Chapter 2 - Finding objects

Overview

In Chapter 1, we introduced the principles of object-oriented problem solving but we have not paid much attention to how object-oriented solutions work .And we completely ignored the question of how to solve a problem using this approach, in other words, how to find the objects. Although there is no magic formula, a lot of experience gathered by experts has been crystallized into procedures that help in this difficult task and we will describe a simplified form of one such procedure and illustrate it on examples.

This chapter is an introduction to design so don't expect to learn it here and don't be frustrated if you don't. Design must be practiced and we will provide much more exposure as soon as we know some Smalltalk.

2.1 Examples of object-based solutions

Now that we the nature and the role of objects, it is time to look at whole object-based solutions and how they can be constructed. Before we introduce a method for constructing object-oriented solutions, we will now present two problems and show their solutions, without showing how they have been obtained. We will limit ourselves to the general features, showing the main classes, their relationship, and their cooperation during the operation of the application. The presented solutions are not complete because our goal is to demonstrate their general nature rather than all the underlying detail.

Example 1: Front-end of a chess playing program

Chess playing programs are among the most difficult problems and we will not attempt to attack this complex task in this section. Instead, we will focus on the 'front-end' of such a program, the part that displays the chessboard and knows how to recognize and display legal moves. This part of the chess program does not posses any intelligence beyond knowledge of chess rules and the current state of the chess board. The desired user interface is shown in Figure 2.1 and the complete implementation is presented in Chapter 13.

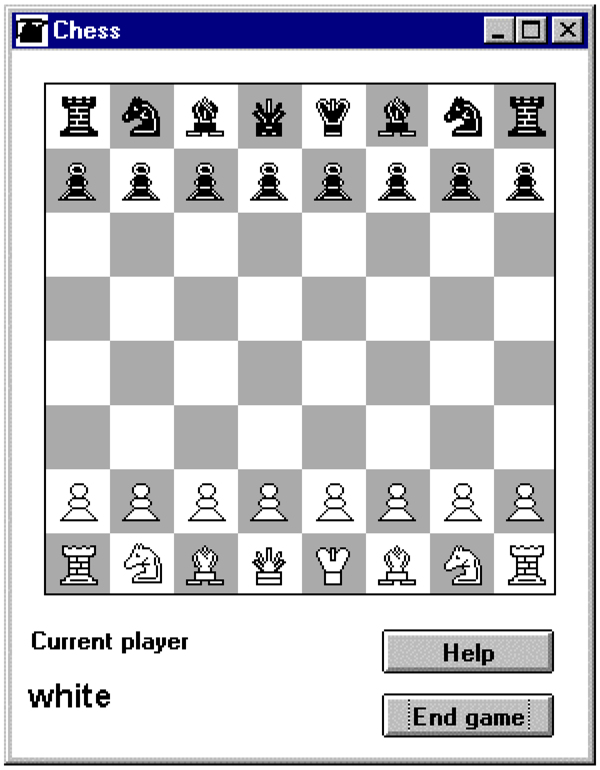


Figure 2.1. Chess user interface.

The program works as follows: When the player clicks a piece and its desired new position, the program first checks whether the move is legal. If it is, it makes the move and displays the new state on the screen. The program displays a message when the player attempts to make a move that is illegal or that results in a check to the opponent's king, a check mate, or a draw.

Solution: A solution consists of classes that collaborate to accomplish the specified goal. The essence of our solution is a class representing the chess board (ChessBoard) and one class for each kind of piece (King, Queen, and so on). All chess pieces share certain basic properties and these properties are collected in an abstract superclass called ChessPiece (Figure 2.1).

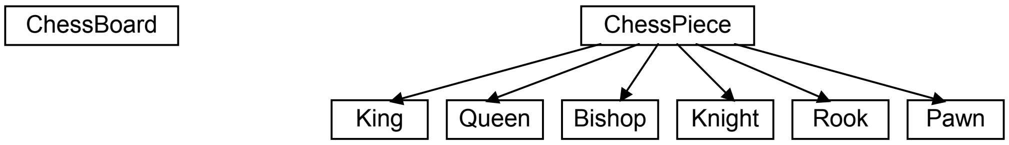


Figure 2.1. Class hierarchy of chess classes in Example 1.

A brief description of some of these classes is as follows:

<u>Class ChessBoard</u> keeps information about each square of the board and knows whether it is occupied and by which piece. It also knows how to draw the board at the start of the game, and how to redraw it after a move. In doing so, it assumes that each piece knows how to draw itself.
<u>Class ChessPiece</u> knows the color of its piece (black or white). It can also determine whether a move is horizontal, vertical, or diagonal and whether it can be made without colliding with other pieces. These methods are used by its concrete subclasses to test the legality of their particular moves. Since ChessPiece is an abstract class, it does not have any instances.

Class King knows how to test whether it can move to a given location (to do this, it uses methods inherited from ChessPiece) and whether the king is in a check position. It knows how to draw itself. Class King is concrete and two instances exist at any time, one for each player.
Class Queen knows how to test whether it can move to a given location and how to draw itself. Class Queen is concrete and the game starts with two instances, one for each player.
Classes implementing other pieces have similar functionality tailored to the rules of their movement.

After this brief description, let's examine a few scenarios to see how our classes work together to implement a chess front end. We will start with the simplest scenarios.

Scenario 1: Clicking an empty square as the starting point of a move
Conversation:
1. *Player* clicks an empty square as the starting point of a move.
2. aChessBoard (an instance of class ChessBoard) checks the square, finds it empty, and ignores the mouse click.

Scenario 2: Attempt to move owned piece to an illegal destination
Conversation:
1. *Player* clicks a square occupied by her piece as the starting point of a move.
2. aChessBoard checks the starting position and finds it occupied by the player's piece. The attempt is legal.
3. *Player* clicks another square as the destination of the move.
4. aChessBoard asks piece to attempt the move.
5. piece tests whether it can move to the specified position and finds that it cannot.
6. piece returns a failure result to aChessBoard.
7. aChessBoard displays a failure message to *player* indicating the cause of the failure.

Scenario 3: Legal attempt to move a piece
Conversation:

1. *Player* clicks a starting position position.
1. aChessBoard checks the square and finds it occupied by the player's piece.
2. *Player* clicks another square as the destination of the move.
3. aChessBoard asks piece to attempt the move.
4. piece tests whether it can move to the specified position and finds that it can.
5. piece returns success result to aChessBoard.
6. aChessBoard asks piece to draw itself in the new position.
7. piece draws itself in the indicated position.
8. aChessBoard redraws the original position of piece as an empty square.
9. aChessBoard asks the opponent's king whether the new position is a check.
10. king tests whether it is in check and returns the result to aChessBoard.
11. If the new position is a check, aChessBoard displays a message to that effect.

By verifying that all steps of all our conversations use only behaviors listed in our class descriptions, we can eventually confirm that our classes indeed solve the problem.

Example 2: Travel advisory program

Problem: This program will be used by travel agents to inform customers about available trips. When started, the program displays a window showing all available destinations. When the client selects a destination, another window opens providing access to details about the selection. It contains a 'notebook' widget with tags providing access to information about flights to this location, the location itself, and local car rental companies and hotels. By clicking a tag, the user opens a page showing more information about the selected topic

When the user clicks the location tag, the page displays textual information and a picture of the location. The car rental page is selected in the same way and shows a list of car rental companies available at that location; when the user selects one, the page displays a list of all available rates. For flights, the page shows a list of flights to this location with the name of the airline company, flight departure and arrival times, and price. The hotel information page shows a list of hotel names and their ratings (one to five stars). When the user selects a hotel, the page displays available rates (a list of number-of-persons-in-room/price items) and a photograph of the hotel, and shows the hotel's location on the map.

Solution: A possible solution uses the following classes, mostly holding information and providing access to it. There are no inheritance relationships.

<u>Class Application</u> is responsible for the main window with its buttons and other parts of the user interface. Its instance variable locations holds Location objects. This class has a long list of responsibilities, unlike the following classes which are essentially just information holders.
<u>Class Location</u> has detailed information about a destination. It has instance variables name (a text object), description (text), picture (a picture object), carRentals (list of CarRental objects), flights (list of Flight objects), and hotels (list of Hotel objects).
<u>Class CarRental</u> describes a car rental company. It holds a name object with the name of the company and a list of RentalRate objects in an instance variable called rentalRates.
<u>Class RentalRate</u> describes a specific car rental choice. It holds a carModel (text) and a pricePerDay numeric value.
<u>Class Flight</u> describes a flight. It holds the companyName (text), departureTime, arrivalTime, and price.
<u>Class Hotel</u> describes a hotel. Its instance variables are rating (text), name (text), listOfRates (instances of HotelRate), picture, and coordinates of the hotel on the map (a point object).
<u>Class HotelRate</u> describes a particular hotel rate. Its instance variables are occupancy - a number representing the number of occupants, and rate - the price of the room at this occupancy rate.

The following selected scenarios shows how this design can handle our problem.

<u>Scenario 1: User starts the application</u>
Conversation:
1. *User* clicks executes a Smalltalk expression or clicks an icon to open the program.
2. Application creates its instance (called anApplication in the following steps) and reads travel information from a file into variable locations.
3. anApplication obtains all destination names from the locations variable.
4. anApplication opens the main window showing all destination names.

<u>Scenario 2: User selects a location and displays information about flights</u>
Conversation:
1. *User* clicks a destination tag.
2. anApplication uses locations to access the corresponding instance of Location (call it aLocation).
3. anApplication opens a containing information about the selected location and tags for car rentals, flights, hotel reservations, and location information.
4. *User* clicks the *Flights* tab.
5. anApplication uses flights to obtain information about individual flights. For each Flight, it obtains and displays the information on the *Flights* page.

We leave it to you to construct additional scenarios and check that our selection of classes can indeed solve the problem.

<u>Exercises</u>

1. Draw a very simple chess board situation with three or four pieces. Identify the objects involved in this situation and assign each to one student. Each student will make an index card and record all information required by his or her assigned object on it. As an example, the *chess board* person will have a record of the chess board and the occupants of individual squares, and the *king* person will record the king's color. Play scenarios similar to those listed above, making sure to update information on cards as you play.

2. Repeat Exercise 1 but assign each classes instead of pieces. Each card contains the name of the class, its brief description, and a list of its functionalities. Replay the scenarios.
3. Construct detailed conversations involved in several other scenarios in Example 2.
4. Play scenarios from Example 2 in the way described in Exercise 2.

2.2 Finding objects

We have now seen the essentials of object-oriented solutions of two problems and it is time to show how such solutions can be found. We will outline the general principles in this section and illustrate them on two examples in the rest of the chapter.

Methodologies for identifying objects

Although there is no set of rules that would let you find objects for an arbitrary problem mechanically, experienced practitioners formulated sets of steps and guidelines that help in finding a solution. These recipes are called methodologies.

Since the emergence of object-oriented programming, several methodologies summarizing such steps and guidelines have been formulated. In this book, we will use a methodology loosely based on Responsibility-Driven Design (RDD) originally described by Rebecca Wirfs-Brock and her co-workers. In this section, we will explain the principles of RDD and the following sections will illustrate the method on two examples. For further details, read the specialized literature listed in the References at the end of the book.

Before outlining our approach, we will give a more accurate definition of what a design methodology is. Formally speaking, a design methodology consists of a process, a set of guidelines, and a set of deliverables. A *process* is a sequence of steps that must be executed to solve a task (in this case finding classes for a given problem). The *guidelines* provide hints that should help in the execution of the process and in finding a good solution. *Deliverables* are the artifacts (the documents and the software) that must be produced during the process. Each methodology prescribes the contents of the required documents, usually as a combination of text and diagrams using a set of carefully defined graphical symbols.

A point worth noting is the great importance attached by all methodologies to documentation. Most of these documents take a lot of time to produce and are difficult to keep up to date, and some developers think that producing them is a waste of time and that the only deliverable should be the software. There are even some who claim that orderly design is a waste of time and that one should simply sit down and develop the application using only the programming environment of the selected programming language. (In all justice, the pressure of deadlines sometimes make it difficult to produce both documentation and the product and there is no difficulty in deciding which of them is more important to produce.)

The reason why *orderly design* is desirable is that any non-trivial application quickly becomes unmanageable if you don't develop it systematically. If the number of classes exceeds ten or so, keeping track of them becomes impossible for most ordinary programmers. If you 'hack' the program and create it in a disorderly fashion, you will soon lose track of the purpose of the classes that you created, their mutual relationship, and the purpose and use of their methods and variables. Furthermore, as the project evolves, you will invariably decide to make changes and since you will be reluctant to redesign the application, you will continue making patches to the wrong design. The result will be that you will spend more time on development than if the design was done systematically from the beginning because making design changes before any code has been written is much easier than making changes when your ideas become a program.

The reasons why you need *good documentation* are much the same. Without documentation, the design soon becomes very difficult to understand and further development becomes impossible. Even worse, when the product is completed and delivered, its maintenance will be very difficult if the original design is not documented. Maintenance without documentation is difficult even when the original developers are still available, and almost impossible when they leave the development team. In summary, although proper process and documentation cost extra time and money in the short term, they save time and money in the long run. They also give a better assurance of quality. Like everything else, preimplementation analysis and documentation can be overdone and too much time and money spent on it. Critics refer to this mistake as 'analysis paralysis'.

Responsibility Driven Design

All Object-Oriented (OO) methodologies divide application development into the formulation of a detailed initial specification of requirements, refinement of problem understanding and preliminary identification of classes needed to obtain a solution, refinement of class description, and implementation. These stages are usually referred to as requirements specification, analysis, design, and implementation (Figure 2.3). The attention of development methodologies centers around object-oriented analysis and design, frequently referred to as OOA/D.

The general structure of the *development life cycle* of a software product has been inherited from software engineering research preceding the adoption of the object-oriented paradigm. Although this division is still used, the boundaries between analysis and design are fuzzier in OOA/D than they were in classical methodologies. The main reason for this is that analysis and design in traditional languages relied on different paradigms and transition from analysis to design required a transformation from one perspective to another. OOA/D, on the other hand, uses the same paradigm (communicating objects) throughout the process.

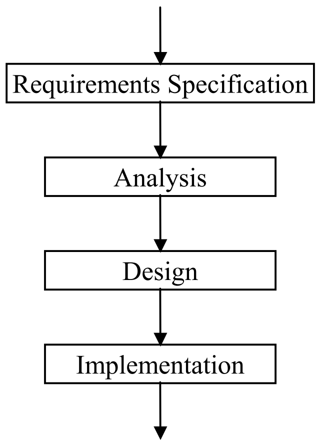


Figure 2.3. Basic software development life cycle.

The development process that we will use and consist of the following major steps:

1. *System Description*. In this step, we obtain a basic problem statement from the client and refine it with the help of subject area expert into a detailed description of the desired product. The specification should avoid, as much as possible, any consideration of the programming language and even the programming paradigm that will be used for implementation.
   The deliverables consist of a textual description, a set of usage scenarios and their expansion into high-level conversations between the user and the system, and sketches or computer models of the desired user interface. For more complicated systems or when a larger design is divided among teams, we require a Context Diagram showing the major components of the system that are to be designed and indicating which parts are outside the system, already exist, or will be obtained somewhere else.
   If the problem is not trivial, it is usually necessary to create a glossary explaining the terms used in the description.
   During this stage, designers and clients closely cooperate to ensure that the specification is complete and all its aspects correctly understood.
2. *Exploratory Design*. Now that we know what the client wants, we study the System Description to find the key classes needed for the solution. We identify candidate classes and write their preliminary descriptions on CRC (Class-Responsibility-Collaborator) cards using the format shown in the

following section. We then expand the high-level conversations from Step 1 into class-level conversations involving the identified classes.
As we develop the conversations, we record the responsibilities that the classes must have to implement their part in the conversations on CRC cards. We also record the attributes that they need to fulfill their responsibilities, and attach names of collaborator classes to individual responsibilities. Finally, we find how and why the classes are related to one another and record this information in an Object Model diagram.
Beginning with this stage, we assume that the solution will be object-oriented. In fact, we even assume a specific programming environment because our choice of new classes depends on classes that are already available. When we are finished, we test the preliminary design informally by executing class-level scenarios with CRC cards and making sure that all conversations can be completed with the recorded information.
The deliverables include class-level scenarios, an Object Model diagram, and preliminary class specifications with class name, brief description, attributes, responsibilities, and collaborators.
3. *Design Refinement*. In this step, we add details to results of Exploratory Design. We check whether any of the previously identified classes are related, and define abstract superclasses factoring our the shared properties if this is the case. We use the results to draw a hierarchy diagram and update the Object Model. We add enough detail to class descriptions to make implementation easy, going as far as deciding the names of methods and describing their roles and principles of implementation. We consider whether any of the identified classes might be useful in future projects and possibly modify the design to make such reuse possible. Finally, we again test whether the identified details are sufficient to satisfy all scenarios.
The deliverables include refined deliverables from Exploratory Design and a class hierarchy diagram.
4. *Implementation*. The detailed design is now translated into a programming language. This step will usually require some additions but if the design was done carefully, these additions will be minimal.

The following table and the diagram in Figure 2.4 summarize the process.

Phase	Activities	Deliverables
System Description	Discuss desired product with client, identify which parts of the system are to be developed and what is outside the system (Context Diagram), find major scenarios and expand them into high-level user-system conversations. Decide preliminary specification of user interfaces. Identify priorities. Write glossary.	Textual specification of requirements, usage scenarios with high-level conversations, model of user interface, Context Diagram. Priority list. Glossary.
Exploratory Design	Analyze specification and scenarios to find candidate classes. Expand high-level conversations into class-level conversations. Use CRC cards for first exploration, convert to electronic form when stable. Test that recorded responsibilities can implement class-level conversations. Identify relationships between classes (Object Model).	CRC cards, preliminary class specifications, class-level conversations, preliminary Object Model diagram.
Design Refinement	Decide whether abstract classes are desirable to factor out shared behaviors. Determine superclass-subclass relationships, decide method names and write descriptions of more complicated methods. Refine Object Model by adding superclass information.	Finalized class specifications, class hierarchy diagram, refined Object Model diagram.
Implementation	Implement design in the selected programming environment. Test functionality using high-level scenarios and other techniques. Evaluate performance and optimize critical parts if necessary. Produce manuals and other user documentation.	Tested application and user documentation.

Although our description is complete, it gives the inaccurate impression that development is a linear sequence in which step n is always completed before step n+1, and never repeated. The reality is generally quite different because when we start working on step n+1, we often find that the results obtained in step n are incomplete or partially incorrect. Results of step n must then be corrected and this may require corrections or additions to step n-1as well, and so on. While this is quite normal, one should always attempt to complete each step as well as reasonably possible. Do not proceed to the next step lightheartedly, assuming that you will have to return to the previous step anyway.

Another reason why development does not usually proceed in a linear sequence is that it is often better to develop an application *iteratively* (Figure 2.4). In other words, the best approach often is to repeat the whole development sequence several times, progressively attaining higher and higher levels of completeness, perfection, and client satisfaction. The main reason for this is that if the project is not trivial, developers rarely achieve full understanding of their clients' desires after the first meeting. In fact, the clients themselves often cannot initially anticipate all details of their needs and spending much time perfectly implementing an imperfectly understood problem would be a waste.

For larger problems, the best approach is thus to start by focusing on the most important and best understood parts of the specification, and design and implement a prototype without spending more time on the first iteration than necessary. The prototype is demonstrated to the client to obtain corrections and further detail, another iteration is performed, the prototype updated, and so on, until a completely satisfactory implementation is achieved. Experts recommend three or four iterations but the number depends on the complexity of the problem, the experience of the development team, client's understanding of the problem, and other factors.

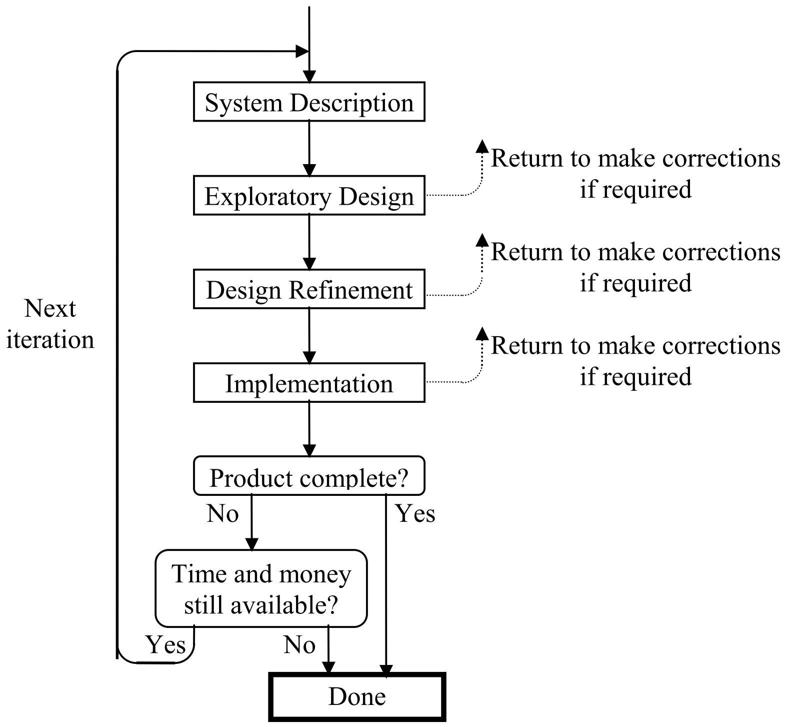

System Description
Exploratory Design
Return to make corrections if required
Design Refinement
Return to make corrections if required
Implementation
Return to make corrections if required
Next iteration
Product complete?
No
Yes
Time and money still available?
Yes
No
Done

Figure 2.4. Development life cycle. Testing that occurs at each stage is not shown.

Iterative development is the most effective development strategy in terms of time, cost, and client satisfaction and since Smalltalk is excellent for prototyping, it is an ideal choice for this development style. In fact, it has been reported that projects whose target implementation language was other than Smalltalk have been initially prototyped in Smalltalk with the goal of converting the prototype to the target language when a satisfactory level of completion has been achieved. In practice, many such projects were never converted to another language because the developers found that Smalltalk was the best environment to implement the final product as well.

After this general introduction, we will now demonstrate the development process on two examples. Because of lack of space, we cannot fully illustrate the iterative nature of program development and our examples will thus give an inaccurate impression of a relatively smooth linear development process.

Main lessons learned:

- Software development and maintenance consists of a sequence of steps referred as the development life cycle.
- Finding objects and describing the properties that they must have to solve a given problem is the essential part of software development.
- A software development methodology consists of a process, a set of guidelines, and deliverables.
- A process is a sequence of steps to be executed to solve a task.
- Guidelines are experience-based hints that should help to obtain a good result.
- A deliverable is a document or a product that must be completed at a given stage of the development process.
- All software development methodologies use a specification-analysis-design-implementation process.
- In object-oriented development, the boundary between analysis and design is fuzzy.
- Development of a non-trivial application cannot be accomplished in a single pass and it is recommended to use several iterations to achieve better and better understanding of client requirements and optimal design.
- Our methodology is based on responsibility-driven design (RDD). We call its phases system description, preliminary design, design refinement, and implementation.
- Besides textual specification of the problem and a possible model of the user interface, a set of scenarios expanded into conversations is a fundamental part of requirements specification.
- The first decomposition of a scenario is a high-level conversation capturing the dialog between the user and the application. High-level conversations are constructed as a part of system description.
- For complex systems, we construct a context diagram to show major parts of the system and separate those parts that are to be designed from the rest.
- During preliminary design, initial class candidates are derived from the specification. High-level conversations are refined into class-level conversations that show cooperation between classes. A preliminary Object Model showing class relationships is constructed.
- Design refinement consists of finalizing class hierarchy and refining class descriptions and the Object Model.
- All stages of the development process are accompanied by testing.

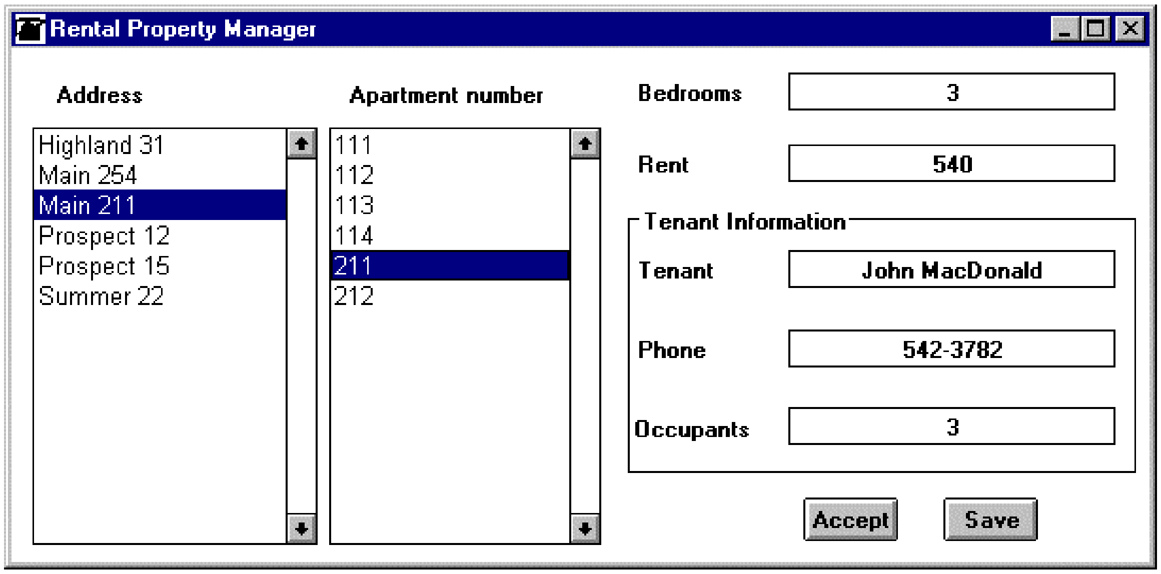
2.3 Example 1 – A Rental Property Management Program

In this section, we will present our first design example, closely following the development process introduced in the Section 2.2. We suggest that you use our CADE program to create, record, maintain, and print all the required documents. A sample implementation of a more sophisticated version of this problem is available on our web site.

2.3.1 System Description

The program allows a manager to maintain and access information about rental properties and their tenants. Each rental property is a building described by its address and apartments. Each apartment is described by its number, number of bedrooms, monthly rate, the name of the tenant, his or her telephone number, and the number of occupants. All information can be updated and saved in a file.

To run the program, the user types and executes a Smalltalk expression and this reads a properties information file and opens the window in Figure 2.5 with a list of all properties arranged alphabetically by their addresses. When the user clicks an address, the apartment list displays the numbers of all apartments in the building. When an apartment is clicked, the window displays all apartment information including the number of bedrooms, monthly rate, the name of the tenant, his or her telephone number, and the number of apartment occupants.



Highland 31
Main 254
Main 211
Prospect 12
Prospect 15
Summer 22

111
112
113
114
211
212

Bedrooms 3
Rent 540
Tenant John MacDonald
Phone 542-3782
Occupants 3

Figure 2.5. Desired user interface.

Given this initial specification, we will now explore the main usage scenarios, presumably with the help of the client. In the first iteration, we will focus on the 'normal' sequences of events and leave the abnormal alternatives (such as what happens when the information file does not exist) for the next iteration.

Scenario 1: *User opens the program.*
High-level conversation:
1. *User* types and executes a Smalltalk expression.
2. *System* reads properties file and displays the window with a list of properties.

Scenario 2: *User closes the program.*
High-level conversation:
1. *User* clicks the window button in the upper right.
2. *System* closes the window and asks whether to save new information.
3. *User* confirms that information should be saved.
4. *System* saves all new information, and terminates the application.

Scenario 3: *User clicks an address in the list.*
High-level conversation:
1. *User* clicks an address in the list.
2. *System* displays list of numbers of apartments in the building.

Scenario 4: *User clicks an apartment number.*
High-level conversation:
1. *User* clicks an apartment number in the list.
2. *System* displays all information available about the apartment.

Scenario 5: *User saves information in file.*
High-level conversation:
1. *User* clicks *Save*.
2. *System* saves current data about rental properties in a file.

Scenario 6: *User changes name of tenant.*
High-level conversation:
1. *User* types a new tenant name.
2. *User* clicks *Accept.*
3. *System* updates internal information.

Scenario 7: Users adds a new property.
This has not been described in the specification and the implementation is not obvious. We will invent a suitable implementation and confirm it with the user.
High-level conversation:
1. *User* opens address list <operate> pop up menu showing commands *add* and *remove* and selects *add*.
2. *System* requests property address.
3. *User* enters property address.
4. *System* displays address and requests number of apartments.
5. *User* enters a number.
4. *System* request a number for each apartment, *System* types individual numbers, *User* displays them immediately.

Scenario 8: Users removes an existing property.
This has not been described in the specification either. We will describe a suitable implementation and confirm it with the user.
High-level conversation:
1. *User selects a property by clicking it.*
6. *User* opens address list <operate> pop up menu showing commands *add* and *remove* and selects *remove..*
2. *System* requests confirmation.
3. *User* confirms.
4. *System* deletes the item from the list and blanks all text except for the list of property addresses.

This list of scenarios is obviously incomplete and we leave the rest as an exercise.

Identify major components of the application

Finding the major components of the application will help us understand the system better and clarify which part of the program is our responsibility and what is obtained from other sources.

Reading of the specification suggests several subsystems. One comprises all information about the properties, their apartments, and tenants. This subsystem does not need to know how to display the information or how to accept user input - it is only a computer model of the concepts involved in rental property management. Technically speaking, this subsystem is the *domain model* of our system. Separating the domain model from the user interface is a very good idea because the domain model should be usable with any user interface and any style of access, not only the reading and authoring modes described in our specification. We will call this part of the application the *properties subsystem*.

The next subsystem consists of all classes responsible for displaying a window and its components, and for accepting user input via the mouse and the keyboard. We will call this the G*UI subsystem* because it is responsible for the graphical user interface (GUI). This subsystem does not have any property information but it knows how to access, display, and modify it. It complements the domain model. Since the classes in this subsystem (classes responsible for windows, buttons, other widgets, and user interaction) are independent of the domain that they display, they can be reused in other applications, for example a spreadsheets or a word processors.

We now have a subsystem that contains domain information but does not have a user interface, and a subsystem that can handle all aspects of the user interface but has no domain information. We thus need a subsystem that builds and opens the user interface, provides communication between the user interface and the domain models, and closes the application. As an example of necessary communication between the user interface and the domain model, when the user enters a new apartment, this information must be communicated from the GUI to the domain model. Almost all applications require such a subsystem and the concept is thus well researched and in the VisualWorks community it is known as the *application model*. We will reuse and extend the built-in facilities and call this subsystem the *property application model*.

A diagram relating the subsystems and the external actors is shown in the Context Diagram in Figure 2.7.

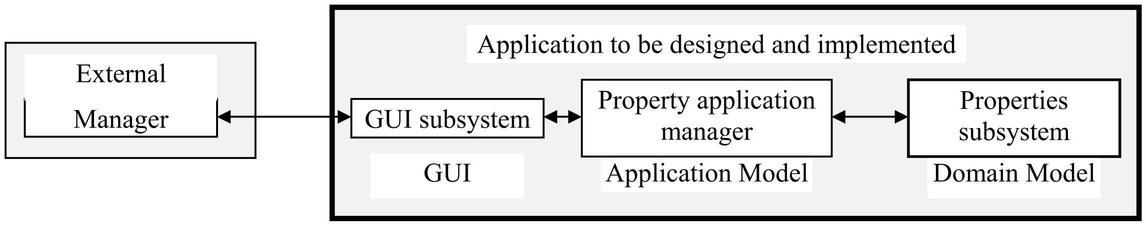


Figure 2.7 Context Diagram showing the main parts of the system.

Glossary

Apartment – information describing an apartment and consisting of its number, monthly rent, number of bedrooms, and tenant.
Manager – person authorized to use the program.
Property– information describing a rental property and consisting of its address and apartment information.
Property application manager – subsystem of program responsible for user interface information and providing a link between the GUI and the domain model.
Properties subsystem – the domain model of the program.
Tenant– information describing a tenant and consisting of name, telephone number, and number of occupants in the apartment..

2.3.2 Preliminary design - find candidate classes and define their responsibilities and collaborators

Identify classes

Identifying classes is the crux of object-oriented development. Unfortunately, as we have already noted, there is no cookbook recipe for finding objects. The following are two of the guidelines formulated by OO experts:

- If you can name it and talk about it, and if it's important to your system, it should probably be an object.
- Don't overload one object with several meanings: One idea – one object.

These two ideas provide conceptual insight by little in the way of identifying objects from the specification. As a more concrete guidance, some experts recommend starting the search for classes by textual analysis of the specification and the scenarios, in other words, by scanning the text for nouns and noun phrases that suggest suitable objects. Nouns that are obviously irrelevant or represent objects that are already a part of the programming environment are ignored. The nouns underlined in the following copy of our specification stand out as potential classes:

The program allows a manager to maintain and access information about rental properties and their tenants. Each rental property is a building described by its address and apartments. Each apartment is described by its number, number of bedrooms, monthly rate, the name of the tenant, his or her telephone number, and the number of occupants. All information can be updated and saved in a file.

To run the program, the user types and executes a Smalltalk expression and this reads a properties information file and opens the window in Figure 2.5 with a list of all properties arranged alphabetically by their addresses. When the user clicks an address, the apartment list displays the numbers of all apartments in the building. When an apartment is clicked, the window displays all apartment information including the number of bedrooms, monthly rate, the name of the tenant, his or her telephone number, and the number of apartment occupants.

We will now evaluate the desirability of the selected nouns as classes:

- *Address*. If the address included the name of the city, postal code, province or state, and other information, we would implement it as a class. As it is, an address is just a string of alphabetic and numeric characters and strings are already in the library. We thus remove *Address* from our list of candidates.
- *Apartment*. Yes, we will implement this concept as a new class because it represents an object that holds a lot of apartment information together and no related class exists in the library. The class will be called Apartment and we will decide shortly what exact meaning it will have.
- *Building*. Another and non-trivial object that does not have a counterpart in the library and will have to be implemented. The class will be called Building.
- *File*. The concept of a file is implemented in VisualWorks library and does not require a new class.
- *Name of tenant*. This is just another string and we don't need a class to implement it.
- *List*. This is a special kind of collection and Smalltalk contains several classes that can be used to implement it.
- *Window*. A window is a part of the GUI subsystem and since GUI classes are already in the library, it is beyond our concerns.

We conclude that we need to design and implement the following classes:

- Apartment
- Building

Note that we have eliminated most of our original candidates. That's quite OK because it is better to have more candidates than required rather than miss some important ones. When in doubt, include a candidate and leave the decision whether to keep it or not until further detailed analysis.

Have we missed anything? Yes, we have. We forgot about the tenant and about the application model. The tenant is a rather complex collection of information, and the application model is a non-trivial extension of class ApplicationModel in the library. We will call its class PropertyManager. The new list is

- Apartment
- Building
- PropertyManager
- Tenant

As you can see, identification of new classes requires a certain knowledge of the anticipated programming environment. Without this knowledge, we could not have concluded that we do not have to design classes to implement lists, buttons, notebooks, files, strings, and other objects. We assumed that we will implement the application in Smalltalk, in fact in VisualWorks Smalltalk. In another version (dialect) of Smalltalk, some of the GUI widgets, for example, might be missing although collection and string classes would be included in any Smalltalk that implements the Smalltalk standard. We also assumed that we have never implemented a similar application and don't have any suitable reusable classes. To this extent, preliminary design and the following stages of the development cycle depend on the selected implementation environment.

Identify class responsibilities

Class responsibilities are determined by the desired functionality, and functionality is described mainly by usage scenarios. We will thus expand high-level scenarios into conversations between classes and record the implied responsibilities as they emerge. When working in a team, we suggest the following procedure:

1. Assign each class to an individual or a small group that will be responsible for maintaining all information about the class.
2. Each group prepares a CRC card (Figure 2.8) for its class. Use 5 x 8" paper index cards.
3. Each group proposes a short description of the purpose of its class, gets it approved by the rest of the team, and writes it on the back of the card.
4. As you work your way through the scenarios, each group records information relevant to its assigned class. If a conversation shows the need for a new class, create a new card and assign it to a new group. (In practice, each individual developer is usually responsible for several classes.)

Class: Building	
Components:	
Responsibilities	Collaborators

I hold all information about a building and provide access to it.

Figure 2.8. The face and the flip side of a CRC card.

We will now go through the first few scenarios, expand high-level conversations into class-level conversations, and update our CRC cards. We leave it to you to check whether our analysis and records are complete, expand the remaining conversations, and fill any gaps found as you test your CRC cards at the end of this stage.

Scenario 1: *User opens the program.*
High-level conversation:
1. *User* types and executes a Smalltalk expression.
2. *System* reads properties file and displays the window with a list of properties.

Class-level conversation:
1. *User* types and executes the Smalltalk expression PropertyManager open.
2. PropertyManager reads the properties file.
3. PropertyManager obtains addresses from individual properties.
4. PropertyManager builds the user interface, and displays the window with the initial information.

As a result of this scenario, we note that PropertyManager is responsible for initializing the data and opening the interface. The display of building addresses requires cooperation of Building. We record this information on the PropertyManager and Building CRC cards.

Note that we restrict our attention to new classes and ignore existing classes such as those handling files. This is because our goal is to learn about the classes that we must design.

Scenario 2: *User closes the program.*
High-level conversation:
1. *User* clicks the window button in the upper right.
2. *System* closes the window and asks whether to save new information.
3. *User* confirms that information should be saved.
4. *System* saves all new information, and terminates the application.

Class-level conversation:
1. *User* clicks the window button in the upper right.
2. PropertyManager clicks the window button in the upper right.
5. PropertyManager closes the window and asks whether to save new information.

6. *User* confirms that information should be saved.
3. PropertyManager saves all new information, and terminates the application.

As a result of this scenario, we note that PropertyManager is responsible for closing the interface and saving the data. We record this information on the PropertyManager CRC card.

Scenario 3: *User clicks an address in the list.*
High-level conversation:
1. *User* clicks an address in the list.
2. *System* displays list of numbers of apartments in the building.

Class-level conversation:
1. *User* clicks an address in the list.
2. PropertyManager detects the selection and uses it to identify the corresponding Building.
3. PropertyManager requests list of apartments numbers from the selected building.
4. Building returns list of numbers.
5. PropertyManager displays the numbers in the apartment number list.

As a result of this scenario, we note that PropertyManager is responsible for knowing about buildings, tracking selections in the address list, obtaining building information, and updating the apartment number list. We also note that Building must provide access to apartment numbers and this is achieved by accessing Apartment objects. Finally, each Apartment object must be able to access its number. We record all new information on the corresponding CRC cards.

We leave the remaining scenarios as an exercise. As you complete a scenario, don't forget to update the CRC cards. Note that the items starting *User* do not changed much because our focus is expanding the *System* response and identifying how it is implemented by the collaboration of candidate classes.

Assuming that we have completed our analysis of the existing scenarios, we now convert the information recorded on CRC cards into the following format:

Apartment: I hold information about an apartment and provide access to it.
Components:
- number of bedrooms
- rent
- tenant

Responsibilities	*Collaborators*
• Create new apartment	
• Access (modify or return) component information.	Tenant

Building: I hold information about a building and provide access to it.
Components:
- address
- list of apartments

Responsibilities	*Collaborators*
• Create new apartment	
• Access (modify or return) component information.	Apartment

PropertyManager: I am the application model responsible for opening and closing the user interface and propagating information between the user and the domain model. I know about all buildings currently managed
Components: list of buildings
- list of buildings

Responsibilities	*Collaborators*
• Open application and display existing data.	Building

• Close application and save existing data.	Building
• Obtain information about selected building	Building
• Obtain information about selected apartment	Building

Tenant: I hold information about a tenant and provide access to it..
Components:
- name
- phone number
- number of co-occupants

Responsibilities	*Collaborators*
• Create new tenant	
• Access (modify or return) component information	

Draw an Object Model

A picture is worth a thousand words and we conclude with a diagram depicting functional relationships among classes. This Object Model diagram (Figure 2.9) helps us understand why we need the classes that we identified and how they relate to one another. Lines with arrows and legend depict the essence of a relationship between two classes, the * symbol means 'contains zero or more' as in 'a building contains zero or more apartments. When the * is missing, the implication is that exactly one object is being referred to.

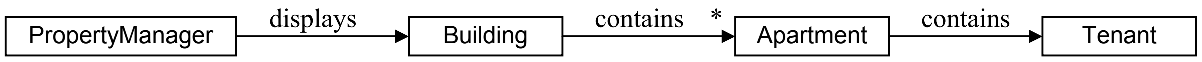


Figure 2.9. Object Model of Electronic Book showing relations between classes.

Testing

To conclude Preliminary Design, we must check that the identified classes and responsibilities are capable of implementing all our scenarios. To do this, redistribute and re-execute the scenarios, making sure that all classes and responsibilities needed to execute all conversations are recorded on the CRC cards. Check that all classes and all responsibilities are used and if not, determine whether they should be deleted or whether scenarios should be corrected or added.

When you are finished testing and the Preliminary Design is confirmed, transfer the information from CRC cards to the format used above and proceed to Design Refinement to identify further details and construct class hierarchies.

2.3.3 Design Refinement

At this stage, we will first check whether we have classes with enough shared behavior to warrant superclasses. We will then draw hierarchy diagrams showing the place of each new class in the class tree, update class specifications making sure that they contain enough detail for implementation, and redraw an Object Diagram if necessary. Finally, we will test that the final model works.

What is the place of our classes in the class hierarchy?

We will now examine all identified classes, check whether any of them share enough behavior to warrant declaring superclasses, and draw a class hierarchy diagram.

All our classes are very closely related but the question is what kind of relationship they have. Texts about OOA/D often distinguish two kinds of relationships: *is-a* and *has-a*. Class A *is-a-kind-of* class B if it is a specialization. As an example, a Lion is a special kind of a Mammal, a Fraction is a special kind of a Number, and a String is a special kind of Collection of objects. Obviously, *is-a* calls for subclassing and this is why a Fraction is a subclass of Number, and why String is a subclass of Collection.

The has-a relationship means containment: A has-a B means that instances of A contain instances of B. As an example, a Car has a Motor and a Chassis, a Fraction has a numerator Integer and a denominator Integer, and a Window has a label, a size, a position, and other properties. A has-a B clearly does not mean subclassing: A Motor is not a special kind of a Car, it's a component of a Car. An integer denominator is not a special kind of Fraction but its part. And a position certainly is not a special kind of a window but one of the necessary components that a Window must have so that it can draw itself.

Considering this perspective, what is the relationship between PropertyManager, Building, Apartment, and Tenant ? We determined that a PropertyManager object knows about the buildings that it manages. The relationship between PropertyManager and Building is thus a has-a relationship – containment and not subclassing. Similarly, an Apartment is not a kind-of building but a building contains an apartment. Apartment is thus a component of Building but not its subclass. And similarly, a Tenant is a part of an Apartment (an Apartment has-a Tenant) but not its special case. Consequently, a Tenant is not a subclass of Apartment but rather its component. As for the rest, the remaining relationships are simply non-existent: A Tenant, for example, is certainly not a special kind of Building, and in our design it is also not a component of a Building. In our design, we decided that a Tenant is a component of Apartment and we will thus not find it listed among the components of Building.

We conclude that there are no is-a (and thus no inheritance) relationship among our classes. But what about some shared meaning or structure? As an example, checking accounts and savings accounts are both accounts (shared meaning) and share a lot of the same properties such as the owner, balance, and number. It thus makes sense that a SavingsAccount is-an Account and so is a CheckingAccount, and both should thus have an class Account for their shared abstract superclass.

In our case, there is no such shared meaning, structure and functionality. The meaning of Apartment is different from the meaning of Building, their components are quite different, and the lists of their responsibilities have very little in common. This means that it does not make sense to define an abstract class to represent the shared meaning and factor out the shared components and behaviors. And the same holds for the remaining classes.

If Building is not a subclass of any new class, then which existing class should be its superclass (because in Smalltalk, every class must have a superclass – except class Object). The classes in the library are mostly system classes such as Number, String, or Window, for use by the environment and their meaning, structure, and responsibilities have nothing in common with Building. Since we have not yet added anything to the class library, we cannot make Building a subclass of any related class of our own and so we decide that Building must be a subclass of Object because Object is the direct or indirect superclass of every class. The same applies to Apartment and Tenant.

The situation with PropertyManager is different because a PropertyManager *is an* application model. This means that PropertyManager must be a subclass of class ApplicationModel.

This completes our investigation of class relationships and our decisions are summarized in the Class Hierarchy Diagram in Figure 2.10. The hashed line indicates additional intermediate classes in VisualWorks library that are not shown. Rectangles with thick border are new classes that we will have to implemented, rectangles with thin borders are classes already in VisualWorks library.

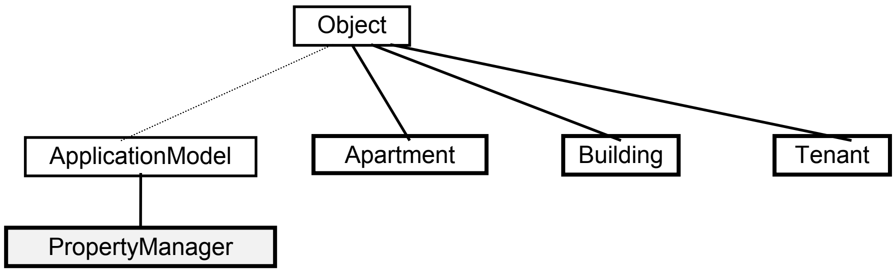


Figure 2.10. Class Hierarchy Diagram for Electronic Book.

<u>Refine class descriptions</u>

Our next task is to examine existing class descriptions and add missing details. The final class descriptions are shown below; we use Smalltalk conventions for naming although they will only be introduced in the next chapter. Note that the names of certain classes such as Integer or Collection, are not quite correct but we don't know better at this point of the book. We are also leaving out certain responsibilities of PropertyManager required for opening and closing the application because they require some knowledge of class ApplicationModel which is covered in Chapter 6.

Apartment: I hold information about an apartment and provide access to it.
Components:
- bedrooms (Integer) number of bedrooms
- rent (Number)
- tenant (Tenant)

Responsibilities	*Collaborators*
• Create new apartment	
• message new	
• Access (modify or return) component information.	
• bedrooms – return the value of bedrooms	
• rent – returns the value of rent	
• tenant – returns the value of tenant	
• bedrooms: anInteger – sets the value of bedrooms to anInteger	
• rent: aNumber – sets the value of rent to anInteger	
• tenant: aTenant– sets the value of tenant to aTenant	

Building: I hold information about a building and provide access to it.
Components:
- be address (String)
- apartments (Collection) list of apartments

Responsibilities	*Collaborators*
• Create new building	
• new	
• Access (modify or return) components	Apartment
• address – returns the value of address	
• apartments – returns the value of apartments	
• address: aString – sets the value of address to aString	
• apartments: aCollection – sets the value of apartments to aCollection	

PropertyManager: I am the application model responsible for opening and closing the user interface and propagating information between the user and the domain model. I know about all buildings currently managed

Components:
- buildings (Collection) list of buildings
- We will also need variables supporting the window widgets but this requires some understanding of user interface components in VisualWorks

Responsibilities	*Collaborators*
• Open application and display existing data.	Building
• open	
• Close application and save existing data.	Building
• Obtain information about selected building	Building
• buildings – returns the value of buildings	
• Obtain information about selected apartment	Building

Tenant: I hold information about a tenant and provide access to it.
Components: name (String), phone (String), occupants (Integer) number of co-occupants

• Close application and save existing data.	Building

- name (String)
- phone (String)
- occupants (Integer) number of co-occupants

Responsibilities	*Collaborators*
• Access (modify or return) component information	
• name – returns the value of name	
• phone – returns the value of phone	
• occupants – returns the value of occupants	
• name: aString – sets the value of address to aString	
• phone: aString – sets the value of phone to aString	
• occupants: aNumber – sets the value of occupants to aNumber	

Refine Object Model

There is no new information because we have not added any new classes and the Object Model diagram thus remains unchanged.

Our design is now complete and we can start implementing it. We are not as optimistic as to assume that we will not have to make any modifications but the design is good enough to allow us to start implementation with a good understanding of what we are doing.

Implementation should now be relatively easy for anybody with sufficient knowledge of VisualWorks Smalltalk. When the prototype is finished, we will show it to the client and obtain corrections and additional details and scenarios. We will then update the design and implementation, and possibly do another specification-analysis-design-implementation iteration. When we are finished, writing user documentation from the detailed specification and scenarios should be easy. It should be noted that some authors recommend writing the manual as early as in the specification phase.

Implementation

An experienced Smalltalk programmer will not have any difficulty implementing our design. Although some details are undoubtedly missing, the overall plan is sound and most of the details resolved.

Conclusion

We started with a rather complete specification, clarified the scenarios and the user interfaces, and identified classes and details of their responsibilities. The procedure was based on progressive refinement, checks, tests, and the use of various perspectives to ensure that no information was missed or left unused. We had to make some decisions that had to be confirmed with the client.

Main lessons learned:

- is-a (or is-a-kind-of) and has-a are two concepts that are very helpful in determining the relationship between two classes.
- A is-a-kind-of B means that class A is a special class of B and should be defined as its subclass.
- A has-a B means that A has an instance of B as its component. No subclassing is implied.

Exercises

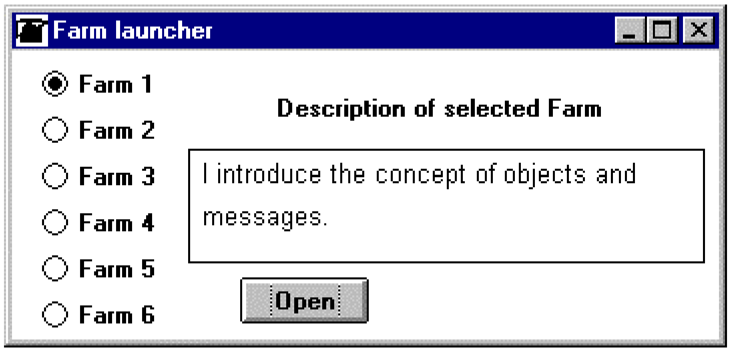
1. Execute the example, completing all CRC cards, writing all missing descriptions, and performing all required tests.
2. Read and explain the Object Model diagram.
3. After seeing the prototype, the client has realized that an apartment should have a list of its occupants and that an address should include postal code and city name. Modify the design accordingly. How much impact did the changes have on the original design? Do you think that we should have designed the classes differently?

2.4 Example 2 – The Farm Program

In this section, we will explore how the Farm program introduced in Chapter 1 could be designed.

2.4.1 System Description

The Farm program is a multi-level simulation of a Farm whose purpose is to illustrate certain concepts of object-oriented programming and Smalltalk. Access to individual levels is through a launcher which first opens with Farm 1 selected (Figure 2.12).



I introduce the concept of objects and messages.

Figure 2.12. Farm Launcher.

When the user selects a farm level and clicks Open, a Farm window as in Figure 2.13 opens.

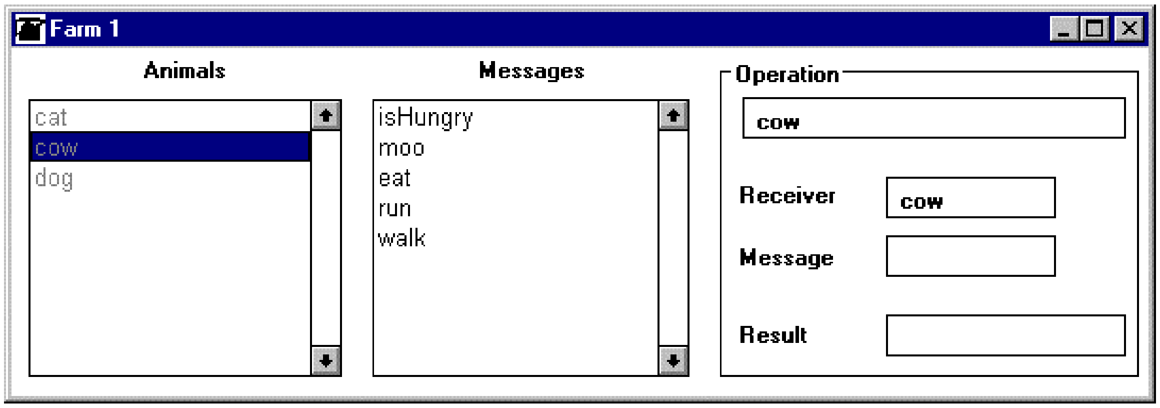


Figure 2.13. Farm 1 window.

The first three levels of Farm share the same user interface, and so do levels 4 and 5. Level 6 uses another user interface.

All six levels are based on three kinds of animals – cat, cow, and dog. In the first level, cat understands commands isHungry, meow, eat, run, and walk . The cow does not understand meow but does understand moo, and dog understands bark instead. When the user clicks an animal, the window displays the commands that the animal understands, and when the user clicks a command, the window shows the result of executing it. Command isHungry produces true or false, meow produces 'I meow', moo produces 'I moo', and bark produces 'I bark'. Command eat produces 'hungry = false', run produces 'I am running' from cat and dog but 'I am too full' from cow, and walk produces 'I am walking' from all three animals.

Level 2 interface is the same as for level 1 but the animals understand two new command - commands color (returns 'white' for cat, 'brown' for cow, and 'black' for dog) and name ('Pussie' for cat, 'Jerky' for cow, and 'Rob' for dog).

Level 3 introduces three new objects, all of them factories for producing animals: Cat is a cat factory, Cow is a cow factory, and Dog is a dog factory. Each of these understands a single command called new and executes it by creating a new animal with name and color specified by user. Consecutive animals of the same kind are numbered consecutively as in cat1, cat2, and all animals understand the same commands as at level 2.

Levels 4, 5, and 6 perform the same functions as level3 and additional introduce various Smalltalk concepts and generate Smalltalk code. Their design should be considered in general terms now but details should be left for the second iteration.

Scenarios and high-level conversations

Since each level is an extension of the previous level, we don't have to separate scenarios into consecutive levels. We will detail only a few of the numerous scenarios because most scenarios are identical except for the selected command and the corresponding result. The remaining scenarios are left as an exercise.

Scenario 1: *User starts Farm.*
High-level conversation:
1. *User* enters and executes a Smalltalk expression.
2. *System* opens launcher window as in Figure 2.12.

Scenario 2: *User closes Farm.*
High-level conversation:
1. *User* clicks the window closing button.
2. *System* closes all Farm windows and ends the program.

Scenario 3: *User opens Farm 1.*
High-level conversation:
1. *User* clicks *Farm 1* radio button.
2. *System* takes note of the new setting.

3. *User clicks Open.*
4. *System opens Farm 1 user interface.*

Scenario 4: *User sends 'meow' to the cat.*
High-level conversation:
1. *User* clicks *cat* in the Animals list.
2. *System* displays commands understood by cat and updates the rightmost part of the window.
3. *User* clicks *meow*.
4. *System* displays results in the right-most part of the window, restores the animal list, and erases the list of commands.

Scenario 5: *User sends 'new' to CatFactory.*
High-level conversation:
1. *User* clicks *Cat* in the Animals list.
2. *System* displays *new* in the command list and updates the rightmost part of the window.
3. *User* clicks *new*.
4. *System* requests cat name and color, creates a numbered cat, updates the rightmost part of the window, erases the command list, and displays the new list now including the name of the new numbered cat.

Context Diagram

The information gathered above seems sufficient to identify external actors and subsystems and we use it to construct the Context Diagram in Figure 2.14.

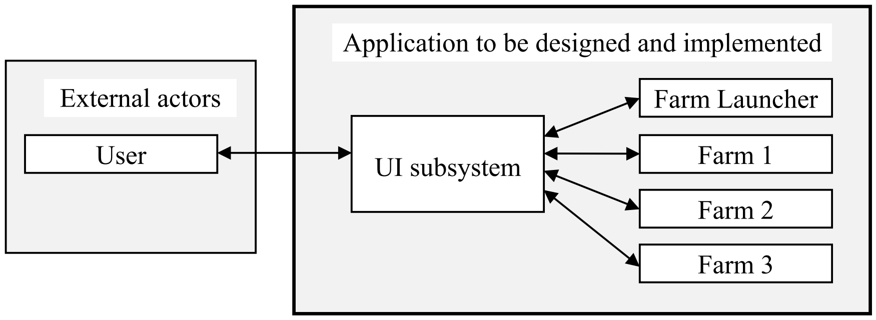


Figure 2.14. Context Diagram of the first iteration of the Farm program.

Glossary

Farm Launcher – main part of Farm providing access to different Farm levels.
Farm 1, Farm 2, Farm 3 – different levels of the Farm program.

2.4.2 Preliminary design

Identify classes

As usual, we start by textual analysis of the specification and the scenarios.

Specification

The Farm program is a multi-level simulation of a Farm whose purpose is to illustrate certain concepts of object-oriented programming and Smalltalk. Access to individual levels is through a launcher which first opens with Farm 1 selected. When the user selects a farm level and clicks Open, a Farm window opens.

The first three levels of Farm share the same user interface, and so do levels 4 and 5. Level 6 uses another user interface.

All six levels are based on three kinds of animals – cat, cow, and dog. In the first level, cat understands commands isHungry, meow, eat, run, and walk . The cow does not understand meow but does understand moo, and dog understands bark instead. When the user clicks an animal, the window displays the commands that the animal understands, and when the user clicks a command, the window shows the result of executing it. Command isHungry produces true or false, meow produces 'I meow', moo produces 'I moo', and bark produces 'I bark'. Command eat produces 'hungry = false', run produces 'I am running' from cat and dog but 'I am too full' from cow, and walk produces 'I am walking' from all three animals.

Level 2 interface is the same as for level 1 but the animals understand two new command - commands color (returns 'white' for cat, 'brown' for cow, and 'black' for dog) and name ('Pussie' for cat, 'Jerky' for cow, and 'Rob' for dog).

Level 3 introduces three new objects, all of them factories for producing animals: Cat is a cat factory, Cow is a cow factory, and Dog is a dog factory. Each of these understands a single command called new and executes it by creating a new animal with name and color specified by user. Consecutive animals of the same kind are numbered consecutively as in cat1, cat2, and all animals understand the same commands as at level 2.

Levels 4, 5, and 6 perform the same functions as level3 and additional introduce various Smalltalk concepts and generate Smalltalk code. Their design should be considered in general terms now but details should be left for the second iteration.

Next, we examine the selected nouns and add some from our understanding of the situation:

Animal – we don't see any need for this class yet, the program always deals with concrete animals.
Cat – yes, include it; the user interacts with it, it has its own behavior.
Cat Factory – the user interacts with it and it has its own behavior. Yes, include it.
Command – this is a part of each animal's and factory's behavior. Don't include it.
Cow – yes, include for the same reason as Cat.
Cow Factory - include for the same reason as Cat Factory.
Dog - include for the same reason as Cat.
Dog Factory - include for the same reason as Cat Factory.
Farm – according to the specification, we always deal with a specific farm. Don't include.
Farm 1 – include; it has its own user interface and sometimes forces animals to behave differently.
Launcher – include. Has its own user interface and associated behaviors. In fact, it starts the Farm application.
Level 2, etc. – this is just a different name for Farm 2, etc. Include, but under the names Farm 2, etc.
Window – don't include. This is just a reference to the user interface and we have included all user interfaces in the Farm classes and in the Launcher.

We conclude that we will attempt to implement the problem with the following classes:

- Cat
- CatFactory
- Cow
- CowFactory
- Dog
- DogFactory
- Farm1
- Farm2
- Farm3
- FarmLauncher

Identify class responsibilities

We will now examine our scenarios and expand high-level conversations into class-level conversations.

Scenario 1: *User starts Farm.*
High-level conversation:
1. *User* enters and executes a Smalltalk expression.
2. *System* opens launcher window as in Figure 2.12.

Class-level conversation:
1. *User* enters and executes FarmLauncher open.
2. FarmLauncher opens launcher window as in Figure 2.12.

Scenario 2: *User closes Farm.*
High-level conversation:
1. *User* clicks the window closing button.
2. *System* closes all Farm windows and ends the program.

Class-level conversation:
1. *User* clicks the window closing button.
2. FarmLauncher closes all Farm windows and ends the program.

Scenario 3: *User opens Farm 1.*
High-level conversation:
1. *User* clicks *Farm 1* radio button.
2. *System* takes note of the new setting.
3. *User* clicks *Open.*
4. *System* opens Farm 1 user interface.

Class-level conversation:
1. *User* clicks *Farm 1* radio button.
2. *System* FarmLauncher takes note of the new setting.
3. *User* clicks *Open.*
4. FarmLauncher asks Farm1 to open.
5. Farm1 creates an instance of itself and asks each of Cat, Cow, and Dog Factory to create an instance of themselves.
6. Cat, Cow, and Dog to create an instance of themselves.
7. Farm1 asks each potential command receiver for a list of commands that it understands.
8. Cat, Cow, and Dog report their commands. Farm1 asks each potential command receiver for a list of commands that it should not understand. (Farm 1 does not understand color and name.)
9. Cat, Cow, and Dog report commands that they do not understand.
10. Farm1 opens its user interface.

This scenario is perhaps less obvious than you would expect. Our goal in designing it was to deal with possible future modifications. As it is, we can add new animals without changing Farm1 because Farm1 does not depend directly on animals. (Note that adding a new animal still requires a change in the list of receivers that Farm1 contacts but we will not worry about this.) We could also change the set of commands that animals understand or should not understand in Farm 1 and class Farm1 will not have to be changed. Our solution is not, of course, the only one possible.

Scenario 4: *User sends 'meow' to the cat.*
Comment: Although the resulting behavior is the same for all farms, we must select a specific farm for the purpose of the conversation. Let's select Farm 2.
High-level conversation:
1. *User* clicks *cat* in the *Animals* list.
2. *System* displays commands understood by cat and updates the rightmost part of the window.
3. *User* clicks *meow*.

4. *System* displays results in the right-most part of the window, restores the animal list, and erases the list of commands.

Class-level conversation:

1. *User* clicks *cat* in the *Animals* list.
2. Farm2 takes note that *cat* is selected as the current animal.
3. Farm2 displays commands understood by cat, and updates the rightmost part of the window.
4. *User* clicks *meow*.
5. Farm2 requests appropriate response to command *meow* from the current animal (a Cat).
6. Cat returns the appropriate response and Farm2 displays it as a part of results in the right-most part of the window, restores the animal list, and erases the list of commands.

Scenario 5: *User sends 'new' to Cat Factory.*
Note: Assume Farm 3.
High-level conversation:

1. *User* clicks *Cat* in the *Command Receivers* list.
2. *System* displays *new* in the command list and updates the rightmost part of the window.
3. *User* clicks *new*.
4. *System* requests cat name and color, creates a numbered cat, updates the rightmost part of the window, erases the command list, and displays the new list now including the name of the new numbered cat.

Class-level conversation:

1. *User* clicks *Cat* in the *Command Receivers* list.
2. Farm3 takes note that a CatFactory is selected as the current command receiver. It displays the commands that it understands in the command list and updates the rightmost part of the window.
3. *User* clicks *new*.
4. Farm3 asks CatFactory to respond to the message.
5. CatFactory requests cat name and color
6. CatFactory creates a numbered cat, updates the rightmost part of the window, erases the command list, and displays the new list now including the name of the new numbered cat.

With this information, we can now write detailed descriptions of all required classes.

Cat: Simulated cat. Knows the commands it understands and how to respond to them.. Knows commands that it should not understand in Farm1. A domain object.
Components:

- list of commands understood
- list of commands not understood in Farm1

Responsibilities	*Collaborators*
• Creation	
• Create new an instance of Cat.	
• Know commands	
• Know list of all commands (color, name, isHungry, run, walk, eat, meow)	
• Know commands forbidden in Farm1(color, name)	
• Respond to commands	
• Respond to color, name, isHungry, run, walk, eat, meow	

CatFactory: Creates new Cat instances.
Components:

Responsibilities	*Collaborators*
• Creation	
• Create new instance of yourself	
• Create new Cat.	Cat

Note that the first creation message is used by Farm3 when it needs a CatFactory – it simply creates an instance of itself - a CatFactory; this message is a *class* message because we send it to *class*

CatFactory. The second creation message is a request to an existing CarFactory (and *instance* of CarFactory) to create a new cat; since we send it to an instance of CarFactory, it is an *instance* message. It must use Cat as its collaborator because only Cat knows how to create a new instance of itself. The usage diagram in Figure 2.15 further illustrates this distinction: The first bold **new** message is a message to *class* CatFactory (so it is a *class* message) and creates an instance of CatFactory. The second bold **new** message is a message to the previously created *instance* of CatFactory (so it is an *instance* message) and its result is a new instance of Cat.

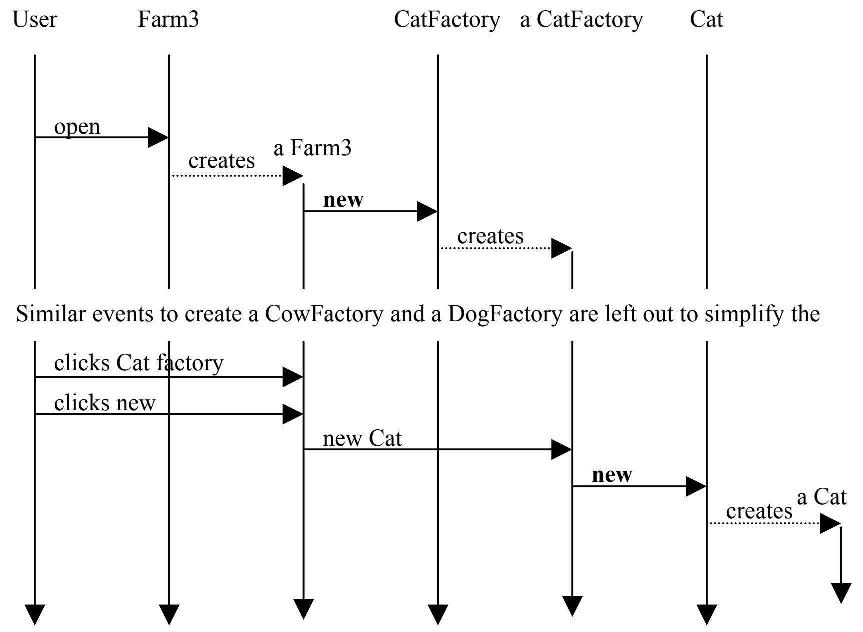


Figure 2.15. Opening *Farm 3* and creating a new cat.

Cow: Simulated cow. Knows the commands it understands and how to respond to them.. Knows commands that it should not understand in *Farm 1*. A domain object.
Components:
- list of commands understood
- list of commands not understood in *Farm 1*

Responsibilities	*Collaborators*
• Creation	
• Create new an instance of Cow.	
• Know commands	
• Know list of all commands (color, name, isHungry, run, walk, eat, moo)	
• Know commands forbidden in Farm1(color, name)	
• Respond to commands	
• Respond to color, name, isHungry, run, walk, eat, meow	

CowFactory: Creates new Cow instances.
Components:

Responsibilities	*Collaborators*
• Creation	
• Create new instance of yourself	
• Create new Cow	Cow

Dog: Simulated dog. Knows the commands it understands and how to respond to them.. Knows commands that it should not understand in Farm1. A domain object.

Components:
- list of commands understood
- list of commands forbidden in Farm1

Responsibilities	*Collaborators*
• Creation	
• Create new an instance of Cat.	
• Know commands	
• Know list of all commands (color, name, isHungry, run, walk, eat, bark)	
• Know commands forbidden in Farm1(color, name)	
• Respond to commands	
• Respond to color, name, isHungry, run, walk, eat, meow	

DogFactory: Creates new Dog instances.
Components:

Responsibilities	*Collaborators*
• Creation	
• Create new instance of yourself	
• Create new Dog.	Dog

Farm1: Application model, ties user interface to domain objects - input of parameters, output of results.
Components: As required by user interface

Responsibilities	*Collaborators*
• Opening	
• create and initialize a new instance of yourself, open window	Cat, Cow, Dog
• User interface actions	
• respond to receiver selection	
• respond to command selection	Cat, Cow, Dog

Farm2: Application model, ties user interface to domain objects - input of parameters, output of results. Same as Farm1 but does not filter out forbidden messages.
Components: As required by user interface.

Responsibilities	*Collaborators*
• Opening	
• create and initialize a new instance of yourself, open window	Cat, Cow, Dog
• User interface actions	
• respond to receiver selection	
• respond to command selection	Cat, Cow, Dog

Farm3: Application model, ties user interface to domain objects - input of parameters, output of results. Same as Farm1 but does not filter out forbidden messages.
Components: As required by user interface.

Responsibilities	*Collaborators*
• Opening	
• create and initialize a new instance of yourself, open window	CatFactory, CowFactory, DogFactory
• User interface actions	
• respond to receiver selection	
• respond to command selection	CatFactory, CowFactory, DogFactory Cat, Cow, Dog

FarmLauncher: Application model, provides access to various Farm levels.
Components: As required by user interface.

Responsibilities	*Collaborators*

- Opening
  - create and initialize a new instance of yourself, open main window
- User interface actions
  - In response to Open, open appropriate farm window Farm1, Farm2, Farm3

Test our descriptions with all scenarios to check whether they are complete and consistent and make any necessary corrections. Record in a formal format.

Object Diagram

Figure 2.16 shows the relationships between the identified classes.

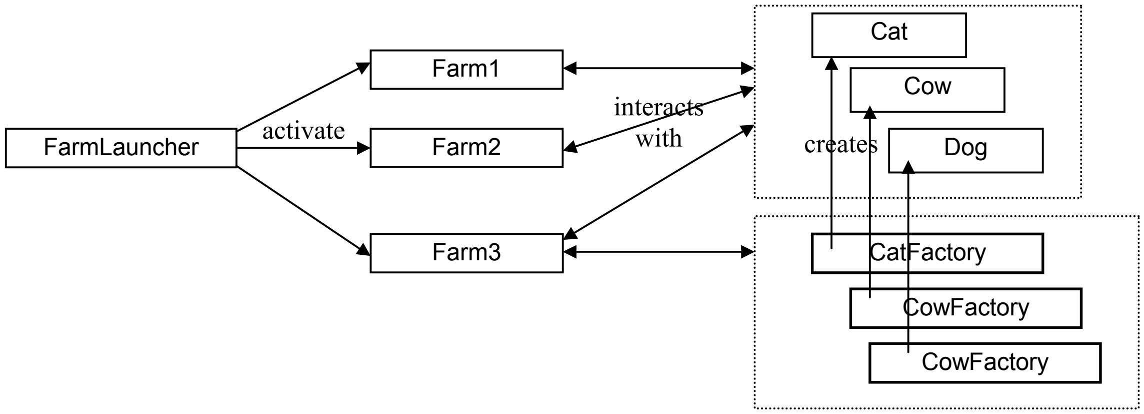


Figure 2.16. Object Diagram of Farm. Only classes to be designed are shown.

2.4.3 Design Refinement

Re-examination of candidate classes and their placeclasses in class hierarchy

Let's examine our candidate classes hierarchically in the order animals, animal factories, farms, and launcher.

The three *animals* are obviously very similar in all important respects: They all are the same kind of objects (animals), they share the same properties (color, name, and isHungry), and they have almost identical functionality - only their form of communication is different (meow, moo, and bark), and they perform some of the same functionality differently (running). The three animals are thus a perfect choice for subclassing and we will make them all subclasses of an abstract class called Animal. Animal does not have any related classes in the existing hierarchy and it will thus be a subclass of Object.

The three *animal factories* are also extremely similar – and simple – their only function is to ask an animal class to create an instance of itself. We don't think that this warrants creating three separate classes – and then a new animal factory class every time we decide to add a new animal. We will thus merge all animal factory classes into one called AnimalFactory and generalize its animal creation message so that it can create an animal of any specified class. This solution will make further expansion relatively easy. As an example, if we decide to add a new animal such as sheep, we don't have to create a new factory class to create sheep. AnimalFactory does not have any related classes in the existing hierarchy and it will thus be a subclass of Object.

When we think about it, we don't actually have to create an instance of AnimalFactory when running a Farm program because we never need more than one animal factory at a time. if we use this approach, the animal creation message is always sent to the AnimalFactory class and it is thus a *class* message. This is illustrated in the revised diagram of Scenario 3 in Figure 2.17.

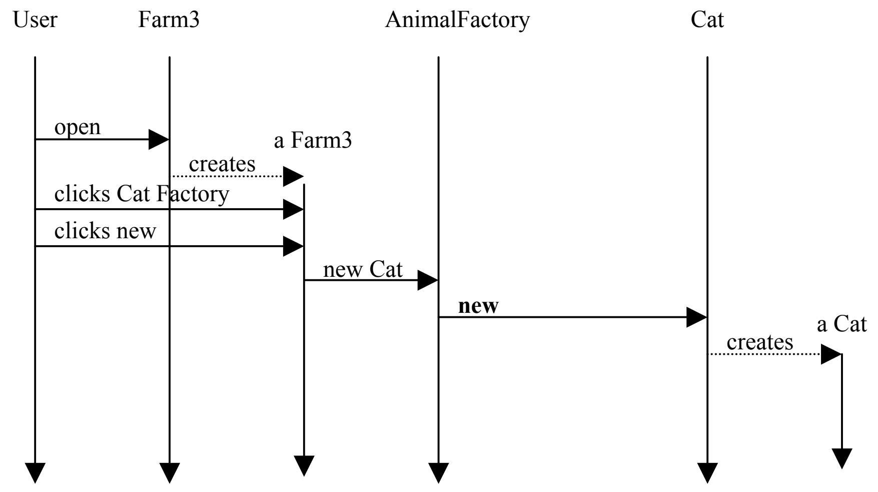


Figure 2.17. Opening *Farm3* and creating a new cat using new design.

We could go even further and decide that since the only function of AnimalFactory is to send a message to an animal class to create an instance of itself, this responsibility could be left to the farm. When the user clicks the Cat Factory, for example, the farm could directly send the new message to class Cat. This approach is certainly possible but we don't like it because it breaks the uniformity of the behavior of farms: Messages to animals would be sent to animal instances but messages to factories would be executed by the farm. Because of this, we will keep AnimalFactory even though its functionality is minimal.

The *farm* category includes three kinds of farms. All are essentially the same – they all have the same interface and the only difference is that Farm1 understands fewer commands and Farm3 knows about animal factories and does not open with any animals. This behavior could also be implemented with a single class. However, we must remember that we will need three more kinds of farms and that these will require two different interfaces and three different sets of behaviors. If we wanted to implement all six farms with a single class, this class would be more complicated than we are willing to accept and we will thus implement each farm with its own class.

Now what about the class hierarchy of farms. First, all of them control a user interface so they will all be subclasses of ApplicationModel. Class Farm2 is identical to Farm1 except that it does not restrict the set of animal commands, and class Farm3 only extends Farm2. We thus decide to make Farm2 a subclass of Farm1, and Farm3 a subclass of Farm2. Farm1 will define the user interface that Farm2 and Farm3 will inherit, as well as the bulk of farm functionality.

Finally the *launcher*. The launcher has a user interface but its purpose, user interface, and behavior is totally unrelated to farms. It will thus stand alone as a subclass of ApplicationModel.

The Class Hierarchy Diagram resulting from our analysis is shown in Figure 2.18.

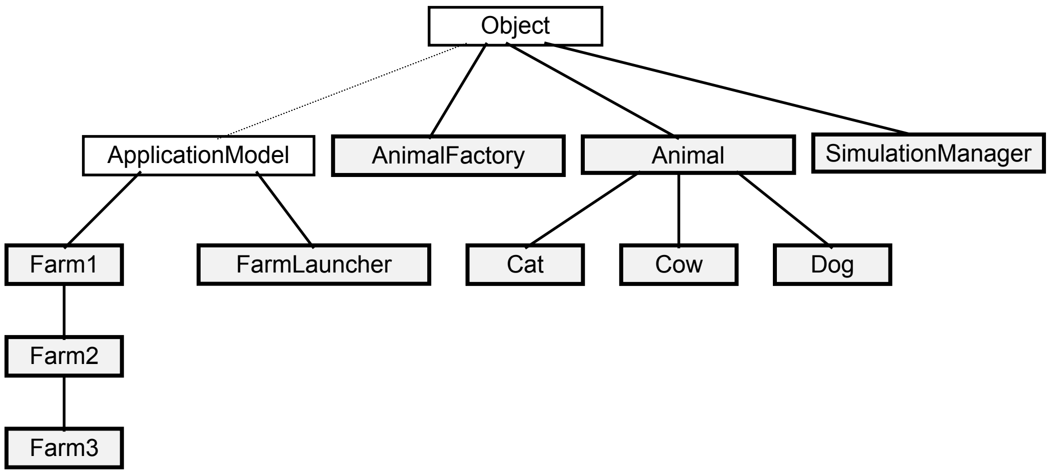


Figure 2.17. Class hierarchy diagram for Farm. Classes that must be constructed have thick borders.

Refinement of class descriptions

After this analysis, we are now ready for the final class descriptions:

Animal: Abstract superclass of all animals.
Superclass: Object
- *Components*: color (string), name (string), isHungry (true or false), commands (list of commands understood), privateCommands (list of commands not understood in Farm1)

Responsibilities *Collaborators*
- Creation
  - Create new instance of yourself
- Respond to shared commands
  - Methods color, name, isHungry, walk, eat
- Know names of all shared commands
  - commands – returns (color, name, run, walk, eat)
- Know commands forbidden in Farm1(color, name, isHungry)

AnimalFactory: Creates new animals.
Superclass: Object
Components:
Responsibilities *Collaborators*
- Creation
  - Create new animal
    - method newAnimal: anAnimal – class method Cat, Cow, Dog

Cat: Simulated cat. Knows the commands it understands and how to respond to them.. Knows commands that it should not understand in Farm1. A domain object.
Superclass: Object
Components: inherited
Responsibilities *Collaborators*
- Creation - inherited
- Know commands
  - Know list of all commands
    - commands – add meow to inherited list
- Respond to private commands and commands behaving differently
  - Methods meow, run

Cow: Simulated cow. Knows the commands it understands and how to respond to them. Knows commands that it should not understand in Farm1. A domain object.
Superclass: Object
Components: inherited

Responsibilities	*Collaborators*
• Creation - inherited	
• Know commands	
• Know list of all commands	
• commands – add moo to inherited list	
• Respond to private commands and commands behaving differently	
• Methods moo, run	

Dog: Simulated dog. Knows the commands it understands and how to respond to them.. Knows commands that it should not understand in Farm1. A domain object.
Superclass: Object
Components: inherited

Responsibilities	*Collaborators*
• Creation - inherited	
• Know commands	
• Know list of all commands	
• commands – add bark to inherited list	
• Respond to private commands and commands behaving differently	
• Methods bark, run	

Farm1: Application model, ties user interface to domain objects - input of parameters, output of results.
Superclass: ApplicationModel
Components: As required by user interface

Responsibilities	*Collaborators*
• Opening	
• create and initialize a new instance of yourself, open window	Cat, Cow, Dog
• Define user interface	
• User interface actions	
• respond to receiver selection	
• respond to command selection	Animal, Cat, Cow, Dog

Farm2: Application model, ties user interface to domain objects - input of parameters, output of results. Same as Farm1 but does not filter out forbidden messages.
Superclass: Farm1
Components: As required by user interface.

Responsibilities	*Collaborators*
• Opening	
• expand inherited initialization	
• User interface actions	
• respond to receiver selection inherited - inherited	
• respond to command selection - inherited	

Farm3: Application model, ties user interface to domain objects - input of parameters, output of results. Same as Farm1 but does not filter out forbidden messages.
Superclass: Farm2
Components: As required by user interface.

Responsibilities	*Collaborators*
• Opening	
• create and initialize a new instance of yourself,	CatFactory, CowFactory, DogFactory

open window

- User interface actions
    - respond to receiver selection - inherited
    - respond to command selection - inherited but depends also on AnimalFactory

FarmLauncher: Application model, provides access to various Farm levels.
Superclass: ApplicationModel
Components: As required by user interface.

Responsibilities	*Collaborators*
• Opening	
• create and initialize a new instance of yourself, open main window	
• User interface actions	
• In response to Open, open appropriate farm window	Farm1, Farm2, Farm3

Object Model Diagram

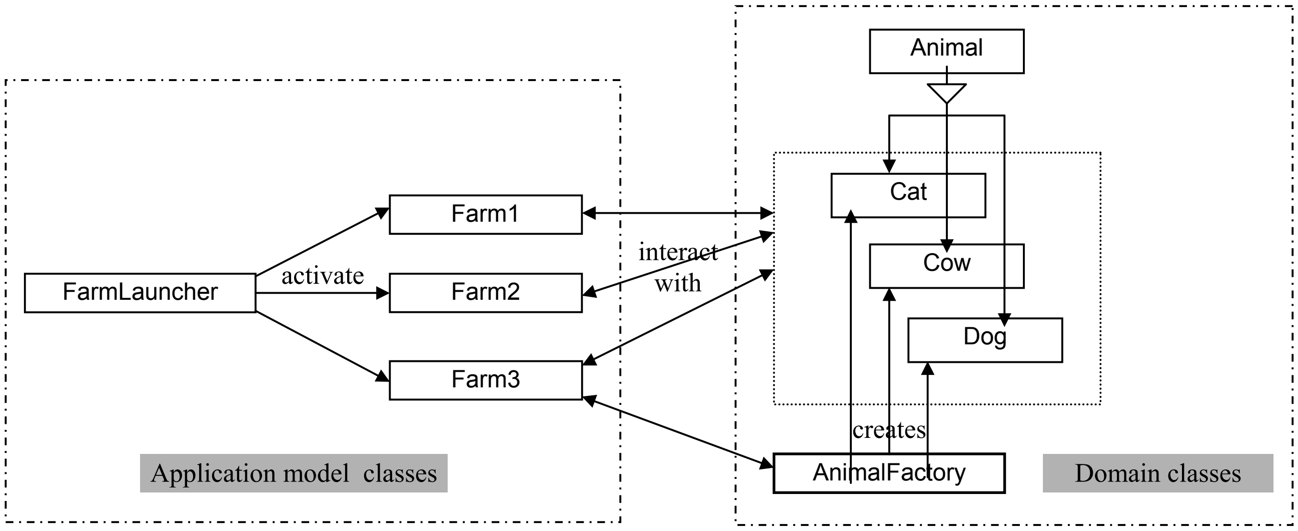


Figure 2.19. Final Object Model diagram. The rectangle below class Animal means that Animal is a superclass of Cat, Cow, and Dog.

Implementation

Our plan for the solution of the problem is now clear and detailed enough and an experienced Smalltalk programmer will not have any problem implementing it. There are, undoubtedly, some details that we have not considered and that will have to be added, possibly even some mistakes that will emerge during implementation, but these will not be difficult to add and they will not affect the overall plan.

In closing

We must admit that our own original design was different because we lumped Cat and CatFactory into Cat, Cow and CowFactory in Cow, and Dog and DogFactory into Dog. This worked but animals and animal factory are more naturally thought of as different objects. Since our experience tells us that it is almost always better to design according to the natural logic of objects, we consider our final design better.

Main lessons learned:

- The initial specification is often vague and incomplete and may require a substantial extension.
- Don't follow the idea of extracting nouns from the specification slavishly. Use any insights that are available such as diagrams and sketches of user interfaces. Use your past experience.
- Watch for possibilities to generalize your design for reuse.
- It is usually possible to implement the same problem in several different ways. Try to use the design that separates the objects in the most natural way.
- Superclasses are usually abstract but there are exceptions.
- Most of the time, we create instances to instantiate behaviors. There are, however, situations in which all the desired behavior can be implemented by the class itself.

Exercises

1. Repeat the example, completing all CRC cards, writing all missing scenarios, descriptions, and performing all required tests.
2. Read and explain the Object Model diagram.
3. Write your own detailed specification and perform preliminary and final design of the following problems:
   a. Video store inventory program. Keeps track of customers and videos.
   b. Checkers game.

Conclusion

The essence of object-oriented software development is finding objects and identifying their responsibilities. Although there does not exist a set of rules that guarantees finding the required objects mechanically, a number of methodologies are available that make the task easier and improve the chances of obtaining a satisfactory result.

A methodology consists of a process, guidelines, and a description of deliverables. A process is a sequence of steps that should be followed to solve the problem. Guidelines capture experiences that may help in finding solutions. A deliverable is an artifact, a document or a software product that must be delivered at the end of a process step. Each methodology lists its deliverables and describes their format.

All development methodologies agree that the basic stages of the development process are requirements specification, analysis, design, and implementation. The acronym OOA/D is used to refer to object-oriented analysis and design, the two essential and most difficult stages of object-oriented development. The boundary between analysis and design in object-oriented development is often fuzzy because they both use the same perspective (objects and messages) and the same notations.

The description of software development as specification-analysis-design-implementation hides the fact that all development must be accompanied by constant testing, and that development constantly uncovers gaps and errors that require corrections of results and deliverables produced by previous stages.

In most practical situations, specification, analysis, design, and implementation are not performed only once. Instead, the whole procedure is repeated several times to obtain better and better understanding of the desired product and to come up with better and better designs. Object-oriented design thus has iterative nature.

The methodology that we use is rather loose and we leave it to you to adjust it to your personal needs. It can be summarized as follows:

Requirements Specification:
- Obtain textual description and possibly a model of the user interface.
- Obtain usage scenarios describing major tasks typically performed by the user.
- Decompose scenarios into high-level conversations describing how a scenario is accomplished as a dialog between the user and the system.

- Construct a Context Diagram showing external actors (outside the scope of current project) and major parts of the system to be developed.
- Construct a Glossary of terms. This Glossary will be continuously updated during the process.

Preliminary Design:
- Use results of Requirements Specification to find candidate classes. Textual analysis of the specification, scenarios, and conversations is the basis but not the only resource for this process.
- Expand high-level scenarios into class-level scenarios describing how system actions are performed in terms of dialogs between classes candidates.
- Use class-level scenarios to develop preliminary class descriptions consisting of a brief outline of the purpose of each class, a preliminary list of its components, and a list of its responsibilities. For each responsibility, identify classes that must collaborate to fulfill the responsibility. Record preliminary versions on CRC cards.
- Draw an Object Model diagram showing the essential relationships between the identified classes.

Design Refinement
- Determine whether any of the identified classes share responsibilities that can be factored out. If so, create abstract classes containing these shared responsibilities. Use is-a-kind-of and has-a relationships to distinguish between situations that require subclassing and situations that require containment.
- Draw a Class Hierarchy Diagram showing the place of all identified classes in the existing class hierarchy.
- Refine class descriptions, refine responsibilities into methods, choose method names, and write a short description of each non-trivial method.
- Revise Object Model diagram and refine it to show subclassing.

Implementation
- Implement detailed class descriptions obtained in the previous stage in the selected programming environment. Write user documentation.

Each step is accompanied by testing.

A methodology is not a ‘silver bullet’ that kills the problem of design and makes design effortless. It is just a set of rules and hints that should help you arrive at a satisfactory solution and produce documentation useful for further development and maintenance.

Finally, it is important to remember that there is no single correct solution and that different designers may produce different working designs. This does not mean, however, that all designs are equally good. The best measures of the quality of design are its extendibility, its maintainabilty, its understandability, and its potential for reuse in other applications.

Terms introduced in this chapter

analysis - development stage following specification and preceding design
artifact – a product such as a document or a program
class-level conversation - conversation expressed as an exchange between the user and the classes implementing the application
Context Diagram - diagram showing which parts of the application are outside the system but collaborate with it, and which parts of the application are to be designed
design - third stage of development resulting in detailed description of classes; followed by implementation
deliverable - a product that must be delivered at a certain point of the development process
has-a - relationship that identifies a class as a component of another class
high-level conversation - conversation expressed in terms of a dialog between the user and the system without consideration of classes
is-a – relationship that identifies a class as a special kind of another class; also *is-a-kind-of*

iterative development - repeated specification-analysis-design-implementation performed to obtain complete understanding of client needs and optimal implementation
life cycle – the sequence of development, usage, and maintenance of a product
methodology - a process, a set of guidelines, and a description of deliverables
Object Model Diagram - a diagram showing the major functional relationships between classes
OOA/D - object-oriented analysis and design
process - sequence of steps to be executed to perform a task
Responsibility Driven Design (RDD) - methodology relying heavily on the use of responsibilities and conversations