such dependency, as it suggests that the system could be closing in on a solution. One might imagine a "previewing" of proposals, where the controller in some sense tried out proposals to see which evoked the most response, as a way of prioritizing their usefulness.

Command and Control: Traffic Management

Discussion Questions

1. What are some of the special concerns that arise during the initial analysis of large projects such as the traffic control system described in this chapter? Where does analysis typically begin?
2. What is the role of the message abstraction in the train traffic control system analyzed in this chapter? What are example instances of this abstraction? What is its reuse potential?
3. What are the difficulties associated with managing plans for multiple trains? How does the proposed design address these difficulties?
4. Given the display needs of the traffic control system (e.g., train traffic flow, engine efficiency models), what are the pros and cons of using off-the-shelf graphics facilities? What design steps would you take to minimize the downsides of using such facilities?
5. In what sense(s) has the architecture described here been built with system evolution in mind? What are examples of evolutionary changes that would be easy to accommodate? Hard to accommodate?

Exercises

1. Figure 12-3 in the book documents the processor and device design for the train traffic control system. Suppose that as the system evolves, the amount of local analysis carried out on the train's single computer increases considerably — for example, the train now monitors detailed indicators of its fuel efficiency and environmental impact and makes constant adjustments to keep these at desirable levels. These real-time analyses make significant demands on the single computer's processing resources, and its ability to simultaneously manage all of the network traffic might be diminished. In response to this resource problem, the engineers might decide to off-load the message manager activity to a second computer. What design issues and opportunities are raised by this modification to the physical architecture?
2. The book offers a high-level proposal for a message class hierarchy (in Figure 12-4). Focus on the `TrainPlanMessage` subhierarchy. Analyze message reception scenarios involving instances of its three subclasses, specifying some of the state and behavioral characteristics that might be expected of these abstractions.
3. Given the train schedule planning mechanism described in the book, what would happen if an engineer entered a change into his plan at the same time as a dispatcher made a change to the train's plan?
4. The book suggests reusing the sensor data acquisition architecture developed in the weather monitoring case study as a mechanism for data acquisition in this train control system. Would the mechanism work just as described for the weather monitoring case, or would it need to be adapted for this situation? If the latter, how would you adapt it?

Projects

1. This chapter describes design issues associated with a rather complex command and control system. Consider a somewhat simpler system — one that adjusts the timing of traffic lights as a function of traffic flow at different times during the day. Develop an analysis and design for this system of the sort presented in the chapter. Does your system have analogs to the four main subproblems addressed in the train control problem

(message passing, train plans, information display, data acquisition)? Are you able to reuse parts of the design work presented in the chapter? Why or why not?

Sample Answers for Chapter 12 Exercises

The exercises represent design problems and have no single answer — any given situation can be analyzed in a variety of ways and to many different levels of detail. These sample answers reflect what we consider to be a reasonable analysis at a reasonable level of detail given the information available in the book.

1. At the most basic level, this physical change simply introduces another interface into the system. All of the incoming and outgoing messages would go through a single computer (as before), but now there would also be an interface from the train's message manager to its dedicated analysis machine. This interface would still involve network traffic, of course (i.e., the network connecting the two computers on the train), but the message traffic from outside the train could now be managed more locally.

   The opportunity presented by this more complex architecture is that messages could be interpreted and handled as a function of the train's current situation. So for example, many of the messages might be information simply intended for display to the train personnel (e.g., that problem on a track has been cleared); such messages would not require resource from the analysis machine at all. Other messages might be relevant to the ongoing analysis and passed on immediately (e.g., that the tracks are now wet from rain, thus changing the efficiency algorithms). Still other messages might be relevant to the analysis but able to be queued up until the machine was "free" (e.g., that part of a route had been altered, implying a new acceleration and deceleration plan).

   A secondary design issue would involve the now shared access to the train's displays and other devices. The responsibility for managing these devices would probably be divided between the two processors, with the message manager in charge of the display (although perhaps the analysis machine would be given its own display devices as well), and the analysis machine responsible for the devices it uses to monitor and control the train's behavior.

2. Instances of the three concrete `TrainPlan` subclasses might participate in scenarios of the following sort:

   a. `PickupMessage` — A message might be constructed to alert trains of a stranded engine needing pickup. A train near in location to the stranded car might ask for further information about the problem (e.g., how heavy the engine is, what connection apparatus it requires). Assuming the information matched its capability, it might then return a modified version of the message indicating that it will take responsibility, following that with an update to its train plan.

      This suggests that the abstraction needs some sort of subject operation, as well as a location operation, so that interested clients can determine whether they should "listen" to the message. It also needs state and behavior for elaborating on the target of the pickup request. Finally, it might have behavior to converting itself from a pickup request into a pickup acknowledgement.

   b. `ClearanceMessage` — A message might be constructed to alert a train that it may proceed with its current plan. The message would be directed to a specific train.

      This message form is much simpler than the pickup message. It would need to have behavior for identifying which train was to receive clearance. Assuming that it makes sense to "clear" more than one situation (e.g., just starting vs. going over a bridge), then it would also need behavior to identify the form of clearance being provided.

   c. `TrackWorkMessage` — This message is somewhat like the pickup case described above. A message might be constructed to alert any train near a certain location at which work was being carried out. A train approaching the work site might want to find out more about the work referenced in the message (e.g., its expected finish time), so as to plan changes to its route accordingly.

      Like the pickup case, this suggests the need for behavior to identify the subject (work category) and location of the problem. In addition, behavior for describing the characteristics of the work is implied.

3. The apparent competition of the simultaneous updates would be resolved by the forced updating mechanism. The engineer's update would invoke a modification message back to the central database, which would in turn invoke modification messages to all other stored copies of this particular plan. No permanent change to the plan would be made until all copies reported a successful modification, at which time a change confirmation

might be sent out. In parallel, an analogous process would ensue from the change initiated by the dispatcher at the central location. Again, no permanent change would be made until all copies had confirmed this modification

The only consequence of the simultaneity of the updates would be in the order in which the changes appeared in the local and central copy of the plan. Even though the change made locally would not be permanent until propagated through the network, it would be applied to the local copy of the plan *before* the one initiated by the dispatcher (and vice versa for the change made by the dispatcher). Normally, this would not be problematic, as most modifications would not be order-dependent. However, the system users might encounter some unusual situations if the engineer and the dispatcher happened to simultaneously update the same piece of data! In that case all copies would confirm both changes, but there is no guarantee that they would have been made in the same order as the central copy.

4. The architecture developed for the weather monitoring station will be satisfactory for much of the data acquisition needs of the system described here. Its time-frame-based sampling mechanism is perfectly adequate for those sensors whose information is needed on a regular and predictable basis. However, some of the sensors in the current system may be tightly linked to safety concerns (e.g., infra-red detection of overheated wheel bearings, rapid drops in brake fluid level). Sensors such as these might be best modeled as active agents who can, for example, take the initiative to report a piece of particularly alarming data. Modeling all sensors as active agents would lead to an overly complex solution, but there may be a need to develop such a mechanism for a small set of safety-critical sensor devices. Before doing this, however, the designers would need to carefully consider the costs of having two different high-level policies for sensor input.