OCCASION

This publication has been made available to the public on the occasion of the 50th anniversary of the United Nations Industrial Development Organisation.

DISCLAIMER

FAIR USE POLICY

CONTACT

Please contact publications@unido.org for further information concerning UNIDO publications.

For more information about UNIDO, please visit us at www.unido.org

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION
Vienna International Centre, P.O. Box 300, 1400 Vienna, Austria
Tel: (+43-1) 26026-0 · www.unido.org · unido@unido.org

Object-Oriented Software Development Using UML

Object-Oriented Software Development Using UML

Contents

1 Introduction 1
1.1 Reading List . . . 1
1.2 Introduction to Software Engineering . . . 2
1.3 Questions . . . 4

2 Software Development Process 5
2.1 Software Development Process . . . 5
2.1.1 Requirement capture and analysis . . . 6
2.1.2 System design . . . 8
2.1.3 Implementation and unit testing . . . 9
2.1.4 Integration and system testing . . . 10
2.1.5 Operation and maintenance . . . 10
2.2 The Waterfall Model . . . 10
2.3 Evolutionary Development . . . 11
2.4 Questions . . . 14
2.5 Further Reading . . . 14

3 Introduction to OO Development 15
3.1 The Inherent Complexity of Software . . . 15
3.2 Mastering Complex Systems . . . 17
3.2.1 Examples of complex systems . . . 17
3.2.2 The five attributes of a complex system . . . 20
3.2.3 The generalization-specialization hierarchy . . . 21
3.2.4 Function-oriented and object-oriented methods . . . 22
3.3 The Rest of The course . . . 25
3.4 Questions . . . 25

4 Requirement Capture and Analysis – Use Cases 27
4.1 Understanding requirements . . . 27
4.1.1 Introduction . . . 27
4.1.2 Case study: Point-of-sale . . . 28
4.1.3 Requirements specification . . . 28
4.2 Use Cases: Describing Processes . . . 33
4.2.1 Use case concepts . . . 33
4.2.2 Identifying use cases . . . 34
4.2.3 Writing use cases . . . 35
4.2.4 Essential and real use cases . . . 39
4.2.5 Use-case model and use case diagrams . . . 40
4.2.6 Use cases within a development process . . . 43
4.2.7 Actors and use cases of the POST system . . . 44
4.2.8 Summing up . . . 45
4.3 Questions . . . 46

5 Requirement Capture and Analysis – Conceptual Model 49
5.1 Conceptual Model – Concepts and Classes . . . 49
5.1.1 Conceptual Models . . . 49
5.1.2 Defining terms and modelling notation for a concept . . . 51
5.1.3 Identifying Concepts . . . 52
5.1.4 Conceptual modelling guidelines . . . 55
5.2 Conceptual Model – Associations . . . 56
5.2.1 Strategies for identifying associations . . . 59
5.2.2 The aggregation association . . . 63
5.3 Conceptual Model–Attributes . . . 66
5.3.1 Adding attributes to classes . . . 67
5.3.2 Attributes for the Point-of-Sale System . . . 73
5.4 Steps to Create a Conceptual Model . . . 74
5.5 Recording Terms in the Glossary . . . 75
5.6 Questions . . . 77

6 System Behaviour: System Sequence Diagrams and Operations 81
6.1 System Sequence Diagram . . . 81
6.1.1 System input events and system operations . . . 82
6.1.2 System sequence diagram . . . 83
6.1.3 Recording system operations . . . 85
6.2 Contracts for System Operations . . . 87
6.2.1 Documenting Contracts . . . 88
6.2.2 Contracts for some operations in POST system . . . 89
6.2.3 How to make a contract . . . 91
6.2.4 Contracts and other Documents . . . 92
6.3 Analysis Phase Conclusion . . . 92

7 Design Phase: Collaboration Diagrams 95
7.1 Object Sequence Diagrams . . . 96
7.2 Collaboration Diagrams . . . 96
7.2.1 Examples of collaboration diagrams . . . 97
7.2.2 More notational issues . . . 98
7.3 Creating Collaboration Diagrams . . . 103
7.3.1 Overview of Design Phase . . . 103
7.3.2 Real Use Cases . . . 103
7.3.3 GRASP: Patterns for Assigning Responsibilities . . . 105
7.3.4 GRASP: Patterns of General Principles in Assigning Responsibilities . . . 106
7.3.5 A Design of POST . . . 117
7.3.6 Connecting User Interface Objects to Domain Objects . . . 128
7.3.7 Design Class Diagrams . . . 129
7.4 Questions . . . 133

8 Implementing a Design 137
8.1 Notation for Class Interface Details . . . 137

8.2 Mapping a Design to Code . . . 139
8.2.1 Defining a class with methods and simple attributes . . . 139
8.2.2 Add reference attributes . . . 139
8.2.3 Defining a method from a collaboration diagram . . . 140
8.2.4 Container/collection classes in code . . . 143
8.2.5 Exceptions and error handling . . . 144
8.2.6 Order of implementation . . . 144
8.3 Questions . . . 145

9 Advanced Modelling Concepts and Design Techniques 147
9.1 Iterative Development Process . . . 147
9.1.1 Use case and iterative development . . . 148
9.1.2 Development cycle 2 of POST . . . 148
9.1.3 Extending the conceptual model . . . 152
9.2 Generalisation . . . 152
9.2.1 The notion of generalization . . . 154
9.2.2 The meaning of the generalization-specialization relationship . . . 154
9.2.3 Abstract classes . . . 156
9.2.4 Type hierarchies in the POST application . . . 156
9.3 More about Associations . . . 159
9.3.1 Associative types . . . 159
9.3.2 Qualified associations . . . 162
9.4 Packages: Organizing Elements . . . 163
9.4.1 POST conceptual model packages . . . 165
9.5 Modelling Behaviour in State Diagrams . . . 167

10 Summing Up 177

Chapter 1

Introduction

Topics for Chapter 1

- Reading List.
- Introduction to Software Engineering

1.1 Reading List

Recommended Reading

1. I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
2. R. Pooley and P. Stevens, *Using UML: Software Engineering with Objects and Components*, Addison-Wesley, 1999.

Background Reading

1. G. Booch, J. Runbaugh and I. Jacobson, *The Unified Modelling Language User Guide*, Addison-Wesley, 1999.
2. J. Runbaugh, I. Jacobson and G. Booch, *The Unified Modelling Language Reference Manual*, Addison-Wesley, 1999.
3. M. Fowler, *UML Distilled – Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.

4. B. Meyer, *Object-oriented Software Construction (2nd Edition)*, Prentice Hall PTR, 1997.

5. O. Nierstrasz and D. Tsichritzis, *Object-Oriented Software Composition*, Prentice Hall, 1995.

6. R. Pressman, *Software Engineering – A Practitioner's Approach (4th Edition)*, McGraw Hill, 1997.

7. M. Priestley, *Practical Object-Oriented Design*, McGraw-Hill, 1997.

8. J. Runbaugh, I. Jacobson and G. Booch,*The Unified Modelling Language Reference Manual*, Addison-Wesley, 1999.

9. S.R. Schach, *Classical and Object-Oriented Software Engineering (3rd Edition)*, IRWIN, 1996.

10. I. Sommerville, *Software Engineering (5th Edition)*, Addison-Wesley, 1995.

1.2 Introduction to Software Engineering

The notion of *software engineering* was proposed in the late 1960s at a conference[1] held to discuss what was then called the 'software crisis'. This software crisis resulted directly from the introduction of third-generation computer hardware. Compared with software systems developed for the second-generation computers, software systems now often have among the others the following features:

- they model parts of the *real world*;
- they are large and complex;
- they are abstract;
- they must be highly dependable;
- they need to be better maintainable: as the real world changes, so too must the software to meet the changing needs and requirements;
- they must be user friendly, and thus the *user interface* in a software system is an important part.

Software development was in crisis because the methods (if there were any) used were not good enough:

- techniques applicable to small systems could not be scaled up;
- major projects were sometimes years late, they cost much more than originally predicted;
- software developed were unreliable, performed poorly, and were difficult to maintain.

[1]NATO Software Engineering Conference, Garmisch, Germany, 1968.

The fact that hardware costs were tumbling while software costs were rising rapidly, as well as the above required features, called for new theories, techniques, methods and tools to control the *development process* of software systems. *Software engineering is concerned with the theories, methods and tools which are needed to develop software.* Its aim is the production of dependable software, delivered on time and within budget, that meets the user's (the application's) *requirement.*

A Software engineering is **not** *to produce a working software system* **only**, but also documents such as *system design*, *user manual*, and so on.

Software products

The objective of software engineering is to produce software products with *high-quality. Software products* are software systems delivered to a customer with the documentation which describes how to install and use the system.

The quality of a software product is judged by, apart from the services provided by the product, the characteristics displayed by the product once it is installed and put into use. The critical characteristics of a software product include

- *Usability*: It must be useful and usable to make people's lives easier and better. For this, the software should have an appropriate user interface and adequate documentation.
- *Maintainability*: It should be possible to evolve software to meet the changing needs of customers. In other words the product should be *flexible* to changes that need to make.
- *Dependability*: Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure.
- *Efficiency*: Software should not make wasteful use of system resources such as memory and processor cycles.

It should be noted that the relative importance of these characteristics obviously varies from system to system. and that optimizing all these characteristics is difficult as some are exclusive. Also the relationship between the cost and improvements in each characteristic is not a linear one.

To build good systems, we need

- a well defined *development process* with clear *phases* of *activities*, each of which has an *end-product*,
- *methods and techniques* for conducting the phases of activities and for modelling their products,
- *tools* for generating the products.

In this course, we shall concentrate on methods, strategies and tools in the object-oriented framework which lead to dependable, maintainable, and user friendly software.

Returning to the issue of maintainability, a software product as a model of the real world has to be maintained constantly in order to to remain an accurate reflection of changes in the real world. Therefore, software engineering must be also concerned with evolving software systems to meet changing needs and requirements.

For instance, if the sales tax rate in the USA changes from 6% to 7%, almost every software product that deals with buying or selling has to be changed. Suppose the product contain the C or C++ statement

```
const float sales_tax = 6.0;
```

declaring that sales_tax is a floating-point constant, and initialized to 6.0. In this case maintenance is relatively simple. With the aid of a text editor the value 6.0 is replaced by 7.0, and the code is recompiled. However, if instead of using the constant declaration, the actual 6.0 has been used in the product wherever the value of the sale tax is invoked, then such a product will be extremely difficult to maintain.

1.3 Questions

1. Consider a piece of software that you enjoy/hate using. In what respects is it a high/low quality system?
2. Do you know some infamous examples of failures of the desirable attributes of software that have dramatic effects? Which of the attributes failed in your example?

Chapter 2

Software Development Process

Topics for Chapter 2

- The main activities in software development
- The waterfall model of software development process
- The Evolutionary development

2.1 Software Development Process

All engineering is about how to produce products in a disciplined process. In general, a **process** defines *who* is doing *what when* and *how* to reach a certain goal. A process to build a software product or to enhance an existing one is called a **software development process**.

A software development process is thus often described in terms of a set of activities needed to transform a user's *requirements* into a software system. At the highest level of abstraction, a development process can be depicted in Figure 2.1.

The client's requirements define the goal of the software development. They are prepared by the client (sometime with the help from a software engineer) to set out the services that the system is expected to provide, i.e. *functional requirements*. The functional requirements should state *what* the system should do rather than *how it is done*. Apart from functional requirements, a client may also have non-functional constraints that s/he would like to place on the system, such as the required response time or the use of a specific language standard. This course is mainly concerned with the functional requirements.

We must bear in mind about the following facts which make the requirement capture and analysis very difficult:

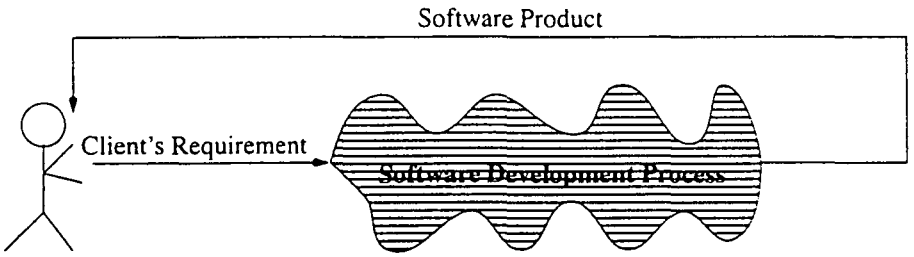


Figure 2.1: A simplified view of the software development process

- The requirements are often incomplete.
- The client's requirements are usually described in terms of concepts, objects and terminology that may not be directly understandable to software engineers.
- The client's requirements are usually unstructured and they are not rigorous, with repetitions, redundancy, vagueness, and inconsistency.
- The requirements may not be feasible.

Therefore, any development process must start with the activities of capturing and analyzing the client's requirements. These activities and the associated results form the first *phase* (or sub-process) of the process called *requirement analysis*.

2.1.1 Requirement capture and analysis

The purpose of the requirement capture analysis is to aim the development toward the right system. Its goal is to produce a document called *requirement specification*. The whole scope of requirement capture and analysis forms the so-called *requirement engineering*. In this chapter, we mainly discuss the main activities needed and the essential attributes of their products, and later we focus on the study of OO techniques for requirement capture and analysis in chapters 4-6.

The document of the requirement specification will be used as

1. the agreed contract between the client and the system development organization on what the system should do (and should not do);
2. the basis used by the development team to develop the system;
3. a fairly full model of what is required of the system.

To fulfill these purposes, the requirement analysis process, depicted in Figure 2.2, should include the following *highly iterative* activities:

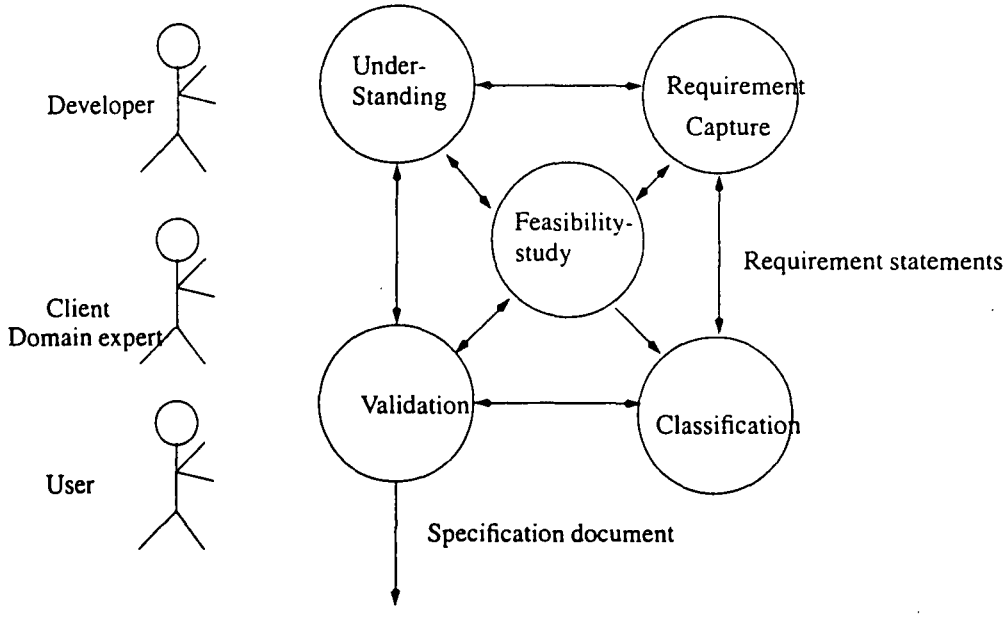


Figure 2.2: A view of the requirement analysis process

- *Domain understanding* Analysts must develop their understanding of the application domain. Therefore, the concept is explored and the client's requirements are elicited.
- *Requirements capture* The analyst must have a way of capturing the client's needs so that they can be clearly communicated to everyone involved in the project. They must interact with the client, and the application domain experts, and the potential system users to discover and capture their requirements. The skills of *abstraction* is important to capture the essences and ignore the non-essences.
- *Classification* This activity takes the unstructured collection of requirements captured in the earlier phase and organizes them into coherent clusters, and then prioritizes the requirements according to their importance to the clients and the users.
- *Validation* This is to check if the requirements are consistent and complete, and to resolve conflicts between requirements.
- *Feasibility study* This is to estimate whether the identified requirements may be satisfied using the software and hardware technologies, and to decide if the proposed system will be cost-effective.

There is no hard guideline rule on when requirement analysis is completed and the development process proceeds into the next phase. The following questions must be asked before the development progresses into the next phase:

- Has the system required been fully understood by the client, end-users, and the developers?

- Has a fairly complete model of what is required built? This is a model of the system about what must be done by the system in terms of
  - what functions (or services) are available;
  - what the input & output are;
  - what data are necessary;

  but no implementation decisions should be described in this model.

Of course, the requirement specification and the model of the system at this level must be adjusted and improved once this is found necessary during a later stage of the development.

An important *side effect of requirement analysis* is *testing the final system*:

- test plan must be made according to the requirement specification;
- test cases should be designed to cover all the crucial services required.

In summary, the requirement specification is the official statement of what is required of the system developer. It is not a design document and it must state *what to be done* rather than *how it is done.* It must be in a form which can be taken as the starting point for the software development. A specification language is often used. Graphical notations are often also used to describe the requirement specification.

The requirement capture and analysis phase is important, as an error which is not discovered at the requirement analysis phase is likely to remain undiscovered, and the later it is discovered, the more difficult and more expensive is it to fix.

2.1.2 System design

After the specification is produced through requirement analysis, the requirement specification undergoes two consecutive *design processes.* The first comes *architectural* (or *logical*) *design* in which the requirements are partitioned into components. This results in an architectural design document which describes *what each component must do* and *how* they interact with each other to provide the over all required services. Then each component in turn is designed; this process is termed *detailed* (or *physical design.* The *detailed design document* describes *how* each component does what it is required to do and thus *how* the whole system does what it is required to do. The activities of the design process and the associated results are depicted in Figure 2.3

Therefore, the document produced in the architectural design phase is an *architectural model* consisting of the specifications of the components which describe *what* each component must do by specifying the *interfaces* of the components. The model of the system at this level is still abstract, implementation independent, and still much about "what" rather than "how".

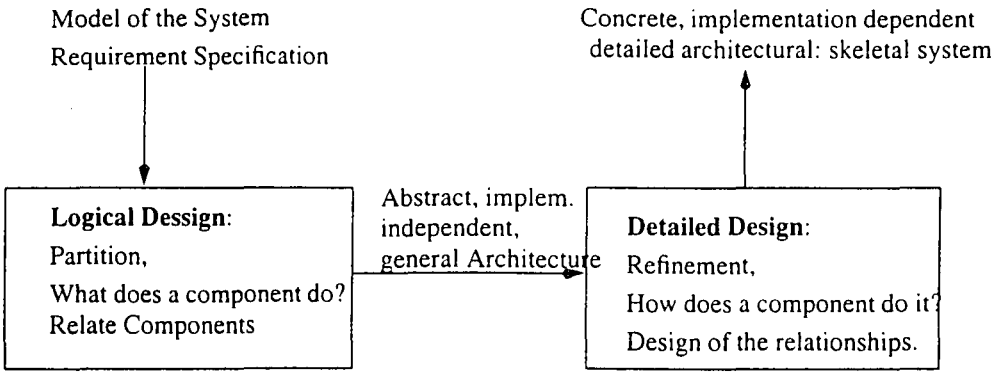


Figure 2.3: A view of the design process

The detailed design sub-process involves a number of steps of refinement to the architectural model, resulting in a detailed design model which describes

- the design of the functions of each component, describing *how* each component provides its required functions,
- the design of the interface of each component, describing "how" each component provides its services to other components.

The model of the system at this level can be seen as a skeletal system which is concrete, implementation dependent, and defines "how".

2.1.3 Implementation and unit testing

During this stage, each of the components from the design is realized as a program unit. Each unit then must be either *verified* or *tested* against its specification obtained in the design stage. This process is depicted in Figure 2.4

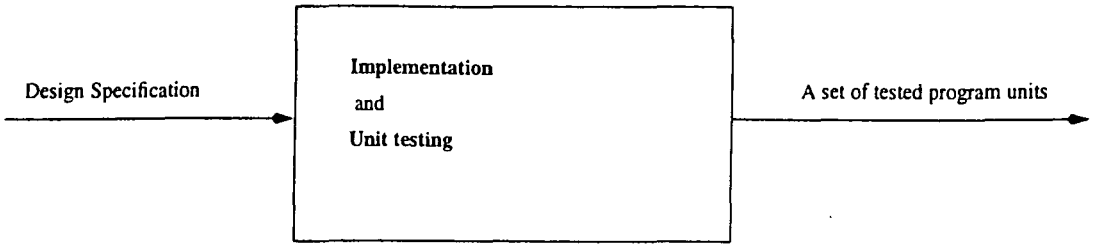


Figure 2.4: A view of the implementation and unit testing process

2.1.4 Integration and system testing

The individual program units representing the components of the system are combined and tested as a whole to ensure that the software requirements have been met. When the developers are satisfied with the product, it is then tested by the client (*acceptance testing*). This phase ends when the product is accepted by the client.

2.1.5 Operation and maintenance

This phase starts with the system being installed for practical use, after the product is delivered to the client. It lasts till the beginning of system's *retirement phase*, which we are not concerned in this course.

Maintenance includes all changes to the product once the client has agreed that it satisfied the specification document. Maintenance includes *corrective Maintenance* (or software repair) as well as *enhancement* (or software update). Corrective Maintenance involves correcting errors which were not discovered in earlier stages of the development process while leaving the specification unchanged. There are, in turn, two types of enhancement:

- *Perfective maintenance* involves changes that the client thinks will improve the effectiveness of the product, such as additional functionality or decreased response time.
- *Adaptive maintenance* are about changes made in response to changes in the environment in which the product operates, such as new government regulations.

Studies has indicated that, on average, maintainers spend approximately 17.5% of their time on corrective maintenance, 60% on perfective maintenance, and 18% on adaptive maintenance.

2.2 The Waterfall Model

According to the software development activities discussed above, the whole development process is often described by the so-called 'waterfall model' depicted in Figure 2.5

The development process of a software product is also called the *life cycle* of the software. We must note that

- In practice, the stages in the waterfall model overlap and feed information to each other: during design, problems with requirements are identified; during coding, design problems are found and so on. Therefore, the development process is not a simple linear model but involves a sequence of iterations of the development activities.

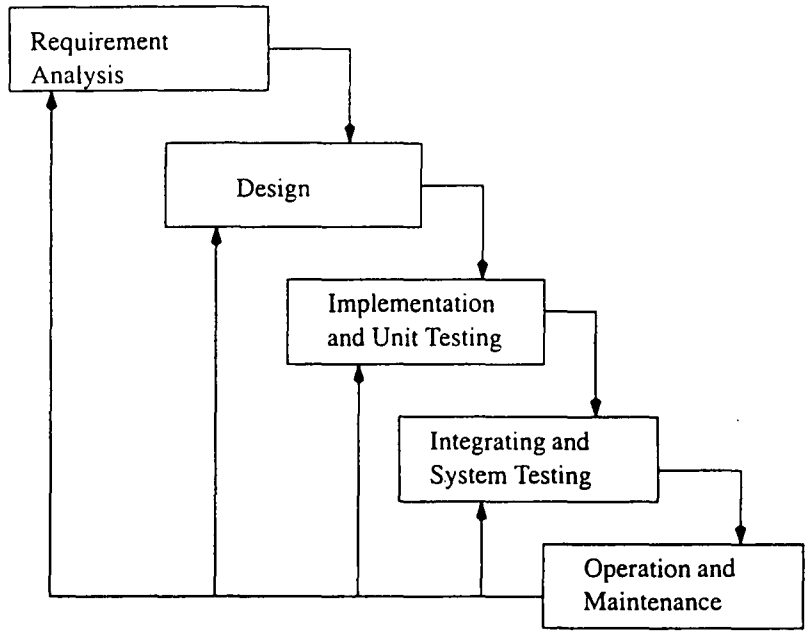


Figure 2.5: The Waterfall Model

- During the final life cycle phase, perfective maintenance activities may involve repeating some or all previous process stages.
- A development process which includes frequent iterations makes it difficult to identify management checkpoints for planning and reporting. Therefore, after a small number of iterations, it is normal to freeze parts of the development such as the specification, and to continue to the later development stages. Problems are left for later resolution, ignored or are programmed around.
- Sometimes it is quite difficult to partition the development activities in a project into these distinct stages.

The waterfall model can be used for some statistic studies about the software development. Figure 2.6 and Figure 2.7 illustrate why high quality software is difficult to produce and to maintain, and why the development activities must be conducted in a good engineering manner.

2.3 Evolutionary Development

A problem with the waterfall model is that some software development project is difficult to be partitioned into the distinct stages of requirement analysis, design and so on. Sometimes, it is also difficult (or impossible) to establish a detailed requirement specification.

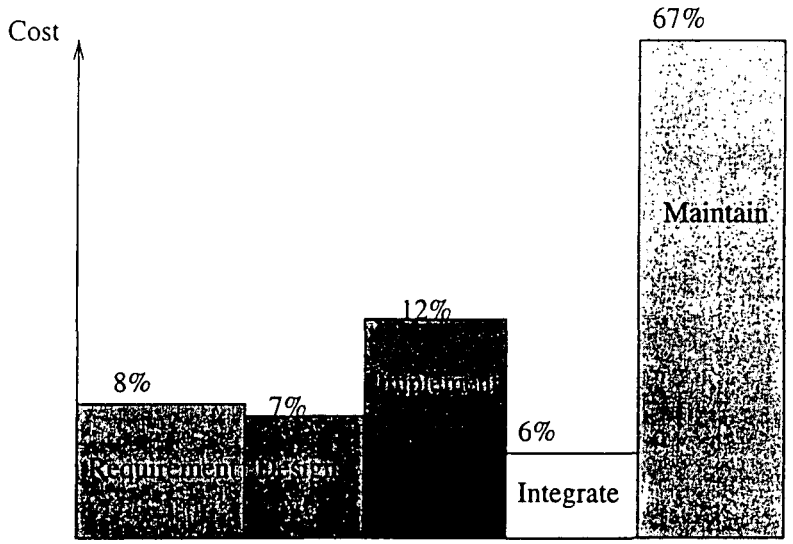


Figure 2.6: Approximate relative costs of software process phases (from Schach 1990)

Evolutionary development is based on the idea of developing an initial implementation, exposing this to user comment and refine through many versions until an adequate system has been developed (Figure 2.8).

The development process starts with an outline description of the system. Rather than having separate specification, development (design, implementation) and validation (testing and/or verification and/or prototyping) activities, these are carried out concurrently with rapid feedback across these activities.

The techniques used in an evolutionary development include

- *Exploratory programming* where the objective of the process is to work with the client to explore their requirements and deliver a final system. The development starts with the parts of the system which are understood. The system evolves by adding new features as they are proposed by the client.
- *Prototyping* where the objective of the development is to understand the customer's requirements and hence develop a better requirements definition for the system. The prototype concentrates on experimenting with those parts of the client requirements which are poorly understood.

Obviously, this model with the iterations of try-see-change activities suffers from the following problems

- *The process is not visible* It is difficult and expensive to produce documents which reflect every version of the system.
- *System are poorly structured* Continual change tends to corrupt the software structure.

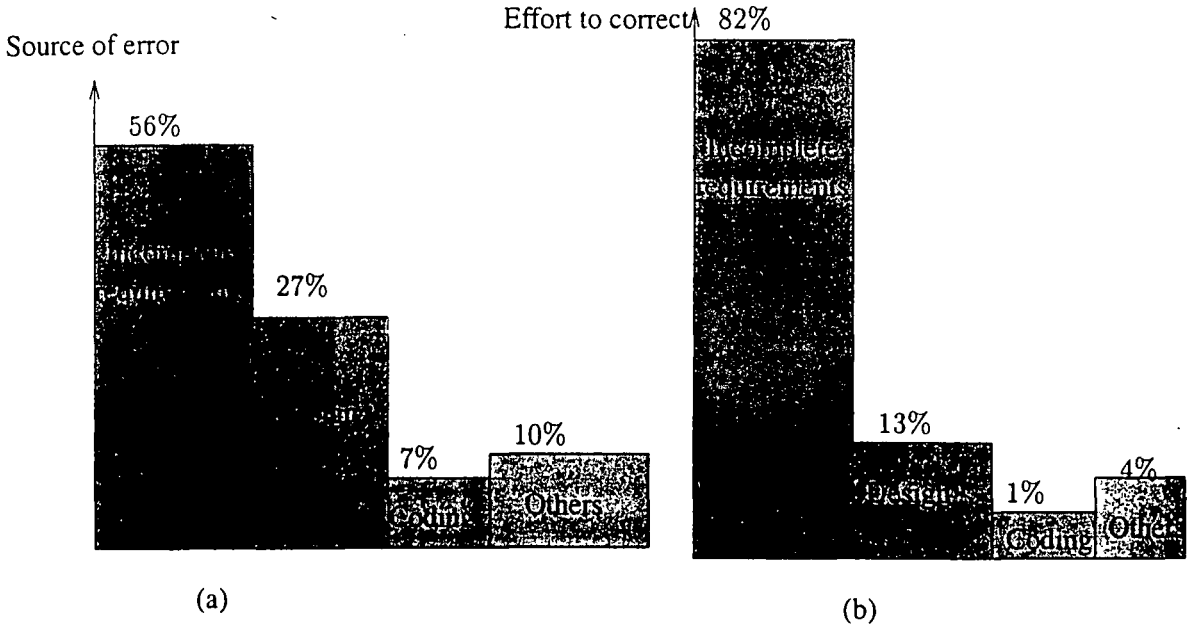


Figure 2.7: Software errors (a) their source and (b) their correction effort (from Finkelstein, 1989)

- *It is not always feasible* For large systems, changes in later versions are very much difficult and sometimes impossible. New understanding and new requirements sometimes force the developer to start the whole project all over again. Software evolution is therefore likely to be difficult and costly. Frequent prototyping is also very expensive.

These problem directly lead to the problems that the system is difficult to understand and maintain. Therefore it is suggested that this model should be used in the following circumstances:

- The development of relatively small systems.
- The development of systems with a short lifetime. Here, the system is developed to support some activity with is bounded in time, and hence the maintenance problem is not an important issue.
- The development of systems or parts of large systems where it is impossible to express the detailed specification in advance (e.g. AI systems and Interfaces).

The ideas, principles and techniques of the evolutionary development process are always useful and should be used in different stages of a wider development process, such as the requirements understanding and validating in the waterfall process.

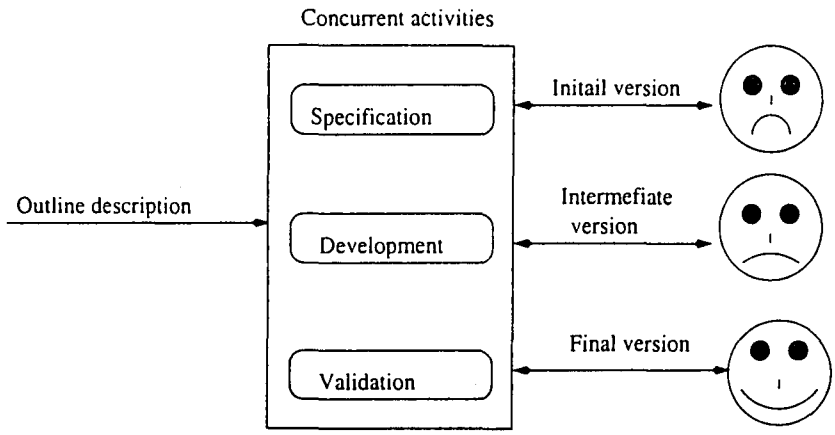


Figure 2.8: Evolutionary development model

2.4 Questions

1. Why do we need a software development process.
2. Find more about software development process, especially the requirements that a good software development process must be met.

2.5 Further Reading

1. I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
2. R. Pooley and P. Stevens, *Using UML: Software Engineering with Objects and Components*, Addison-Wesley, 1999.
3. R. Pressman, *Software Engineering – A Practitioner's Approach (4th Edition)*, McGraw Hill, 1997.
4. I. Sommerville, *Software Engineering (5th Edition)*, Addison-Wesley, 1995.

Chapter 3

Introduction to OO Development

Topics for Chapter 3

- The inherent complexity of software
- Mastering complex systems
- The hierarchy of complex systems
- Motivation to OO
- The outline of the rest of the course

3.1 The Inherent Complexity of Software

Within the scope of software engineering, the models for software development process and their associated principles and techniques have provided much better understanding of the activities in the process and have led to great improvements in productivity and quality of software. On the other hand, we are still in the software crisis. Now we understand that this is because of the inherent complexity of software.

The following three observations show why the *complexity of software is an essential property.*

The complexity of the problem domain This, in turn, derives from the following elements:

1. *Difficulties in capturing requirements* This usually comes from the "impedance mismatch" that exists between the users of the system and its developers:

- Users generally find it very hard to give precise expression to their needs in a form that developers can understand.
- Users may only have vague ideas of what they want in a system.

2. *Competing and contradictory requirements* Consider the requirements for the electronic system of a multi-engine aircraft, or an autonomous robot. The raw functionality of such systems is difficult enough to comprehend, but now add all of the (often implicit) nonfunctional requirements such as usability, performance, cost, and reliability.

3. *Changing requirements* Requirements of a software system often change during its development, largely because

   - Seeing early products, such as design documents and prototypes, and then using a system once it is installed and operational, lead the users to better understand and articulate their real needs.
   - At the same time, the development process itself helps the developers master the problem domain, enabling them to ask better questions that illuminate the dark corners of the system's desired behaviour.

Elements 1&2 imply the problems and difficulties that are involved during the requirement analysis phase, while element 3 implies that a software development is an evolving process and iterations are inescapable, and that evolution and maintenance are inherently difficult.

The difficulty of managing the development process The fundamental task of the software development team is to engineer the illusion of simplicity - to shield users from the vast and often arbitrary complexity (See Figure 3.1) of the system.

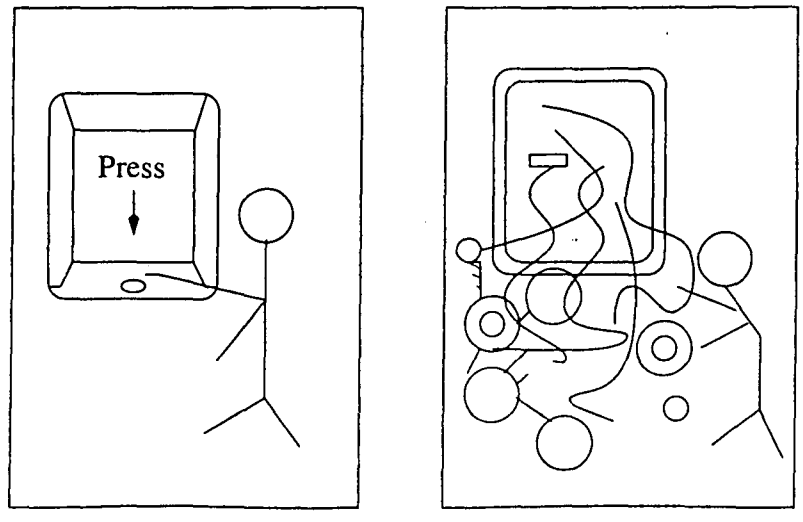


Figure 3.1: To Engineer the illusion of simplicity

However, the sheer volume of a system requirements often requires us to write a large amount of new software or to reuse existing software in novel ways. No one can ever completely understand a system containing hundreds of thousands. or even millions of lines of code. The amount of work demands that we use a team of developers. More developers means more complex communication and hence more difficult coordination.

The problems of characterizing the behaviour of system In a software system for a large application, there may be hundreds or even thousands of variables as well as more than one thread of control. The behaviour of the system is how it changes from one state to another. Although a system can in practice have only finite number of states,

- in a large system, there is a combinatorial explosion that makes this number very large;
- each event external to the system may change the state of the system;
- the system may react to an external event *nondeterministically*, i.e. it may not be predictable which of a number of states the system will enter after a certain event.

This observation indicates the importance and difficulty of the *decomposition* in the design phase.

3.2 Mastering Complex Systems

Having said what make software inherently complex, An essential problem of software engineering is how we can master this complexity during the software development process.

3.2.1 Examples of complex systems

A personal computer A personal computer is a device of moderate complexity. Most of them can be *decomposed* into the *same* major elements:

- a central processing unit (CPU),
- a monitor,
- a keyboard, and
- some sort secondary storage device, either a floppy disk or a hard disk drive.

Each of these parts can be *further decomposed* into subparts. For example, a CPU is composed of

- a primary memory unit,
- an arithmetic/logic unit (ALU), and
- a bus to which peripheral devices are attached.

Each of these parts may be in turn decomposed: an ALU may be divided into

- registers, and random control logic

which themselves are constructed from even more primitive elements, such as

- NAND gates, and inverters,

and so on.

Here we see the *hierarchic* nature of a complex system. A personal computer functions properly only because of the *collaborative activity* of each of its parts. We can reason about how a computer works only because we can decompose it into parts that we can study separately.

The important observations that we can make about a hierarchic system are:

- The layers of this hierarchy represents different levels of abstractions, each built upon the other, and each understandable by itself.
- At each level of abstraction, we find a collection of parts that *collaborate* to provide services to the higher layers.
- The layer we choose to investigate depends on our current needs.

The structure of plants To understand the similarities and differences among plants, a plant can be seen as a complex system which can be decomposed into the hierarchy of subsystems illustrated in Figure 3.2

All parts at the same level of abstraction interact in well-defined ways. For instance, at the highest level of abstraction:

- roots are responsible for absorbing water and minerals from the soil;
- roots interact with stems, which transport these raw materials up to the leaves;

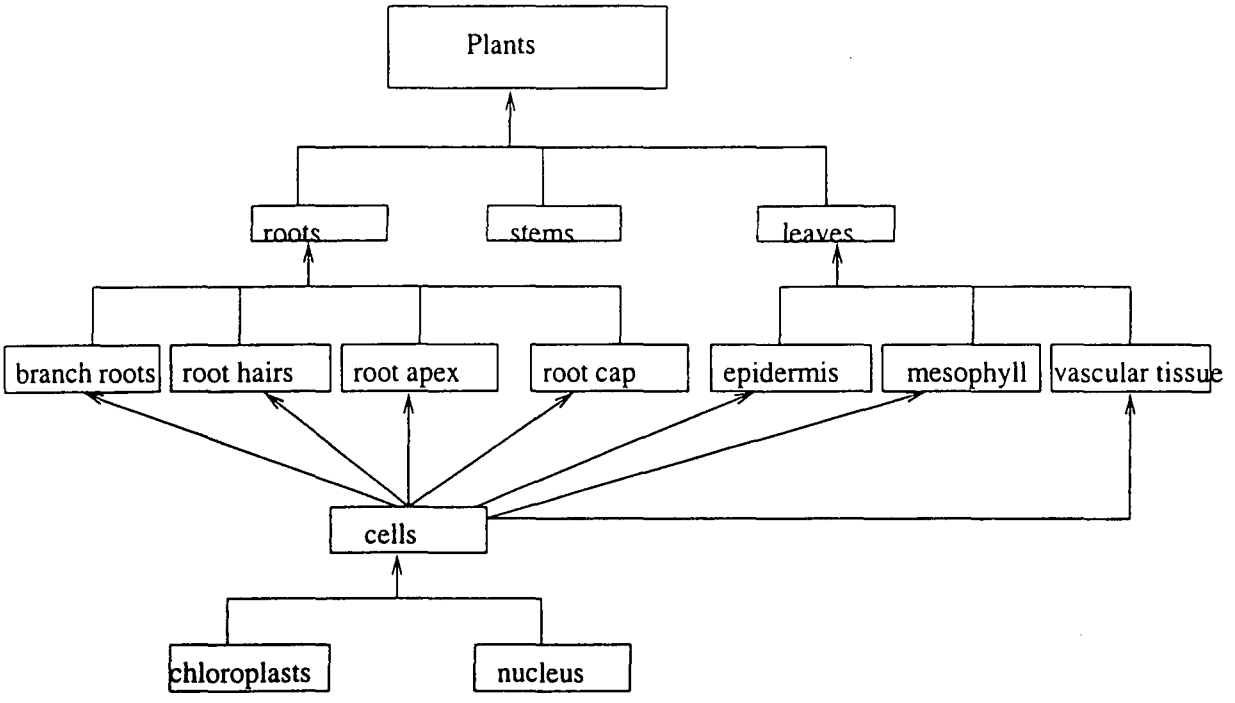


Figure 3.2: The hierarchy of plants

- the leaves in turn use the water and minerals provided by the stems to produce food through photosynthesis.

There are always *clear boundaries* between the outside and inside of a given level. For example, we can imagine that the parts of a leaf work together to provide the functionality of the leaf as whole, and yet have little or no direct interaction with the elementary parts of the roots. In other words, *there is clear separation of concerns among the parts at different level of abstractions.*

Social Institutions As a final example of complex systems, we turn to the structure of social institutions, which are groups of people join together to accomplish tasks that cannot be done by individuals.

The structure of a large organisation is clearly hierarchy. Multinational corporations, for example, contain companies, which in turn are made up of divisions, which in turn contain branches, which in turn encompass local offices, and so on (see Figure 3.3)

The relationship among the various parts of a corporation are just like those found among the components of a computer or a plant. For instance, the degree of interactions among employees within an individual branch is greater than that between employees of different branches.

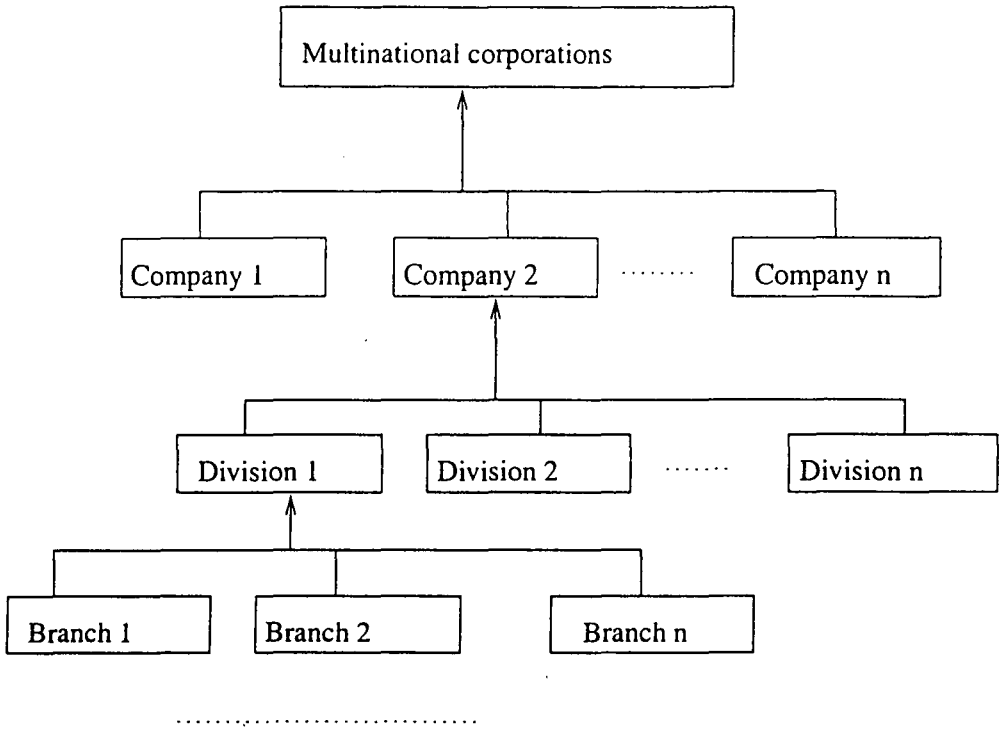


Figure 3.3: The hierarchy of multinational corporations

3.2.2 The five attributes of a complex system

From the examples, we have five attributes common to all complex systems.

The fact that many complex system have a decomposable hierarchic structure is a major facilitating factor enabling us to understand, describe, and even 'see' such systems and their parts. This is summarised as the following attribute.

1. *Complexity takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, an so on, until some lowest level of elementary components is reached*

Regarding to the nature of the *primitive components* of a complex system, we have that:

2. *The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system*

The components in a system are not completely independent. The understanding of the linkages between

the parts of the system is also important. The nature of these linkages is another attribute common to all complex systems:

3. *Intracomponent linkages are generally stronger than intercomponent linkages. This fact has the effect to separating the high-frequency dynamics of the components - involving the internal structure of the components - from the low-frequency dynamics - involving interaction among components*

The following attribute says that complex systems have common patterns in their construction and implementation:

4. *Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements*

In other words, a considerable amount of commonality cuts across all parts in the structural hierarchy of a system. For example, we find, in a computer, NAND gates used in the design of the CPU as well as in the hard disk drive. Likewise, cells serve as the basic building blocks in all structures of a plant.

Complex systems tend to evolve over time. As systems evolve, objects that were once considered complex become the primitive objects upon which more complex systems built. This fact is stated as another attribute of complex systems:

5. *A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system*

Furthermore, we can never craft the primitive objects correctly the first time: we must use them in context first, and then improve them over time as we learn more about the real behaviour of the system.

3.2.3 The generalization-specialization hierarchy

Having that complex system can generally be decomposed into components which are combined and arranged into layers representing different levels of abstraction, such a decomposition represents a *structural* (or "*part-of*") hierarchy. More interesting systems embody another hierarchy. For example, with a few minutes of orientation, an experienced pilot can step into a multi-engine jet s/he never flown before and safely fly the vehicle. In general, if a pilot already knows how to fly a given aircraft, it is far easier to know how to fly a similar one. Therefore, the *concept* of **aircraft** represents an abstraction which

generalizes the properties common to every kind of aircraft. In other words, and particular plane is a *special kind* of aircraft, which has the properties that are common to all aircraft and properties which are special to its kind. We shall see this secondary hierarchy is also essential to understand a complex system. In the OO framework, this "is a (or kind of)" hierarchy is called the *class structure* and the "part-of" hierarchy is called the *object structure*. This is illustrated in Figure 3.4

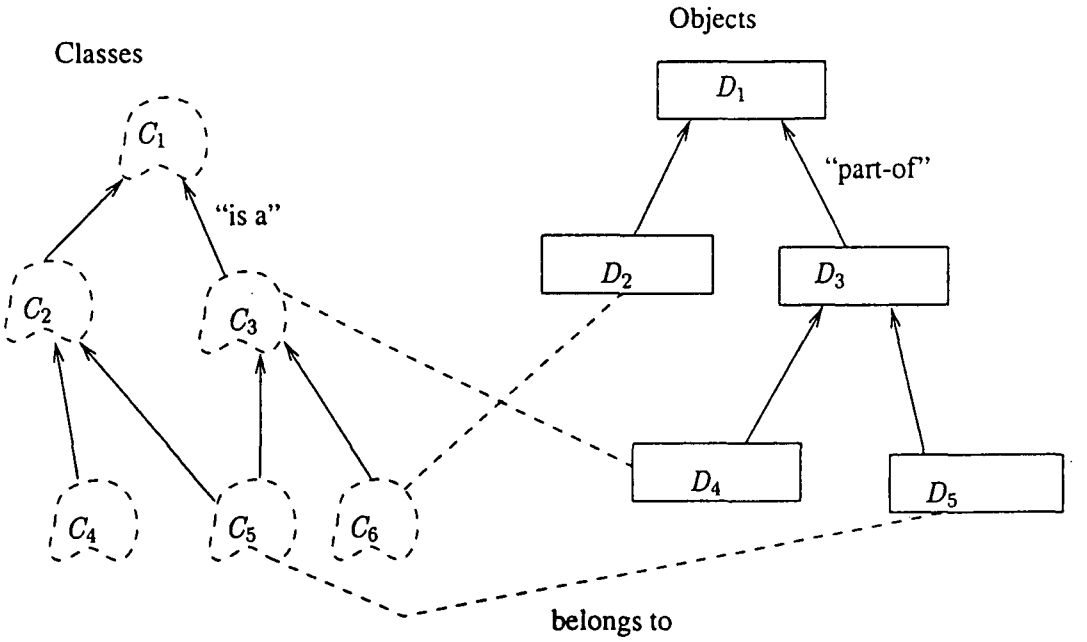


Figure 3.4: A canonical form of a complex system

This figure represents the two hierarchies:

- any instance (object) of class C_6 is a kind of instances of class C_3;
- objects D_2 and D_2 are parts of object D_1;
- object D_2 is an instance of class C_6, and so on

OO software engineering is about *how to engineering the class and object structures which have the five attributes of complex systems.* And in the context of OO software engineering, *the structure of a system means the class and object structure of the system.*

3.2.4 Function-oriented and object-oriented methods

Having said decomposition of a system into components is essential in mastering complex systems, what is the role of decomposition in the development of a software system?

Function oriented methods Until the middle of the 1990s, most of software engineers are used to the *top-down functional* design methods (or *structured design*), whose defining aspects include

- It is strongly influenced by the programming languages such as ALGOL, Pascal and C, all of which offer *routines* as their highest-level abstractions.
- The functions of a software system are considered as the primary criterion for its decomposition.
- It separates the functions and data, where functions, in principle, are active and have behaviour, and data is a passive holder of information, which is affected by functions.
- The top-down decomposition typically breaks the system down into functions, whereas data is sent between those functions. The functions are broken down further and eventually converted into source code (see Figure 3.5).

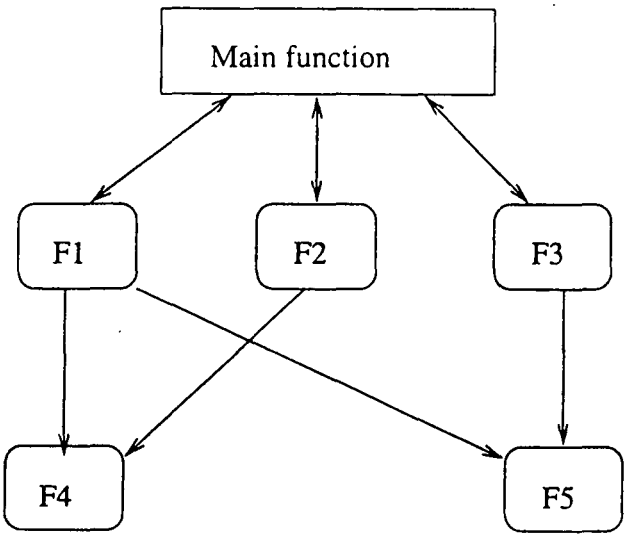


Figure 3.5: Functional, top-down decomposition

People have found the following problems with the functional technology:

- Products developed with this approach are difficult to maintain. This is mainly because that all functions share a large amount of data, and they must know how the data are stored. To change a data structure, we must modify all the functions relating to the structure.
- The development process is not stable as changes in the requirements will be mainly reflected in its functionality. Therefore, it is difficult to retain the original design stricture when the system evolves.
- The development process gets into the "how" business too soon, as when we decompose a function into subfunctions, we cannot avoid from talking about first do this and then do that, and so on.

- This approach only captures the "part-of" abstraction.
- Obviously, this approach does not support programming in object-oriented languages, such as Smalltalk, Eiffel, C++, and Java.

Object-oriented technology The strategy in the OO software development is to view the world as a set of objects. They interact with and collaborate with each other to provide some higher level behaviour. Objects have the following characteristics (see Figure 3.4):

- An object is simply a tangible entity in the real world (at the requirement analysis phase) or represents a system entity (at the design stage).
- Objects are responsible for managing their own private state, and offering services to other objects when is it requested. Thus, data and functions are encapsulated within an object.
- The system's functionality is observed by looking at how services are requested and provided among the objects, without the need to get into the internal state changes of the objects.
- Objects are classified into *classes*, and objects belonging to the same class have common properties, such as *attributes* and *operations*.

It seems that the OO approach reflects the "is a" abstraction as well as the "part of" abstraction better, and can overcome some of the problems in the function approach (notice that I do not want to make stronger claims).

What actually is OO Technology?

There is currently an war in arguing about what OO is and how it differs from the classical structured design methods. Here I quote

> ..... Unfortunately, object-oriented programming means different things to different people. As Rentsch correctly predicted, "My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Every one will be in favour of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is". Rentsch's predications still apply to the 1990s.
>
> – Grady Booch 1994

I do not think we should join this war. We rather like to learn the concepts, techniques, and notations that have developed and found useful in the OO framework. One thing will become clear to us, OO design helps in developing good software written in C++ and Java.

3.3 The Rest of The course

The rest of course is to discuss the *object-oriented software* methodology. The methodology includes the following

- *Notation* The language for expression each model of the system at a level of abstraction. For this purpose, we shall use UML, standing for **Unified Modelling Language** which is widely used as a standard modelling language.
- *Process* The activities leading to the orderly construction of the system's models. We shall focus on the activities of *object-oriented analysis* (OOA), *object-oriented design* (OOD), and *object-oriented implementation strategies* (OOI).
- *Tools* The artifacts that eliminate the tedium of the model building and enforce rules about the models themselves, so that errors and inconsistencies can be exposed. The tool that we use for the practicals will be the *Rational Rose*[1]

3.4 Questions

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

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

[1] For information about National Rose, a thorough treatment of raw UML notation, the complete specification is available at Ration Corporation's web site: www.rational.com

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

   Characterize the East-West library system described above in terms of the five attributes of complex systems.

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

4. Figure 3.6 depicts a possible object structure for a simple blackjack game, listing some of the important elements connected in a "part-of" hierarchy. In analyzing complex systems, understanding the *relationship(s)* between hierarchical components is just as important as identifying the components themselves. For each connection among notes in the blackjack graph, specify the relationship(s) between the connected objects.

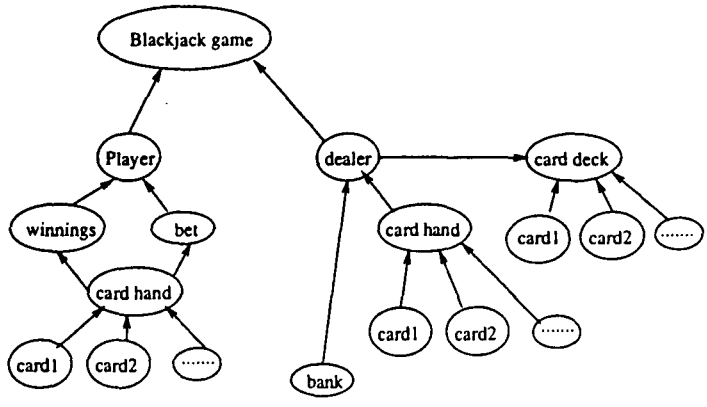


Figure 3.6: Object structure for a Blackjack Game

5. Describe one or more extensions to the blackjack game depicted above that would have little impact on the complexity of the system. Then describe an extension that would make the system noticeably more complex.

Chapter 4

Requirement Capture and Analysis – Use Cases

Topics of Chapter 4

- Understanding requirements specification, such as functional specification, and nonfunctional attributes of systems
- Understanding use cases, use case diagrams, and use-case model for describing the functional requirements
- Use-case diagrams and their use for describing a use-case model

4.1 Understanding requirements

4.1.1 Introduction

The creation of correct and thorough requirements specification is essential to a successful project. However, the treatment of the whole host skills necessary to elucidate meaningful requirements is simply too much to cover in this course. Instead of going into every aspect of every skill, we use a case study to illustrate the following three problems.

- What should be produced in the requirements capture and analysis?
- How to identify the elements of these artifacts?
- How are artifacts expressed?

4.1.2 Case study: Point-of-sale

A point-of-sale terminal (POST) is a computerized system used to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and a bar code scanner, and software to run the system (See Figure 4.1).

Assume that we have been requested to create the software to run a point-of-sale terminal. Using an object-oriented development strategy, we are going to proceed through the requirement analysis, design, and implementation.

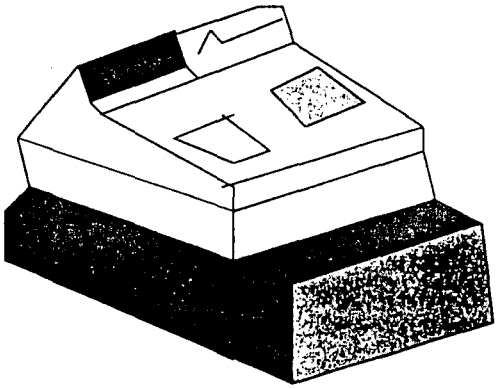


Figure 4.1: A point-of-sale terminal

4.1.3 Requirements specification

The requirements specification is a description of needs or desires for a product.

The requirements must be described

- unambiguously, and
- in a form that clearly communicates to the client and to the development team members.

It is recommended that the requirements specification at least include the following parts:

- an overview of the project
- goals
- system functions
- system attributes (non-functional requirements)

- Glossary – definition all relevant terms
- Use cases – narrative descriptions of the domain processes
- Conceptual model – a model of important concepts and their relationships in the application domain.

These are typically produced through gathering and digesting

- varied paper such as the client's statements about their needs, preliminary investigation report, and electronic documents,
- interview results,
- requirements definition meetings, and so on

This section only discusses the first four parts of a requirements specification, and the others are left for subsequent sections.

Overview of the project

This should describe the need for the system. For a large system, it should also briefly describe the system's functions and explain how it will work with other systems.

The overview of the POST case study project can be simply written as the following statement:

> The purpose of this project is to create a point-of-sale terminal system to be used in retail sales.

Goals

This is to describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.

The goals of the POST system can be stated as

> In general the goal is increased checkout automation, to support faster, better and cheaper services and business processes. More specifically, these include:
>
> - quick checkout for the customer,

- fast and accurate sales analysis,
- automatic inventory control.

The overview and the goals can be combined into an *introduction* section in the document.

System functions

System functions are *what* a system is supposed to *do*.

> To verify that some X is indeed a system function, it should make sense in the following sentence:
>
> The system should do < X >

For example, "ease-of-use" does not fit in the verification sentence. Thus, system qualities of this kind should not be part of the functional requirements specification.

The system's function should be categorized in order to priorities them or to avoid from missing them. Categories include

- *Evident functions* should be performed, and user should be cognizant that is performed.
- *Hidden functions* should be performed, but not visible to users. This is true of many underlying technical services, such as *save information in a persistent storage mechanism*. Hidden functions are often incorrectly missed during the requirements gathering process.
- *Frill functions* are optional; adding them does not significantly affect cost or other functions.

In the presentation of the functions,

- they should be divided into logical cohesive groups,
- each function should be given a reference number that can be used in other documents of the development,
- the category that a function belongs to should be indicated.

For the POST application, we present two groups of the system functions, *the basic functions* and the functions related to *payment*. These functions serve as a set of representative sample, rather than a complete list.

Basic functions of the POST system

Ref #	Function	Category
R1.1	Record the underway (current) sale - the items purchased.	evident
R1.2	Calculate current sale total.	evident
R1.3	Capture purchase item information from a bar code using a bar code scanner, or manual entry of a product code, such as a universal product code (UPC).	evident
R1.4	Reduce inventory quantities when a sale is committed.	hidden
R1.5	Log completed sales.	hidden
R1.6	Cashier must log in with an ID and password in order to use the system.	evident
R1.7	Provide a persistent storage mechanism.	hidden
R1.8	Provide inter-process and inter-system communication mechanisms.	hidden
R1.9	Display description and price of item recorded.	evident

Payment functions

Ref #	Function	Category
R2.1	Handle cash payments, capturing amount tendered and calculating balance due.	evident
R2.2	Handle credit payments, capturing credit information from a card reader or by manual entry, and authorizing payment with the store's (external) credit authorization service via a modem connection.	evident
R2.3	Handle cheque payments, capturing drivers license by manual entry, and authorizing payment with the store's (external) cheque authorization service via a modem connection.	evident
R2.4	Log credit payments to the accounts receivable system, since the credit authorization services owes the store the payment amount.	hidden

System attributes

System attributes are also called *non-functional requirements*. They are constraints imposed on the system and restrictions on the freedom of the designer. These may include

- response time and memory requirements,
- ease of use,
- operating systems platforms,
- interface metaphor,
- retail cost,
- fault-tolerance, and so on

Some attributes may cut across all functions, such as the operating system platform, or be related to a particular function or a group of functions.

Here are examples of attributes for the POST system:

Attribute	Constraints
response time	When recording a sold item, the description and price will appear within 5 seconds
interface metaphor	Forms-metaphor windows and dialog boxes
fault-tolerance	Must log authorized credit payments to accounts receivable within 24 hours, *even if power or device fails*
operating system platform	Microsoft Window 95 and NT

4.2 Use Cases: Describing Processes

One of the major aims in OOA is to decompose the requirements into concepts and objects in the application domain and produce a conceptual model. An important technique to help in this decomposition is to consider *use cases* – narrative description of domain processes in terms of interactions between the system and its users. This section introduces use case concepts.

4.2.1 Use case concepts

Informally speaking, a use case is a story or a case of using a system by some users to carry out a process. A bit more precisely speaking, a *use case* describes the sequence of *events* of some types of users, called *actors*, using some part of the system functionality to complete a process.

For example, to carry out the process of buying things at a store when a POST is used

- two actors must be involved: Customer and Cashier,
- the following sequence of events must be performed:

  The Customer arrives at a checkout with items to purchase. The Cashier records the purchase items and collect payment. On completion, the Customer leaves with the items

Therefore, to a user, a use case is a way of using the system. A use case is described in terms of a sequence of interactions between some actors and the system by which the system provide a service to the actors. Each use case then captures a piece of functional requirements for some users. All the use cases together describe the overall functional requirements of the system. The first step in *requirement capture* is to *capture requirements as use cases*. All the use cases allow software developers and the client/customer to agree on the requirements, that is, the conditions or capabilities to which the system must conform.

Actors

An *actor* represents a coherent set of roles that are entities *external* to the system can play in using the system, rather than representing a particular individual. An actor represents a type of *users* of the system or external systems that the system interacts with.

Note that

- An actor typically stimulates the system with input events, or receives something from the system.
- Actors communicate with the system by sending *messages* to and receiving messages from the system as it performs use cases.
- Actors model anything that needs to interact with the system to exchange information: human users, computer systems, electrical or mechanical devices such as timers.
- A physical user may act as one or several actors as they interact with the system; and several individual users may act as different occurrences of one and the same actor.
- If there are more than one actor in a use case, the one who generates the starting stimulus is called the *initiator actor* and the other *participating actors.*
- The actors that directly interacts the system are *primary/direct actors*, the others are called secondary actors.

Example: An ATM system (an auto bank machine system) interacts with a type of users who will use the system to withdraw money from accounts, to deposit money to accounts, and to transfer money between accounts. This type of users is then represented by The Bank Customer actor. Therefore a *use-case model* can be used to represent the three use cases **WithDraw Money, Deposit Money**, and **Transfer Money**, that have association to the Bank Customer actor.

Thus, actors represent parties outside the system that collaborate with the system. Once we have identified all the actors of a system, we have identified the *external environment* of the system.

4.2.2 Identifying use cases

Each of the following steps for identifying use cases involves brainstorming and reviewing existing documents of requirements specification:

1. One method to identify use case is *actor-based*:
   (a) Identify the actors related to a system or organization, i.e. find and specify all the actors by looking at which users will use the system and which other systems must interact with it.

(b) For each actor, identifying the processes they initiate or participate in by looking at how the actor communicate/interact with (or use) the system to do his work.

2. A second method to identify use cases is *event-based.*

   (a) Identify the external events that a system must respond to.

   (b) Relate the events to actors and use cases.

To identify use cases, read the existing requirements from an actor's perspective and carry on discussions with those who will act as actors. It will help to ask and answer a number of questions, such as

- What are the main tasks of the actor?
- Will the actor have to read/write/change any of the system information?
- Will the actor have to inform the system about outside changes?
- Does the actor wish to be informed about changes?

For the POST application the following potential use actors and use cases can be identified:

Actor	Processes to Initiate
Cashier	Log In Log Out
Customer	Buy Items Refund Items

4.2.3 Writing use cases

When we use the methods given in the above subsection to identify a use case, we first create a high-level use case to obtain some understanding of the overall process, and then expand it by adding to it with more details.

A *high-level use case* describe a process very briefly, usually in two or three sentences. They are often only concerned with the events that the actors perform. It is described in the following format:

Use case:	**Name of use case** (use a phrase starting with a verb).
Actors:	List of actors (external agents), indicating who initiates the use case.
Purpose:	Intention of the use case.
Overview:	A brief description of the process.
Cross References:	Related use cases and system functions.

For example, the high-level description of the **Buy Items with Cash** process can be described as follows.

Use case:	**Buy Items with Cash**
Actors:	Customer (initiator), Cashier.
Purpose:	Capture a sale and its cash payment.
Overview:	A Customer arrives at a checkout with items to purchase. The Cashier records the purchase items and collects a cash payment. On completion, the Customer leaves with the items.
Cross References:	*Functions:* R1.1, R1.2, R1.3, R1.7, R1.9, R2.1.

The references to the system functions indicate that

- the use case is created through further understanding of these functions,
- these required functions are allocated to this use case,

This is useful that

- it is possible to verify that all system functions have been allocated to use cases,
- it provides a link in terms of tractability between the products produced at different stages in the development,
- all system functions and use cases can be traceable through implementation and testing.

An *expanded use case* shows more details than a high-level one, and is often done in a *conversational* style between the actors and the system. Typically, an expanded use case *extends a high-level one with* two sections *typical course of events* and *alternative courses* of events (or *exceptions*):

Use case:	**Name of use case** (use a phrase starting with a verb).
Actors:	List of actors (external agents), indicating who initiates the use case.
Purpose:	Intention of the use case.
Overview:	A brief description of the process.
Cross References:	Related use cases and system functions.

Typical Course of Events

Actor Action	System Response
Numbered actions of the actors.	Numbered descriptions of system responses.

Alternative Courses

- Alternatives that may arise at *line_number*. Description of exception.

For example, the high-level **Buy Items with Cash** use case can be expanded with the two sections in the following way.

Typical Course of Events

Actor Action	System Response

1. This use case begins when a Customer arrives at a POST checkout with items to purchase.

2. The Cashier records the identifier from each item.

   If there is more than one same item, the Cashier can enter the quantity as well.

3. Determines the item price and adds the item information to the running sales transaction.

   The description and price of the current item are presented.

4. On completion of the item entry, the Cashier indicates to the POST that item entry is completed.

5 Calculates and presents the sale total.

6. The Cashier tells the Customer the total.

7. The Customer gives a cash payment, possibly greater than the sale total.

8. The Cashier records the cash received amount.

9. Shows the balance due back to the Customer.
   Generate a receipt.

10. The Cashier deposits the cash received and extracts the balance owing.

    The Cashier gives the balance owing and the printed receipt to the Customer.

11. Logs the completed sale.

12. The Customer leaves with the items purchased.

Alternative Courses

- Line 2: Invalid identifier entered. Indicate error.
- Line 7: Customer didn't have enough cash. Cancel sales transaction.

We must note, the use case **Buy Items with Cash** is a simplified description of the full buy item process. It assumes the following:

- Cash payments only.
- No inventory maintenance.

- It is a stand-alone store, not part of a larger organization.
- Manual entry of UPCs; no bar code reader.
- No tax calculations.
- No coupons.
- Cashier does not have to log in; no access control.
- There is no record maintained of the individual customers and their buying habits.
- There is no control of the cash drawer.
- Name and address of store, and date and time of sale, are not shown on the receipt.
- Cashier ID and POST ID are not shown on the receipt.
- Completed sales are recorded in an historical log.

It is important to remember the 5th attribute of complex and we always begin the development with building a simple system and then enhance and improve it over time as we learn more about the behaviour.

4.2.4 Essential and real use cases

At the requirement level, the use cases are relatively free of technology and implementation details; design decisions are deferred and abstracted, especially those related to the user interface. An use cases of this kind is said *essential* as it describes the process in terms of the essential activities and motivation.

In contrast, a *real use case* concretely describes the process in terms of its real current design, committed to the specific input and output technologies, and so on.

For example, consider an *ATM Withdraw Cash* use case. The **Typical Course of Events** sections in an essential form and in a real form can be used for illustration:

Essential

Actor Action	**System Response**
1. The Customer identifies him/herself	**2.** Present options.
3. and so on	**4.** and so on

Real

Actor Action	System Response
1. The customer inserts his/her card.	2. Prompts for PIN.
3. Enter PIN on keypad.	4. Display options menu.
5. and so on.	6. and son.

4.2.5 Use-case model and use case diagrams

All the actors and use cases of a system make up a *use-case model* which describes how the use cases relate to each other and to the actors, and specifies the system functional requirements.

A *use case diagram* describes part of the use-case model and illustrates a set of use cases for a system, the actors, and the relation between the actors and use cases. The UML notation for a use case diagram is shown in Figure 4.2, in which

- an oval represents a use case,
- a stick figure represents an actor,
- a line between an actor and a use case represents that the actor initiates and/or participates in the process.

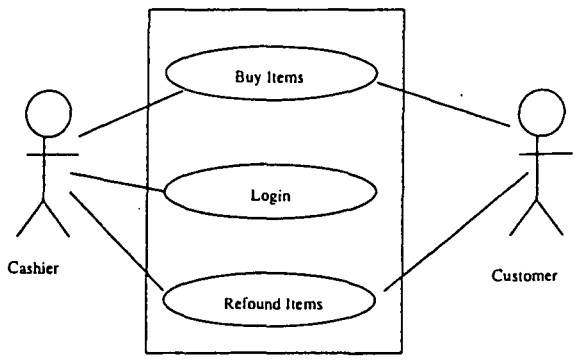


Figure 4.2: A sample use case diagram

Making up a big use case from simpler ones

It is not difficult to see that the sequence of actions from 7 to 10 in use case **Buy Items with Cash** can be treated as a use case, which we can call **Pay by Cash**. It can be refined by adding more details such as:

Pay by Cash

Typical Course of Events

Actor Action	System Response
1. The Customer gives a cash payment, possibly greater than the sale total.	
2. The Cashier records the cash tendered.	3. Show the balance due back to the Customer.
4. The Cashier deposits the cash received and extracts the balance owing. The Cashier gives the balance owing and the printed receipt to the Customer.	

Alternative Courses

- Line 3. If cash the amount tendered is not enough, exception handling
- Line 4: Insufficient cash in drawer to pay balance. Ask for cash from supervisor.

Using exactly the same techniques in the creation of **Pay by Cash**, we can create two use cases **Pay by Credit** and **Pay by Cheque**.

Now it is not difficult to create the general use case **Buy Items** that can be written as follows.

Buy Items

Typical Course of Events

Actor Action	System Response
1. This use case begins when a Customer arrives at a POST checkout with items to purchase.	
2. The Cashier records the identifier from each item.	3. Determines the item price and adds the item information to the running sales transaction.
If there is more than one of the same item, the Cashier can enter the quantity as well.	The description and price of the current item are presented.
4. On completion of the item entry, the Cashier indicates to the POST that item entry is completed.	5 Calculates and presents the sale total.
6. The Cashier tells the Customer the total	
7. The Customer chooses payment method: (a) If cash payment, **initiate** *Pay by Cash.* (b) If credit payment, **initiate** *Pay by Credit.* (c) If cheque payment, **initiate** *Pay by Cheque.*	
	8. Logs the completed sale.
	9. Prints a receipt.
10. The Cashier gives the printed receipt to the Customer.	
11. The Customer leaves with the items purchased.	

Alternative Courses

- Line 2: Invalid identifier entered. Indicate error.
- Line 7: Customer didn't have enough cash. Cancel sales transaction.

In general, a use case may contain decision points. If one of these decision paths represents the overwhelming typical case, and the others alternatives are rare, unusual or exceptional, then the typical case should be the only one written about in the *Typical Course of events*, and the alternatives should be written in the *Alternative Courses* section.

However, if the decision point represents alternatives which all relatively equal and normal in their likelihood, this is true of the payment types, then we have to describe them as individual use cases and they can be *used* by a 'main' use case, such as **Buy Items**.

UML provides a notation for describing such a *uses* relationship between use cases, and this is illustrated in Figure 4.3.

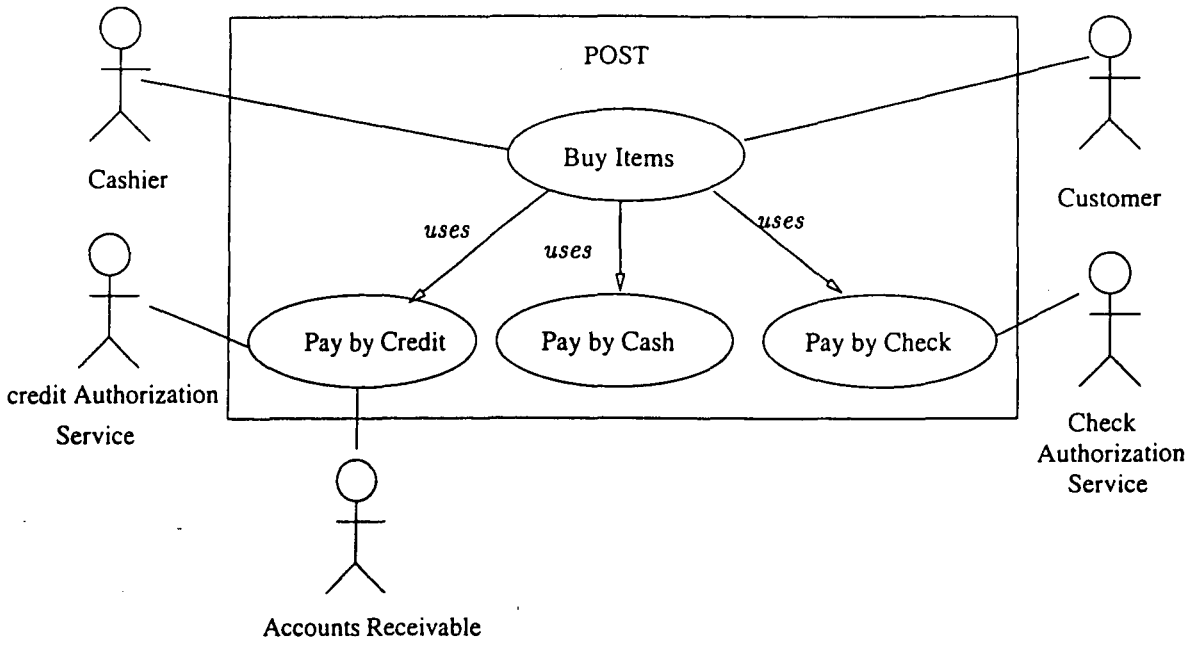


Figure 4.3: Relating use cases with a *uses* relationship

4.2.6 Use cases within a development process

This section summarizes the activities and artifacts of use case analysis.

1. After system functions have been listed, then identify actors and use cases.
2. Write all use cases in the *high-level* format. You may like to categorize them as primary, secondary or optional.
3. Draw a use case diagram for the system.

4. Relate use cases and illustrate relationships in the use case diagram.

5. Write the most critical, influential and risky use cases in the *expanded essential* format to better understand and estate the nature and size of the problem. Defer writing the expanded essential form of the less critical use cases until the design or event the implementation starts.

6. Ideally, *real* use cases should be deferred until the design phase of a development cycle, since their creation involves design decisions. However, it is sometimes necessary to create some real use case during the early requirement phase if

   - Concrete descriptions significantly aid comprehension.
   - Clients demand specifying their processes in this fashion.
   - For prototype purpose.

7. You may like to rank the use cases to plan which should be taken into the design and implementation phases first.

4.2.7 Actors and use cases of the POST system

Using the techniques of this section, a sample list (not an exhaustive) of relevant actors and use cases they initiate include:

Actor	Use Case
Cashier	Log In Cash Out
Customer	Buy Items Refund Items
Manager	Start Up Shut Down
System Administrator	Add New Users

We leave the construction of the use cases as exercises.

4.2.8 Summing up

Use case definition

A use case specifies a sequence of actions that the system can perform and that yields an observable result of value to a particular actor. A use case is full end-to-end story about the use of the system to carry out a task, not an arbitary combination of a number of steps of computation. For example, we should not combine two tasks into a use case. such a Borrow a Book and Return a Book, if it is not alway the case that one must be carried out after the other.

Why use cases?

The reasons why use cases are good for requirement capture include

- They do not only answer the question *what the system should do* but answer it in terms of *what the system should do for each user/actor.* They therefore identify the system functions a system provides to add values to its users, and help to remove functions that they do not add values to the users.
- They identify system functions and the relationship among the system functions.
- They also identify concepts and objects involved in the application domain. These concepts and objects are very important in modeling the architecture of the system, and are later in the design realized as software classes and objects.
- Use cases are also used as "placeholders" for *nonfunctional requirements*, such as performance, availability, accuracy, and security requirements that are specific to a use case. For example, the following requirement can be attached to the **Withdraw Money** use case: The response time for a Bank Customer measured from selecting the amount to withdraw to the delivery of the notes should be less than 30 seconds in 95% of all cases.
- Use cases are also important for project plan, system design and testing.

Use Cases Specify the System

The **use-case model** represents all the use cases and their associated actors and specifies all the functional requirements and most of the nonfunctional requirements of the system. *Requirement capture* is to *capture requirements as use cases.* Its main purpose is to work with the customers and users to identify the use cases and to create the use-case model. The use-case model serves as an agreement between the client and the developers, and it provides essential input for analysis, design, and testing.

We find the use case by looking at how the users need to use the system to do their work. Each such way of using the system that adds value to the user is a candidate use case. These candidates will then be elaborated on, changed, divided into smaller use cases, or integrated into more complete use cases. The use-case model is almost finished when it captures all functional requirements correctly in a way that the customer, users and developers can understand.

4.3 Questions

1. As we said in Chapter 2, the task of the requirements capture and analysis is difficult and complex for a variety of reasons.

   - The requirements are the client's requirements which are actually the needs of the users of the system to develop, rather than those of the developers.
   - Any system has many users (or types of users), and while each user may know what he or she does, no one can see the whole picture.
   - Very often, users do not know what the requirements are or how to specify them in a precise manner, either.
   - Different types of users have different, sometimes conflicting, requirements.

   How can you, as a requirement analyst, help to over come any of these problems, and to create a use-case model?

2. What are the differences and relationships between system functions and use cases? How can they be identified?

3. Each use case has to represent a task, or a coherent unit of functionality, which the system is required to support. Normally, this means that the use case has value for at least one of the actors. An actor for whom a use case has value is normally called a *beneficiary* of the use case. Discuss the following questions.

   (a) Consider the use case Buy Items with cash in the POST system, who is a beneficiary and what value he/she gets from the system?

   (b) How important is it to identify the beneficiaries of each use case?

   (c) Is a initiating actor always a beneficiary?

   (d) Does a beneficiary have to interact with the system carrying out a task to get benefit?

   (e) What is the role of an actor of a use case who is not a beneficiary, do you think that we do not need to show such an actor in the use case model?

4. Discuss the possible use of use cases for the planning of the project.

5. Discuss the possible use of use cases in *system validation*.

6. About 25% of projects were canceled simply because someone somewhere decided it wasn't worth going ahead with. Discuss the possible use of use cases in dealing with *political aspects* of such in a project.

7. Discuss the possible problems with use cases.

8. Consider an example of a library system. Assume that a member of the library is allowed to borrow up to six items (e.g. a copy of a book, a journals, etc). The system supports to carry out the tasks of *borrowing a copy of a book*, *extending a loan*, and *checking for reservation*. Write the use cases that represent these tasks, and use a use-case diagram to represent relationships among these uses cases.

   Do you realize, from this problem, an issue of factoring, sharing, and reusing functionality? Can you discuss the advantages and pitfalls in documenting shared and reused functionality like this on a use-case diagram?

9. Suppose that you are involved in the design of East-West University's library (described in a question at the end of Chapter 3) browsing and loan system. What are some of the important use cases you might develop in the analysis phase (simply enumerating them with brief discussions or present them in the high-level format is sufficient)? Try to identify at least 4-5 cases, and draw a use-case diagram to illustrate them

Chapter 5

Requirement Capture and Analysis – Conceptual Model

Topics of Chapter 5

- Identifying concepts in the application domain which are related to the system to be developed and building a conceptual model
- Associations between concepts (classes)
- Attributes of Classes
- Class diagrams and their use for describing a conceptual model

5.1 Conceptual Model – Concepts and Classes

An important and typical activity in object-oriented *requirement analysis* is to *identify concepts* related to the requirements and to create a *conceptual model* of the *domain*. The term *domain* covers the application area we are working with, e.g. the retail store in our POST case study. This section is to introduce the skill in identifying concepts which are meaningful in the problem domain, and the notation for representing such a model.

5.1.1 Conceptual Models

A *conceptual model* illustrates abstract and meaningful concepts in the problem domain. The creation of concepts is the most essential *object-oriented* step in analysis or investigation of the problem domain for

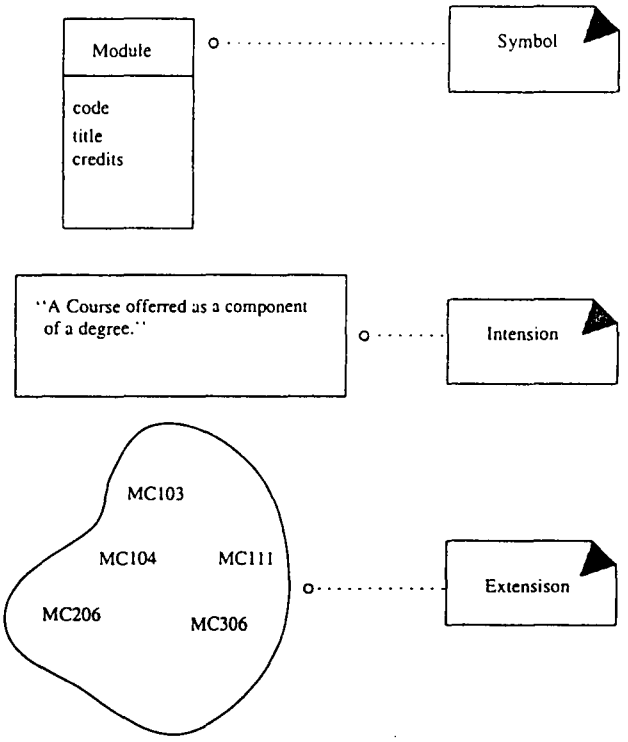


Figure 5.1: A concept has a symbol, intension, and extension

building genuinely extensible software with reuse. The aim of this step is *decomposition of the problem into individual concepts or objects.*

Concepts

Informally, a *concept* is an idea, thing, or object. More formally, a concept may be considered in terms of its symbol, intension, and extension:

- *Symbol* – words or images representing a concept. It can be referred to when we talk about the concept.
- *Intension* – the definition of a concept.
- *Extension* – the set of examples or instances to which the concept applies.

Each individual example that the concept applies is called an *instance* of the concept.

For an example, the symbol *Module* in University of Leicester is a concept with

- the intension to "represent a course offered as part of a degree in that university" having a code, title, and number of credits; and
- the extension of all the modules being offered in the university.

This concept is illustrated in Figure 5.1

5.1.2 Defining terms and modelling notation for a concept

In the UML, the terms *class* and *type* are used, but NOT *concept*. We may interchange these terms as far as we are clear that we are at the requirement analysis stage and we are understanding and investigating the problem domain.

And in the UML, we use the notation shown in Figure 5.2 to denote a class.

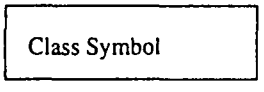


Figure 5.2: UML notation for a class

Each instance of a class is called an *object* of the class. For example, see Figure 5.3. Therefore, a *class* defines a set of *objects*.

The notions of class and object are interwoven as one cannot exist without the other, and any object belongs to a class. The differences are:

- an object is a concrete entity – exists in space and time (*persistence property of objects*);
- a class is an abstraction of a set of objects.

The UML definition of a *class* is "a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics". This covers classes used at all stages in an OO development process. We shall see the concepts of *attributes*, *operations*, *methods*, and *relationships* one by one when we come to the particular stages of the OO development of process.

A defining aspect of objects is that every object is distinguishable from every other object. This is true even if two objects have exactly the same properties. For example, two instances of *Student* may have the same name, age, doing the same degree, in the same year, taking the same courses, etc. The property that enables objects to be distinguished from each other is know as an object's *identity*.

Some other important features of objects are:

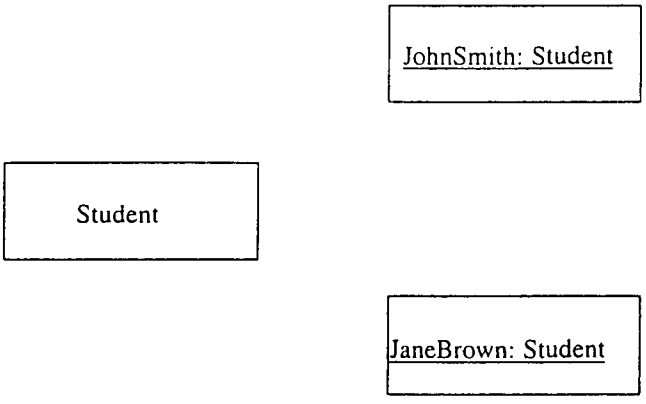


Figure 5.3: Class and Object

- *Object's Identity:* Every object is distinguishable from every other object. This is true even if two objects have exactly the same properties.

- *Object's persistence:* Every object has a life time, and this property implies the static nature of the system.

- An object has behaviour and may act on other objects and/or may be acted on by other objects, this property implies the dynamic nature of the system

- An object may be in different state at different and may behave differently in different states.

5.1.3 Identifying Concepts

A central distinction between OOA and structured analysis is decomposition by concepts (objects) rather than decomposition by functions. The key point in this decomposition is to find what can behave, and then to decide later on in the design and implementation how they behave to realize the system functionality.

There are usually two strategies to identify concepts. The first one is to *find concepts from the textual descriptions of the problem domain according to concept category list.* Concepts and objects (things) can be divided into different categories according to their nature of its instances. The concept category list given below has been found useful in identifying concepts.

Concept Category	Examples
physical or tangible objects (or things)	*POST, House, Car, Sheep, People, Airplane*
places	*Store, Office, Airport, PoliceSation*
documents, specifications, designs, or descriptions of things	*ProductSpecification, ModuleDescription, FlightDescription*
transactions	*Sale, Payment, Reservation*
roles of people	*Cashier, Student, Doctor, Pilot*
containers of other things	*Store, Bin, Library, Airplane*
things in a container	*Item, Book, Passenger*
other computers or electro-mechanical systems external to our system	*CreditCardAuthorizationSystem, AirTrafficControl*
abstract noun concepts	*Hunger, Acrophobia*
organisations	*SalesDepartment, Club, ObjectAirline*
historic events, incidents	*Sale, Robbery, Meeting, Flight, Crash, Landing*
processes (often *not* represented as a concept, but may be)	*SellingAProduct, BookingASeat*
rules and policies	*RefundPolicy, CancellationPolicy*
catalogs	*ProductCatalog, PartsCatalog*
records of finance, work, contracts, legal matters	*Receipt, Ledger, EmploymentContract, MaintenanceLog*
financial instruments and services	*LineOfCredit, Stock*
manuals, books	*EmployeeManual, RepairManual*

This strategy is to create a list of *candidate concepts* from the client's requirements description, initial investigation reports, system functions definitions, and use cases.

Notice that the categories are not mutually exclusive and one concept may belong to more than category.

Another useful and simple technique for identification of concepts is to identify the *noun and noun phrases* in the textual descriptions of a problem domain, and consider them as candidate concepts or attributes.

Warning:

Care must be applied when these methods are used; mechanical noun-to-concept mapping is not possible, words in natural languages are ambiguous, and concept categories may include concepts which are about either *attributes*, *events*, *operations* which should **not** be modelled as *classes*. We should concentrate on the objects/classes involved in the *realization* of the use cases.

Consider our POST application, from the use case **Buy Items with Cash**, we can identify the some noun phrases. Some of thenoun phrases in the use case are candidate concepts; some may be attributes of concepts (see Section 5.3). For example, **price** is obviously an attribute of an **item** or a **product description**. Also, the concept **price** is not easy to fit in any of the categories in the *Concept Category List*, and thus it should not a candidate concept. The following list is constrained to the requirements and the use case **Buy Items with Cash**:

Post	*ProductSpecification*
Item	*SalesLineItem*
Store	*Cashier*
Sale	*Customer*
Payment	*Manager*
ProductCatalog	

This list of concept names may be represented graphical in the UML *static structure diagram* notation to show the *concepts only* conceptual model (See Figure 5.4).

We may find it is a bit odd that *Receipt* is not listed as a candidate concept, as a receipt is a record of a sale and payment and a relatively important concept. However, here are some factors to consider:

- In general, showing a record or a report of a thing, such as a receipt, in a conceptual model is not very useful since all its information can be derived from other concepts. For example, a receipt can be produced by extracting the relevant information from the *ProductSpecification* and the *Payment*.
- A record or a report, such as a receipt, sometimes has a special role in terms of the application and need to be maintained. For example *Receipt* confers the right to its bearer to return his/her bought items. In this case, it should be shown in the conceptual model. However, item returns are not being considered, and thus *Receipt* is not at the moment included in the list. When the system evolves and needs to tackle the *Return Items* use case, it would be justified to include *Receipt*.

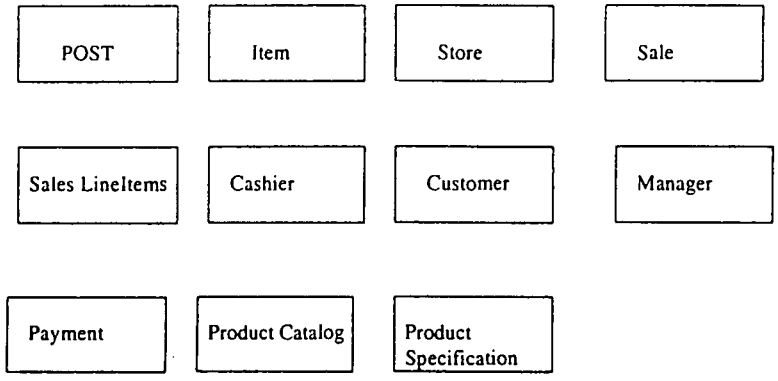


Figure 5.4: Initial conceptual model for the point-of-sale domain (concepts only)

Obviously, these factors can be very easily overlooked. However, we should remember our goal is to create a conceptual model of interesting and meaningful concepts in the domain under consideration. The following guidelines are useful:

- It is better to overspecify a conceptual model with lots of fine-grained concepts, than to underspecify it.
- Do not exclude a concept simply because the requirements do not indicate an obvious need to remember information about it.
- It is common to miss concepts during the initial identification phase, and to discover them later during the consideration of attributes or associations, or during the design phase. When found, they are added to the conceptual model.
- Put a *concept* down as a candidate in the conceptual model when you are not sure it must not be included.

As a rule of thumb, a conceptual model is not absolutely correct or wrong, but more or less useful; it is a tool of communication.

5.1.4 Conceptual modelling guidelines

The *mapmaker strategy* applies to both maps and conceptual models:

- *Use the existing names in the territory* Map makers do not change the names of cities on the map. For a conceptual model, this means to *use the vocabulary of the domain when naming concepts and attributes*. For example, we use *POST* rather than *Register*.

- *Exclude irrelevant features* A mapmaker deletes things from a map if they are not considered relevant to the purpose of the map: for example, topography or populations need not be shown. For examples, we may exclude *Pen* and *PaperBag* from our conceptual model for the current set of requirements since they do not have any obvious noteworthy role.

- *Do not add things that are not there* A mapmaker does not show things that are not there, such as a mountain that does not exist. For example, we should include *Dog* in a model of a library.

Finally, remember that *a conceptual model focuses on domain concepts, not software entities.*

5.2 Conceptual Model – Associations

A conceptual model with totally independent concepts only are obviously useless, as objects in different classes must be related to each other so that they can interact and collaborate with each other to carry out processes.

In UML, an *association* is a *relationship* between two classes that specifies how instances of the classes can be linked together to work together. In the same sense that instances of a class are objects, instances of an association are *links* between objects of the two classes – this what we meant that objects in the same class "share the same relationships" (see the end of Section 5.1).

For example, an association called *takes* links the classes *Student* and *Module*. An individual student, say one with the name *John Smith* is linked with a particular module, say the one with the code *MC206* if this student takes this module.

In UML, an association between two classes is denoted by a line joining the two classes, see Figure 5.5.

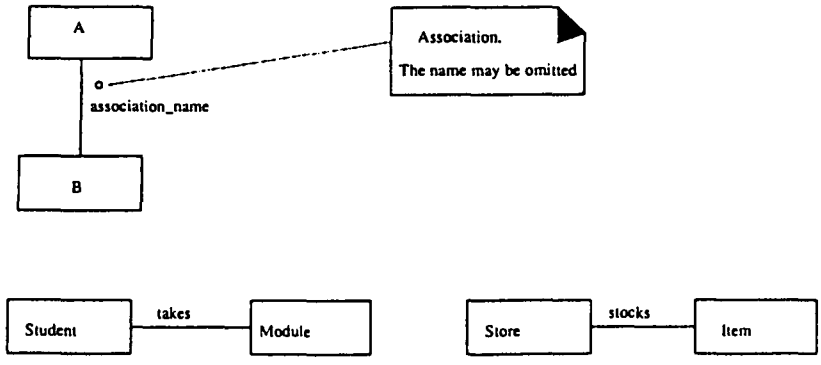


Figure 5.5: An association between two classes

Multiplicity

With respect to an association between classes A and B, an important information is about how many objects of class A can be associated with one object of B, at a particular moment in time. We use *multiplicity* of A to represent this information. Figure 5.6 shows some examples of multiplicity expressions and their meanings.

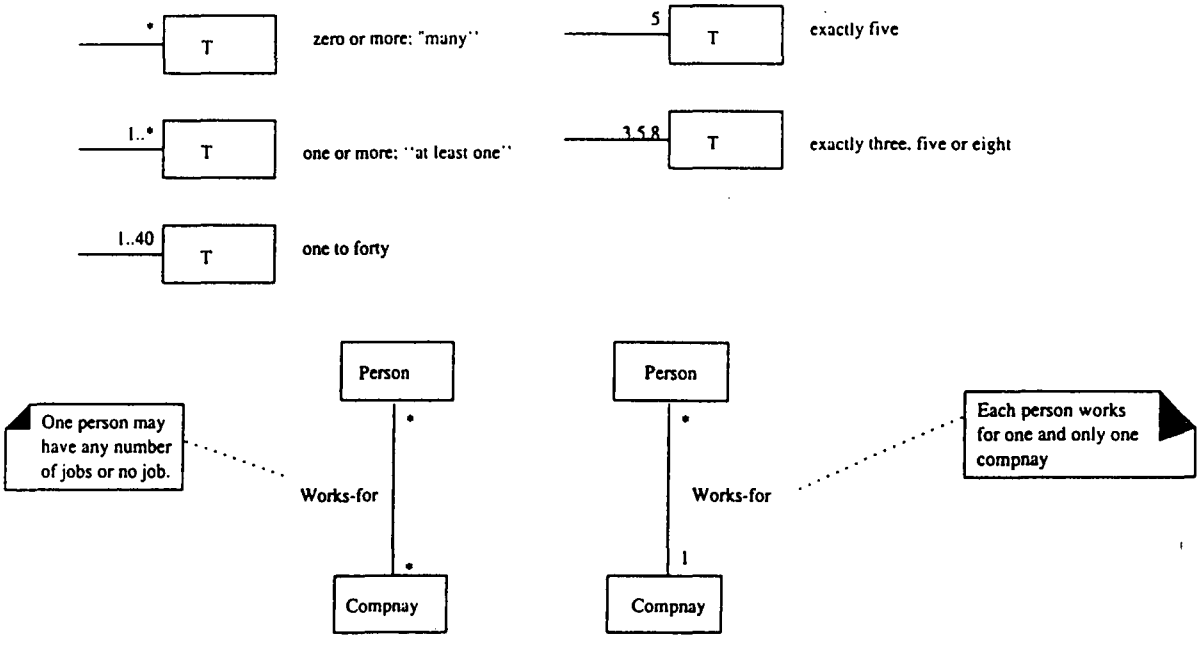


Figure 5.6: Multiplicity values

Determining multiplicity often exposes hidden constraints built into the model. For example, whether *Works-for* association between *Person* and *Company* in Figure 5.6 is one-to-many or many-to-many depends on the application. A tax collection application would permit a person to work for multiple companies. On the other hand, an auto workers' union maintaining member records may consider second jobs irrelevant (See Figure 5.6).

More examples of associations are shown in Figure 5.7. The default direction to read an association in such a diagram is *from left to right* or *from top to bottom*, otherwise the small arrow shows the reading direction.

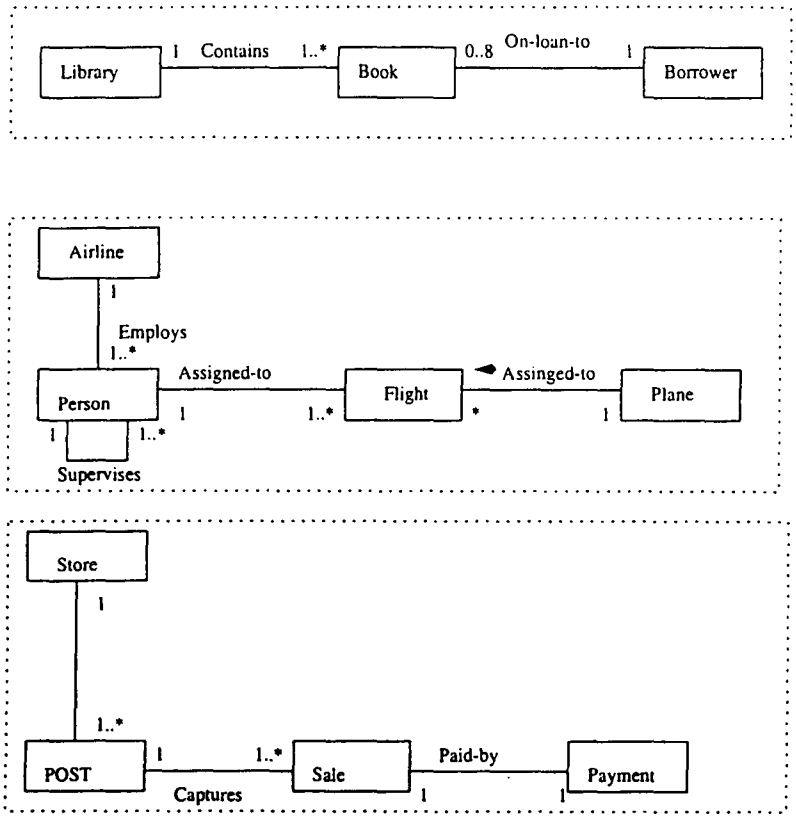


Figure 5.7: Examples of associations

Roles of associations

Each end of an association[1] is called a *role* of the association, which may has a *role name*. Naming a role in a conceptual model is sometimes useful. For example, the role names *boss* and *worker* distinguish two employees who work for a company and participate in the *Manages* association (See Figure 5.8).

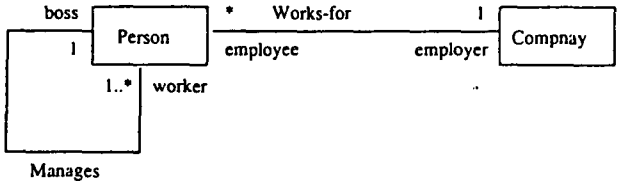


Figure 5.8: Examples of role names

When we come up to the design and implementation of the system, roles provide a way of viewing an association as a traversal from one object to a set of associated objects.

[1] In this course, we are mainly concerned with *binary associations* which relate two classes. In general, an association may be *ternary or even higher order* which relate three or more classes.

The purpose of an association and a link between two objects Objects may have many sorts of relationship, and thus classes (or concepts) may have all sorts of associations in a problem domain. Whether an association is useful or not depends on whether it fulfill the following purpose.

- An association between two classes is to provide *physical* or *conceptual* connections between objects of the classes.
- Only objects that are associated with each other can collaborate with each other through the links.

Booch describes the role of a link between objects as follows:

> "A link denotes the specific association through which one object (the client) applies the services of another object (the supplier), or through which one object may navigate to another".

This can be taken as a criterion for whether two objects should be linked.

Multiple associations between two classes Two classes may have multiple associations between them; this is not uncommon. For example, in the domain of an airline example, there are two relationships between a *Flight* and an *Airport*, *Flies-to* and *Flies-from*, as shown in Figure 5.9.

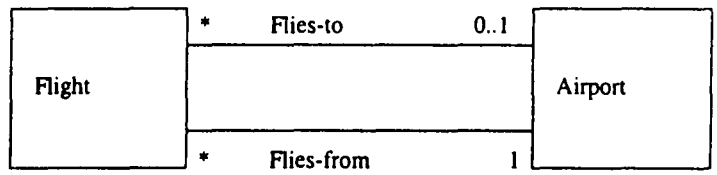


Figure 5.9: Multiple associations

These two associations are distinctly different relationships between flights and airports. Note that not every flight is guaranteed to land at any airport! However, there is always one airport from where a flight flies.

5.2.1 Strategies for identifying associations

For the requirement analysis, we have the following principles:

- An *useful association* usually implies knowledge of a relationship that needs to be preserved for some duration ("need-to-know" association); and
- an important link between two objects should fulfill the role to provide a means for the objects to collaborate or interact with each other.

Like the identification of concepts, an *Association Category List* are helpful. The following list contains common categories that are usually worth considering.

Association Category	Examples
A is a physical part of B	*Drawer-POST, Wing-Airplane*
A is a logical part of B	*SalesLineItem-Sale, FlightLeg-FlightRoute*
A is a kind/subclass/subtype of B	*CashPayment-Payment, NonstopFlight-Flight*
A is physically contained in/on B	*POST-Store, Item-Self*
A is logically contained in B	*ItemDescription-Catalog, Flight-FlightSchedule*
A is a description for B	*ItemDescription—Item, FlightDescription-Flight*
A is a line item of a transaction or report B	*SaleLineItem-Sale*
A is known/logged/recorded/ reported/captured in B	*Sale-POST, Reservation-FlightManifest*
A is member of B	*Cashier-Store, Pilot-Airline*
A is an organisational subunit of B	*Department-Store, Maintenance-Airline*
A uses or manages B	*Cashier-POST, Pilot-Airplane*
A communicates with B	*Customer-Cashier, ReservationAgent-Passenger*
A is related to a transaction B	*Customer-Payment, Passenger-Ticket*
A is a transaction related to another transaction B	*Payment-Sale, Reservation-Cancellation*
A is next to B	*POST-POST, City-City*
A is owned by B	*POST-Store, Plane-Airline*

Naming Associations

- Name an association based on a *ClassName-VerbPhrase-TypeName* format where the verb phrase creates a sequence that is readable and meaningful in the model context.
- Association names should start with a capital letter.
- A verb phrase should be constructed with hyphens.

High priority associations Here are some high priority associations that are invariably useful to include in a conceptual model:

- A is a *physical or logical part* of B.
- A is *physically or logically contained* in/on B.
- A is *recorded* in B.

Association and implementation During the analysis phase, an association is *not* a statement about data flows, instance variables, or object connections in a software solution; it is a statement that a relationship is meaningful in a purely analytical sense – in the real world. Practically speaking, many of these relationships will typically be implemented in software as paths of navigation and visibility, but their presence in an investigative or analytical view of a conceptual model does not require their implementation.

When creating a conceptual model, we define associations that are not necessary during construction. Conversely, we may discover associations that needed to be implemented but were missed during the analysis phase. In that case, the conceptual model should be updated to reflect these discoveries.

Point-of-sale domain associations

We can now add associations to the POST system conceptual model created in the previous section (see Figure 5.4).

We first apply the *need-to-know* policy to the use case **Buy Items with Cash**, from that we can identify the following association:

- *POST Captures Sale* in order to know the current sale, generate a total, print a receipt.
- *Sale Paid-by Payment* in order to know if the sale has been paid, relate the amount tendered to the sale total, and print a receipt.
- *ProductCatalog Records ItemsSpecification* in order to retrieve an ItemSpecification, given a UPC.

We can run the Association Category Checklist, based on Figure 5.4 and the **Buy Items wish Cash** use case.

Association Category	**POST System**
A is a physical part of B	*not applicable*
A is a logical part of B	*SalesLineItem—Sale*
A is a kind/subclass/subtype of B	*not applicable as we only consider on kind of payment*
A is physically contained in/on B	*POST—Store, Item–Store*
A is logically contained in B	*ProductDescription—ProductCatalog* *ProductCatalog–Store*
A is a description for B	*ProductDescription—Item*
A is a line item of a transaction or report B	*SaleLineItem—Sale*
A is known/logged/recorded/ reported/captured in B	*(current) Sale—POST* *(completed) Sale—Store*
A is member of B	*Cashier—Store*
A is an organisational subunit of B	*not applicable*
A uses or manages B	*Cashier—POST* *Manager—POST* *Manager—Cashier, but probably N/A*
A communicates with B	*Customer—Cashier*
A is related to a transaction B	*Customer—Payment* *Cashier—Payment*
A is a transaction related to another transaction B	*Payment—Sale*
A is next to B	*POST—POST, but probably N/A*
A is owned by B	*POST—Store*

This check derives the conceptual model shown in Figure 5.10 for the POST system, which extended the model in Figure 5.4 with associations.

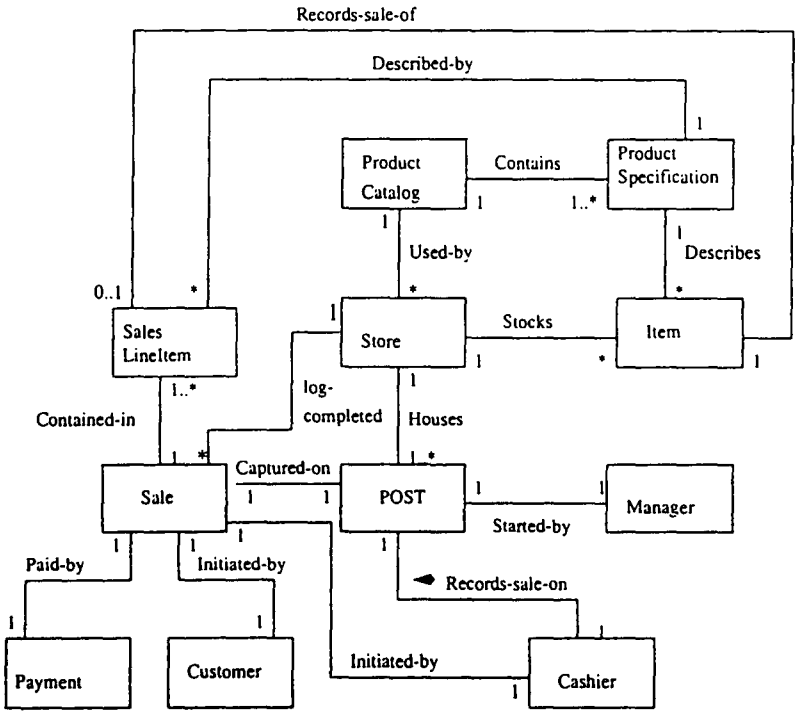


Figure 5.10: Conceptual model for the point-of-sale domain (concepts and associations)

5.2.2 The aggregation association

Most OO modelling technologies singles the "part-of" relationship between two objects for special treatment.

Aggregation is a kind of association used to model whole-part relationships between objects. The whole is generally called the *composite*.

Aggregation is shown in the UML with a hollow or filled diamond symbol at the composite end of a whole-part association (Figure 5.11).

There are two kinds of aggregation:

- *Composite aggregation* or *composition* means that the multiplicity at the composite end may be at most one, and is signified with a filled diamond. It implies that the composite solely owns the part, and that they are in a tree structure hierarchy; it is the most common form of aggregation shown in models.
- *Shared aggregation* means that the multiplicity at the composite end may be more than one, and is signified with a hollow diamond. It implies that the part may be in many composite instances. Shared aggregate seldom (if ever) exists in physical aggregates, but rather in nonphysical concepts.

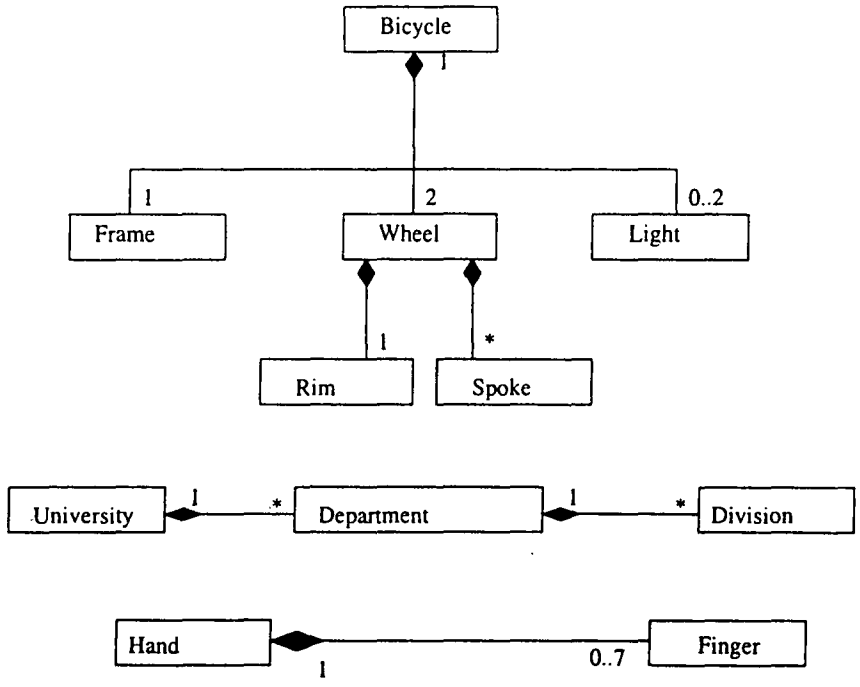


Figure 5.11: Aggregations

For instance, a UML package may be considered to aggregate its elements. But an element may be referenced in more than one package (it is owned by one, and referenced in others). Another example is that a directory can contain a number of files, each of which may be another directory or a plain file. A file or a directory may be contained in a number of directories. These examples are shown in Figure 5.12.

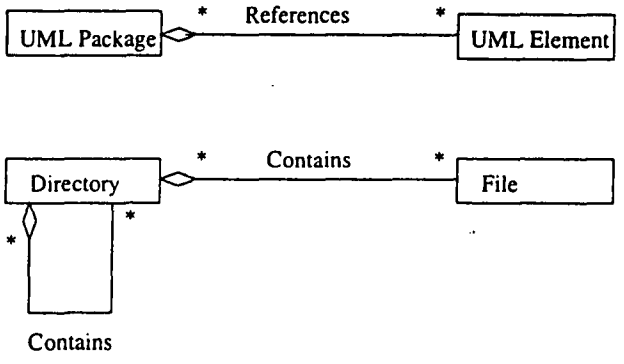


Figure 5.12: Shared Aggregations

However, we must note two important properties of aggregation;

1. *Antisymmetry:* this states that if an object a is related to an object b by an aggregation, then it is

not possible for b to be related to a by the same aggregation. In other words, if b is a part of a then a cannot be a part of b.

2. *Transitivity:* this states that if a is related to b by an aggregation link, and b is related to c by the same aggregation, then a is also linked to c.

When to show aggregation

Here are some guidelines that suggest when to show aggregation:

- The lifetime of the part is bound within the lifetime of the composite–there is a create-delete dependency of the part on the whole. The part may not exist outside of the lifetime of the whole.
- There is an obvious whole-part physical or logical assembly.
- Some properties of the composite propagate to the parts, such as its location.
- Operations applied to the composite propagate to the parts, such as destruction, movement, recording.
- *If you are not sure when to use it, ignore it and stick to plain association.* Most of the benefits of discovering and showing aggregation relate to the software solution phases.

Based on the above discussion, we can polish the conceptual model in Figure 5.10 by replacing the *Contain-in* association between *SalesLineItem* and *Sale*, and the *Contains* association between *ProductCatalog* and *ProductSpecification* with the aggregation associations shown in Figure 5.13

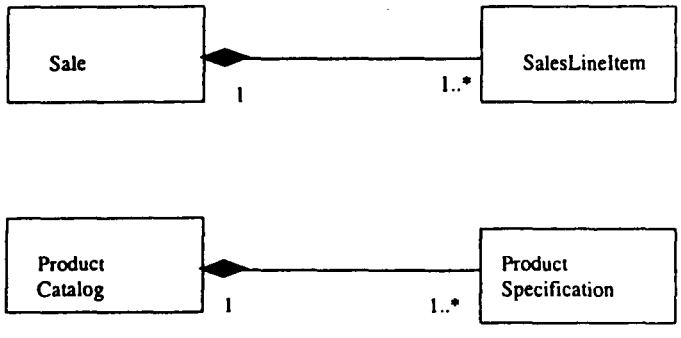


Figure 5.13: Aggregation in the point-of-sale application

Example Consider an Environmental Information System (ENFORMS) that is used to establish a dynamic network of distributed earth-science data archieves with integrated on-line access services. The problem domain in which ENFORMS operates consists of a set of data centers that may be physically distributed throughout a geographic region. ENFORMS allows these data centers to make their data available through a regional data sharing network (established by ENFORMS) and provide a variety of

tools (a mapping utility, for example) for locating and manipulating that data. the most general requirements that guide the problem analysis are as follows.

1. Data centers must be able to independently manage both content and access mechanisms for their data stores.
2. Data centers may disable access to their data at any time (i.e. go off-line).
3. Data centers may make their data stores avaolable at any time (i.e. go on-line).
4. at any time, all on-line data centers must be available to any ENFORMS user.

The concepts and associations identified are as follows.

- access: $User \times DataCenter \times Dataset$ $[*, 1, *]$
- uses: $User \times NetworkManager$ $[*, 1]$
- is-online: $DataCenter \times NetworkManager$ $[*, 0..1]$
- queries: $User \times DataCenter$ [0..1, *]
- stores: $DataCenter \times Dataset$ [Compositie Aggregation]

The above simple conceptual model conveys a significant amount of information about the possible *instances* of an ENFORMS system. One such an instance ($\mathcal{I}_1$) could be a system with three Data Centers d_1, d_2, d_3 and one User u that *queries* the Data Centers d_1 and d_3.

Another instance can be a system with two Users u_1 and u_2 using a Network Manager n; one Data Center d that *stores* two Datasets i_1 and i_2, and is in the system and it *is on-line with* the Network Manager n. In this instance, we can also assume that only u_2 *access* i_1.

We can also show that a state that is not consistent with a conceptual model. For example, a system with two Network Managers n_1 and n_2 and one *user* that uses the two managers as entry points is not a valid state as the conceptual model indicates that a user can enter an ENFORMS network via a single network manager.

5.3 Conceptual Model–Attributes

Each instance of a concept may have some useful properties. For examples, a *Sale* has a date and time; a *Module* has a code, title, and number of credits, a *Student* has a name and age, etc..

An *attribute* of a class is the abstraction of a single characteristic or a property of entities that have been abstracted as objects of the class. At any given moment of time, the *attribute* of an individual object in the class is a logical data value representing the corresponding property of the object, and called the *value of attribute for the object* at that time. One object has exactly one value for each attribute at any given time. Therefore, the value for an attribute of an object may change over time. For examples,

- *time* and *date* are attributes of class *Sale*, and an instance of *Sale* can be a sale at 13.30 on 1/10/1998.
- *code*, *title*, and *credit* are three attributes of class *Module*, and an instance of *Module* can have a code $MC206$, title: Software Engineering and System Development, and credit: 20.
- *name* and *age* are attributes of *Student*, an individual student can have the name John Smith, and age 19.

An attribute has a name such as *date*, and can specify the *type* of the data described by the attribute, and a default (or initial) value for the attribute.

In UML, attributes are written in the class box, separated from the class name by a horizontal line, as shown in Figure 5.14. The features other than the name are optional, however. Especially in the early stages of the development it is very common to show nothing more than the name of the attribute.

Now we can understand that the objects of a class "share the same attributes" (see the end of Section 5.1).

5.3.1 Adding attributes to classes

For each class in a conceptual model created in Section 5.2, we would like to find a set of attributes that are

- *complete:* Capture all relevant information about an object,
- *fully factored:* each attribute captures a different property of an object,
- *mutually independent:* For each object, the values of the attributes are independent of each other, i.e. we should do our best to avoid derived attributes,
- *relevant to the requirements and use cases:* Focus on those attributes for which the requirements suggest or imply a need to remember information.

The common mistakes in OO modelling is to represent something as an attribute when it should have been a concept (class) or an association. In the following, we see how to avoid from making this mistake.

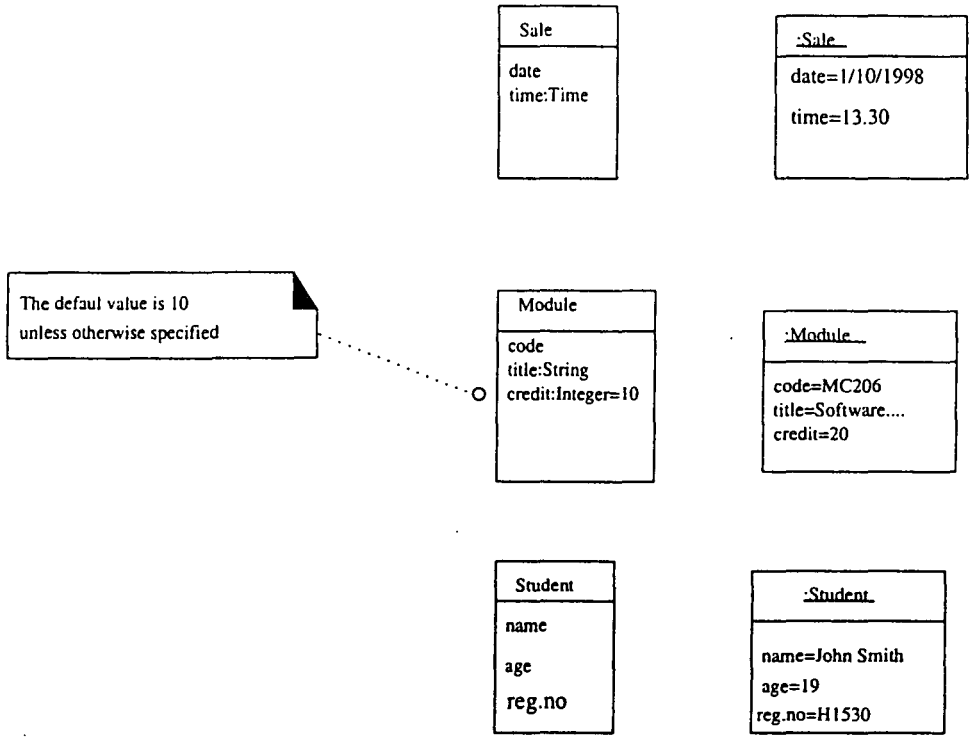


Figure 5.14: Attributes

Keep attributes simple

The attributes in a conceptual model should preferably be *simple attributes* or *pure data types*:

- Intuitively, most simple attribute types are what often thought as *primitive* data types: *Boolean, Date, Number, String(Text), Time.*
- Other simple types include *Address, Colour, Geometrics (Points, Rectangle, ...), Phone Numbers, Social Security Number, Universal Product Code (UPC), Post Code, enumerated types.*

The type of an attribute should *not* normally be a complex domain concept, such as *Sale* or *Airport*. For examples (see Figure 5.15):

- It is not desirable to let *Cashier* to have an attribute *currentPOST* to describe the POST that a Cashier is currently using. This is because the type of *currentPOST* is meant to be *POST*, which is not a simple attribute type. The most sensible way to express that a Cashier uses a POST is with an association, not with an attribute.
- It is not desirable for a *destination airport* to be an attribute of a *Flight*, as a destination airport is not really a string; it is a complex thing that occupies many square kilometers of space. Therefore,

Flight should be related to *Airport* via an association, say *Flies-to*, not with an attribute.

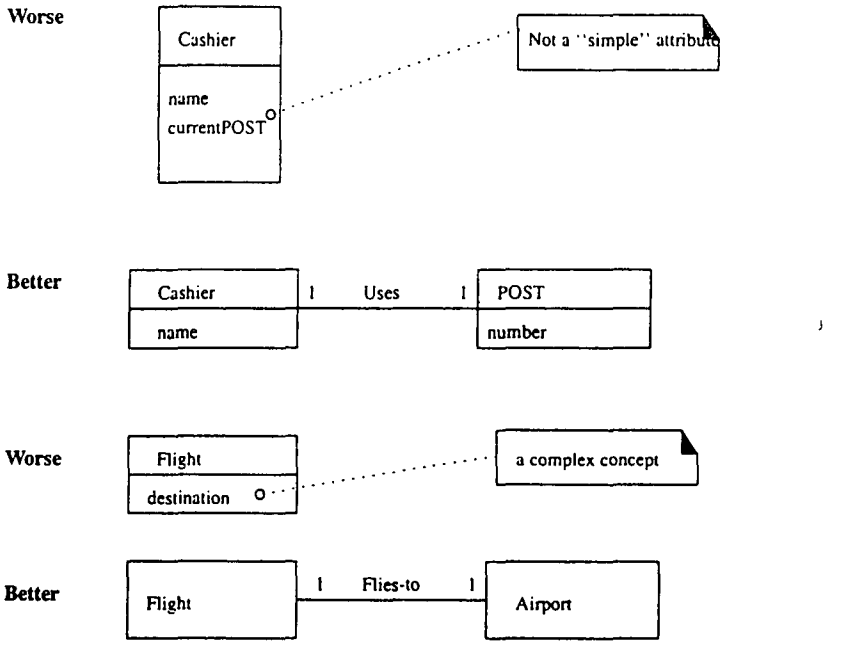


Figure 5.15: Avoid representing complex domain concepts as attributes; use associations

The first guideline is

Relate concepts with an association, not with an attribute.

What about attributes in code?

The restriction that attributes in the conceptual model be only of simple data type does not imply that C++, Java, or Smalltalk attributes (data members, instance variables) must only be simple, primitive data types.

Pure data value

More generally, attributes should be *pure data value* (or *DataTypes* in UML)–those for which unique identity is not meaningful in the context of our model or system. For example, it is not (usually) meaningful to distinguish between:

- Separate instances of the *Number 5*.

- Separate instances of the *String 'cat'*.
- Separate instances of the *PhoneNumber* that contain the same number.
- Separate instances of the *address* that contain the same address.

However, it *is meaningful* to distinguish (by identity) between two instances of a *Person* whose name are both "Jill Smith" because the two instances can represent separate individuals with the same name.

So the second guideline for identifying attributes is:

> *An element of a pure data type may be illustrated in an attribute box section of another concept, although it is also acceptable to model it as a distinct concept.*

Pure data values are also known as *value objects*.

The notion of pure data values is subtle. As a rule of thumb,

> *stick to the basic test of "simple" attribute types: make it an attribute if it is naturally thought of as number, string, boolean, or time (and so on); otherwise represent is as a separate concept.*

We always have the following rule.

> *If in doubt, define something as a separate concept rather than as an attribute.*

Design creep: no attributes as foreign key

Attributes should not be used to relate concepts in the conceptual model, but to store some information about the objects themselves. The most common violation of this principle is to add a kind of *foreign key attribute*, as is typically done in relational database designs, in order to associate two types. For example, in Figure 5.16, the *currentPOSTNumber* attribute in the Cashier type is undesirable because its purpose is to relate the *Cashier* to a *POST* object. The better way to express that a *Cashier* uses a *POST* is with an association, not with a foreign key attribute. Once again

> *relate types with an association, not with an attribute.*

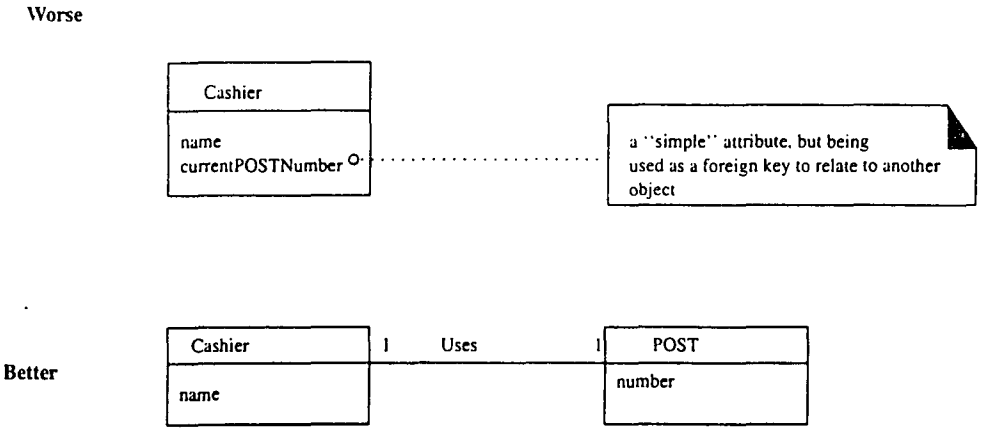


Figure 5.16: Do not use attributes as foreign keys

Non-primitive attribute types

The type of an attribute may be expressed as a non-primitive type in its own right in the conceptual model. For example, in the point-of-sale system, there is a UPC. It is typically viewed as just a number. So should it be represented as non-primitive type? We have the following guidelines:

Represent what may initially be considered a primitive type as a non-primitive type if:

- It is composed of separate sections.
  - phone number, name of person
- There are operations usually associated with it, such as parsing or validation.
  - social security number, credit card number
- It has other attributes.
  - promotional price could have a start and end date
- It is a quantity with a unit.
  - payment amount has a unit of currency

Applying these guidelines to the following examples:

- The *upc* should be a non-primitive *UPC* type, because it can be check-sum validated and may have other attributes (such as the manufacturer who assigned it).

- *price* and *amount* attributes should be non-primitive *Quantity* types, because they are quantities in a unit of currency.
- The *address* attribute should be a non-primitive *Address* type because it has separate sections.

These types are pure data values as unique identity is not meaningful, so they may be shown in the attribute box section rather than related with an association line. On the other hand, they are non-primitive types. with their own attributes and associations, it may be interesting to show them as concepts in their own boxes.

There is no correct answer; it depends on how the conceptual model is being used as a tool of communication, and the significance of the concepts in the domain.

This is illustrated in Figure 5.17

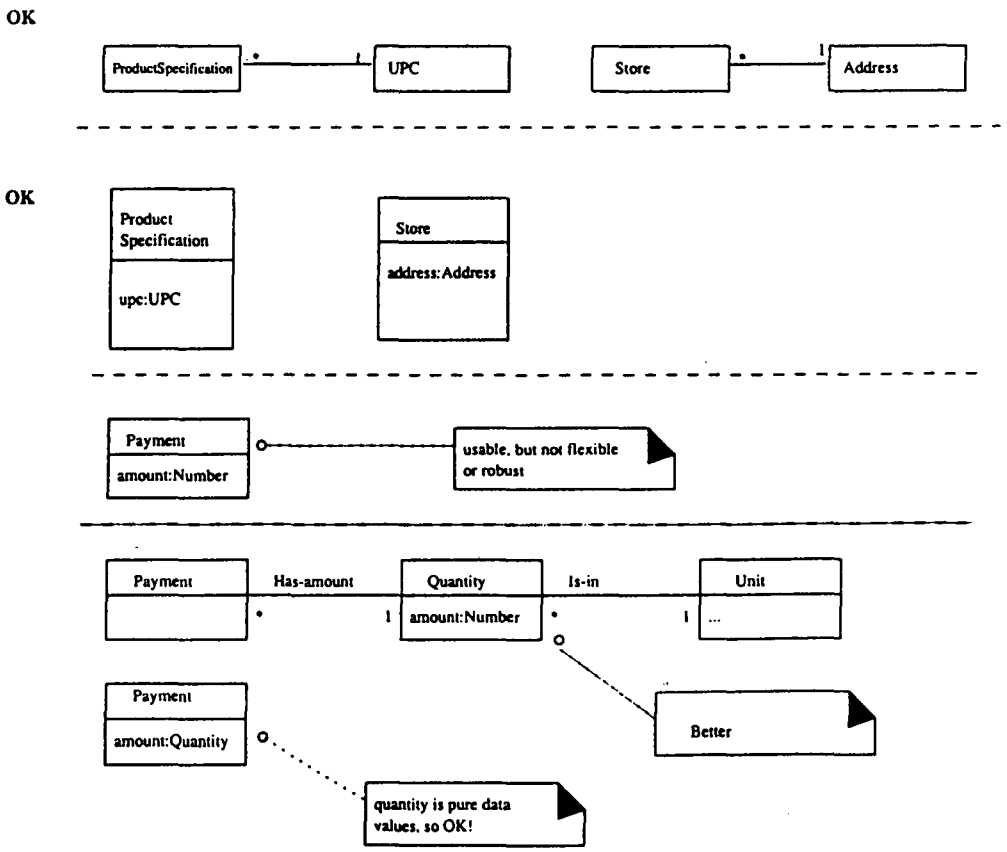


Figure 5.17: Modelling non-primitive attribute types

Where and how to Find Attributes for a class?

- Read the problem description and system functions document, and use cases, find what information or data should be maintained and updated. These also usually occur as nouns or noun phrases. Make them as attribute candidates.
- Use the guidelines discussed in this section to rule out and rule in these candidates.
- Read the simplification and assumptions that are currently made for the development to judge if a candidate is relevant to the current development.
- Assign the identified candidate to classes in the conceptual model which has been created so far, according to characteristics of the objects of a class

5.3.2 Attributes for the Point-of-Sale System

Based the discussions presented in this section, we can clearly add attributes to the concepts in Figure 5.4, by reading

- The system functions
- The the use case **Buy Items wish Cash**
- The simplification, clarification and assumption documents

Figure 5.18

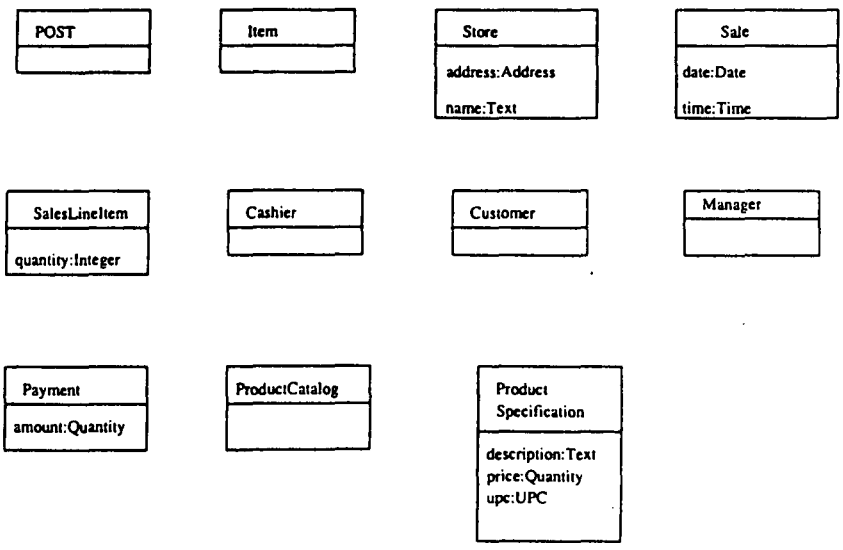


Figure 5.18: Conceptual model showing attributes for the point-of-sale-system

These attribute are only those relevant to the **Buy Items with Cash**. Combine the Figure 5.18 with Figure 5.10, we can obtain the conceptual model in Figure 5.19 for the consideration of the **Buy Items with Cash** use case.

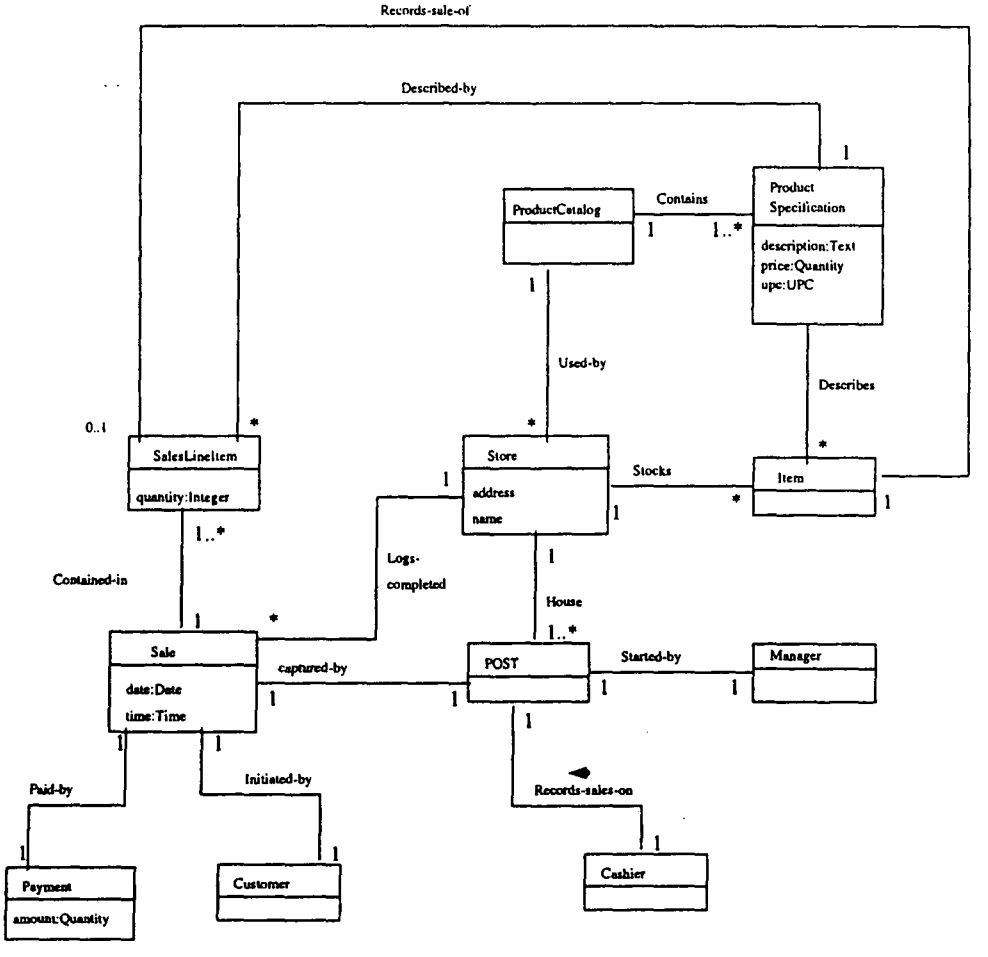


Figure 5.19: A conceptual model for the point-of-sale domain

Such a class diagram is called a *static structure diagram*. It describes the structural relationships holding among the pieces of data manipulated by the system. It describes how the information is parcelled out into objects, how those objects are categorized into classes, and what relationships can hold between objects.

5.4 Steps to Create a Conceptual Model

Apply the following steps to create a conceptual model:

1. List candidate concepts using the Concept Category List and noun phrase identification related to the current requirements under consideration.
2. Draw them in a conceptual model.
3. Add the associations necessary to record relationships for which there is a need to preserve some memory.
4. Add the attributes necessary to fulfill the information requirements.

The conceptual model should be created by investigating the system function, use cases and other initial reports on the domain.

Conceptual models are not models of software designs, such as Java or C++ classes

Therefore a conceptual model may show

- concepts
- associations between concepts
- attributes of concepts.

The following elements are not suitable in a conceptual model:

- Software artifacts, such as a window or a database, unless the domain being modelled is of software concepts, such as a model of a graphical user interface.
- Operations (responsibilities) or methods.

This is illustrated in Figure 5.20

5.5 Recording Terms in the Glossary

A *glossary* is a simple document that defines terms used in the system development. At the very least, a *glossary* or *model dictionary* lists and defines the terms that require clarification in order to improve communication and reduce the risk of misunderstanding.

Consistent meaning and a shared understanding of terms is extremely important during application development, especially when many team members are involved.

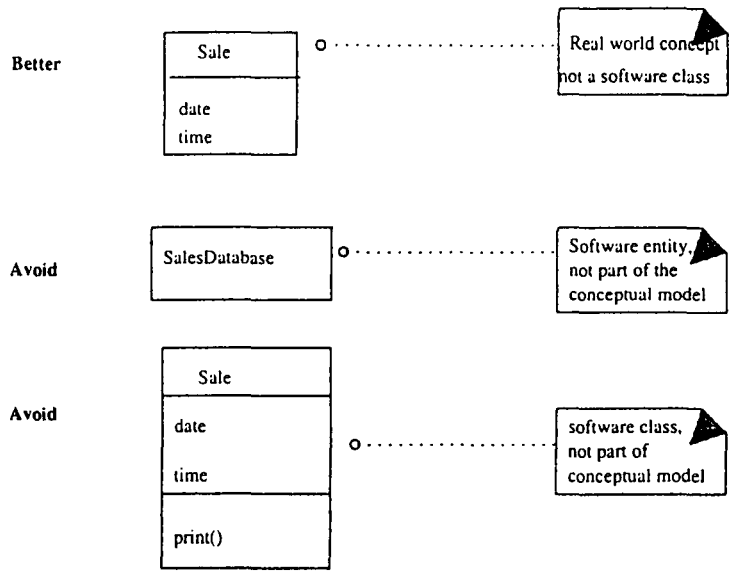


Figure 5.20: A concept does not show software artifacts or classes

The glossary is originally created during the system functions definition, and use case identification, as terms are generated, and is continually refined within each development phase as new term are encountered. It is thus usually made in parallel with the requirements specification, use cases, and conceptual model. Maintaining the glossary is an ongoing activity throughout the project.

There is not official format for a glossary. Here is an sample.

Term	Category	Comments
Buy Items	use case	Description of the process of a customer buying items in a store
ProductSpecification.description:Text	attribute	A short description of an item in a sale, and its associated *ProductSpecification*.
Item	type (class)	An item for sale in a *Store*
Payment	type (class)	A cash payment
Productspecification.price:Quantity	attribute	The price of an item in a sale, and its associated ProductSpecification
SalesLineItem.quantity:Integer	attribute	The quantity of one kind of *Item* bought
Sale	type (class)	A sales transaction
SalesLineItem	type (class)	A line item for a particular item bought within a *Sale*
Store	type (class)	The place where sales of items occur
Sale.total:Quantity	attribute	The grand total of the *Sale*
Payment.amount:Quantity	attribute	The amount of cash tendered, or presented, from the customer for payment
ProductSpecification.upc:UPC	attribute	The universal product code of the *Item.* and it ProductSpecification

5.6 Questions

1. Express in UML that a Student takes up to six modules, where at most 25 students can be enrolled on each Module.
2. Express in UML that a Module is a part of an Honors Course, that a Wheel is a part of a Car, that an Employee is a member of a Team. Discuss the use shared aggregation and composite aggregation.
3. Express in UML the relationship between a person and his/her shirts. What about the person's shoes? Do you think you have exposed a weakness in UML? Why, or why not?
4. Objects have properties (attributes) which are usually static, and values, which are usually dynamic. What are the properties of a 5p coin in you pocket? What values do these properties have – are they dynamic? What about a literature review you are editing in a word processor – what are its properties and values?
5. Consider a car as an object. What are some of its behaviors that are dependent on its state? How about a computer chess game?
6. What is the multiplicity of the association between classes in each of the following pairs?
   (a) the classes *Dessert* and *Recipe* in a cooking tutorial

(b) the classes *Vegetable* and *Nutrient* in a hydroponics farm management system

(c) the classes *Room* and *Window* inan architectural design system

7. Although we said that a conceptual model is not absolutely correct or wrong, but more or less useful, we still wonder what makes a class model good. In general a useful class model should aim at, among the others, the following two objectives:

   - Build, as quickly as possible and cheaply as possible, a system which satisfies the current requirements;
   - Build a system which will be easy to maintain and adapt to future requirements.

   Discuss how these two aims conflict with each other, how they can be met by a class model. What considerations might determine whether an organization considered one more significant than the other?

8. Some people may like to discard a noun phrase identified by the following list of reasons:

   - **redundant**, where the same class is given more than one name.
   - **vague**, where you can't tell unambiguously what is meant by the noun. Obviously you have to clear up the ambiguity before you can tell whether the noun represents a class.
   - **event or an operation**, where the noun refers to something which is done to, by or in the system.
   - **meta-language**, where the noun forms part of the way we define things. We use the nouns *requirements* and *system*, for instance, as part of our language of modeling, rather than to represent objects in the problem domain.
   - **outside the scope of the system**, where the noun is relevant to describing how the system works but does not refer to something inside the system. For example, 'library' in a Library System and names of actors are then often discarded by this rule, when there is no need to model them in the system.
   - **an attribute**, where it is clear that a noun refers to something simple with no interesting behaviour, which is an attribute of another class.

   Is this list of reasons for discarding reasonable? How applicable are these rules to our POST system and the Library System in Question 8 of Chaper 4? Can you think of cases where it might be too selective or too permissive?

9. In general, if you are not sure about whether to keep a class, do you think it is better to keep it in (possibly throwing it out later) or to throw it out (possibly reinstating it later)? Do you have any other alternative?

10. A classic error for people not yet steeped in OO is to invent a class, often called [Something]System, which implements all the system's interesting behavior. It is then easy to drift into a *monolithic* design in which there is just one object that knows and does everything. Can you see why this is bad? Then discuss your understanding about the use of *POST* to represent the *overall system* in our case study.

11. An inherent difficulty in oject classification is that most objects can be classified in a variety of ways, and the choice of classification depends on the goals of the designer. Consider the holdings of the East-West University library first described in Chapter 3 questions. What are some of the ways you might classify these holdings. For what purpose(s) might you use different classifications?

12. In Section 5.4, we said that it is not suitable to show in a conceptual model operations. Do you have any reason to disagree from this principle? When do you think it may be helpful to show operations in the conceptual model?

13. Consider the following game: two players each draw three cards in turn from a pack; the player with the highest total is declared the winner.

    Start the development for a software of this game by drawing a conceptual model using UML notation.

Chapter 6

System Behaviour: System Sequence Diagrams and Operations

Topics of Chapter 6

- Identifying system events and system operations
- Create system sequence diagram for use cases
- Create contracts for system operations

6.1 System Sequence Diagram

This chapter is to identify the operations that the sytem needs to perform and in what order the system need to perform these operations to *carry out* a use case, and the effect of such an operation on the system, i.e. on the objects of the systems.

A use case defines a *class* of conversations between the actors and the system, and and individual conversation of this class is a *realization* of the use case. Obviously, there may be many realizations for a use case. For example, consider the **Buy Items with Cash** use case.

- Different customers buy different items and therefore carry out the use case with different particulars, even though the overall pattern of the interaction remain the same.
- Also, in some occasions, the amount tendered is less than the sale total.

A *scenario* of a use case is a particular instance or realized path through the use case, i.e. a particular realization of the use case.

6.1.1 System input events and system operations

During the interaction in any realization, the actors generates events to a system, requesting the systen to perform some operations in response. *Events generated by actors* are very tightly related to *operations that the system can perform.* This implies that we identify system's operations by identifying events that actors generates.

We first define our terminology:

- A *system input event* is an external input generated by an actor to a system. A system input event initiates a responding operation.
- A *system operation* is an operation that the system executes in response to a system input event. Some system operations also generate *output events* to the actors to prompt the next system event that an actor can perform.
- A *system event* is either an input event or an output event.

Therefore, *a system input event triggers a system operation, and a system operation responses to a system input event.*

Now we can formally define a *scenario* of a use case as a sequence of system events that occur duration a realization of the use case. For example, consider the use case of **Make Phone Calls** for a telephone system, which involves two actors, *Caller* (initiator) and *Callee.* The following sequence of events is a scenario of this use case:

Input Events	Output Events
1. Caller lifts receiver	2. dial tone begins
3. Caller dials(5551234)	4. phone rings to Callee
	5. ringing tone to Caller
6. Callee answers phone	7. phone ring stops
	8. ringing tone stops
	9. phones connected
10. Callee hangs up	11. connection broken
12. Caller hangs up	

The sequence of system events, their order, the actors that interact *directly* with the system, and the system as a *black box* can be shown in a *event trace diagram* (see Figure 6.1). Time proceeds downwards, and the ordering of events should follow their order in the use case.

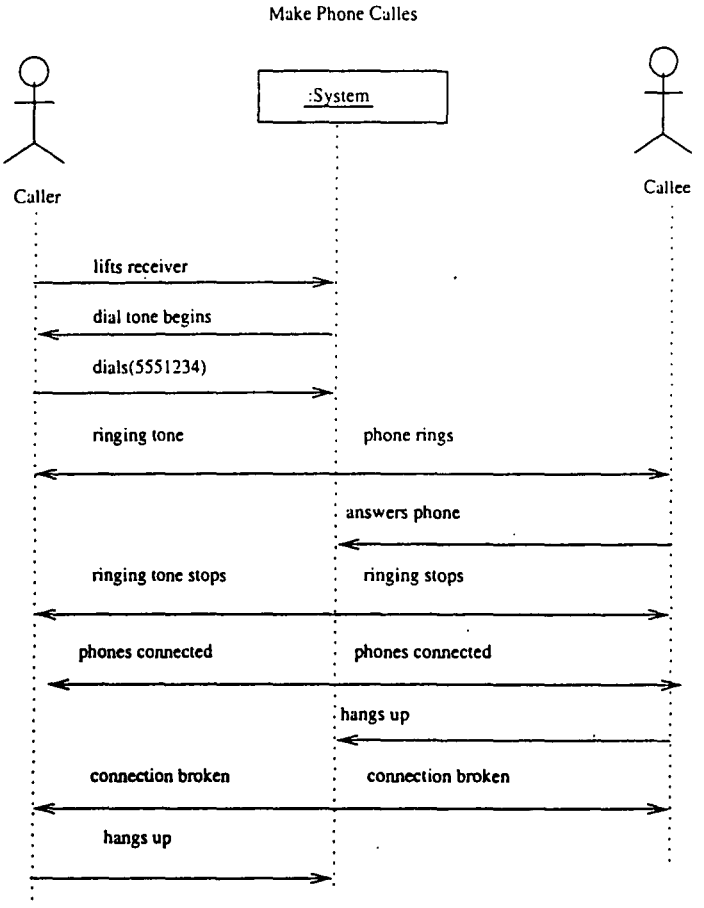


Figure 6.1: Telephone system event trace diagram for the Make Phone Calls

6.1.2 System sequence diagram

At the requirements analysis phase, it is necessary to define the system by treating it as a *black box* so that the behaviour is a description of *what* a system does, without explaining *how* it does it. Therefore, we are mainly interested in identifying and illustrate the system operations that an actor requests of the system. This means, we are interested in finding the system input events by inspecting the use cases and their scenarios.

For example, the typical course of events in the *Make Phone Calls* indicates that the caller and callee generate the system input events that can be denoted as *liftReceiver, dialPhoneNumber, answersPhone, hangsUp*. In general, an event takes parameters. We would like to present them in an unified form: $Event(x_1, ..., x_n)$ to indicate that event E carries n parameter, and $Event()$ to indicate that *Event* does not have parameters.

To identify the system operations only, we also would like to use a trace diagram to show, for a particular

course of events within a use case, the external actors that interact directly with the system, the system (as a black box), and the system input events that the actors generate. And we call such a simplified trace diagram a *system sequence diagram*[1]. A system sequence diagram for the *Make Phone Calls* use case can be illustrated in Figure 6.2.

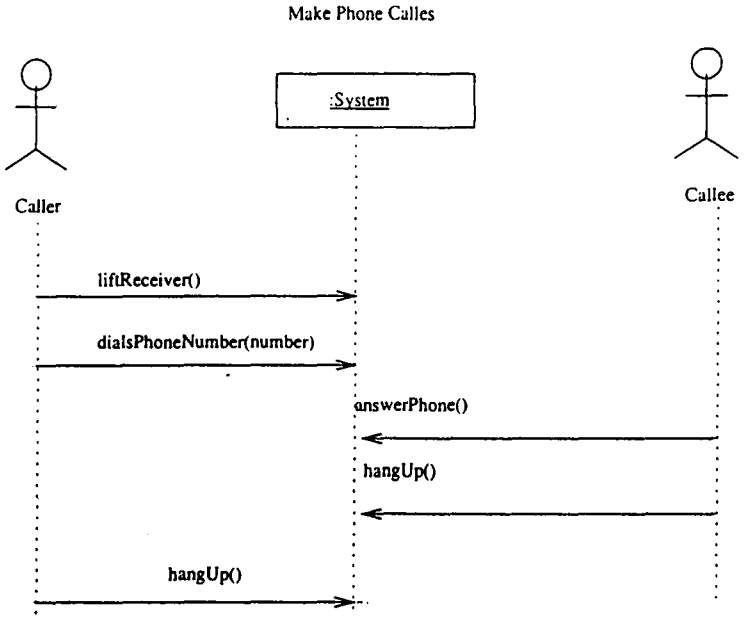


Figure 6.2: Telephone system sequence diagram for the Make Phone Calls

A system sequence diagram should be done for the typical course of events of the use case, and possibly for the most interesting alternative courses.

Consider the *Buy Items with Cash* use case for the POST system. The use case indicates that the cashier is the only actor that directly interacts with the system, and that the cashier generates *enterItem, endSale, and makePayment* system input events. A sequence diagram Figure 6.3 is given for *Buy Items with Cash* use case for the POST system.

Steps in making a sequence diagram

1. Draw a line representing the system as a black box.
2. Identify each actor that directly interacts with the system.
3. Draw a line for each such actor.
4. From the use case typical course of events text, identify the system (external) input events that each actor generates. Illustrate them on the diagram.

[1]Scenario, event trace diagram, and sequence diagram are used inconsistently in books: some use scenario as a synonym for use case, some use event trace diagram in the same way as sequence diagram and *vice versa*.

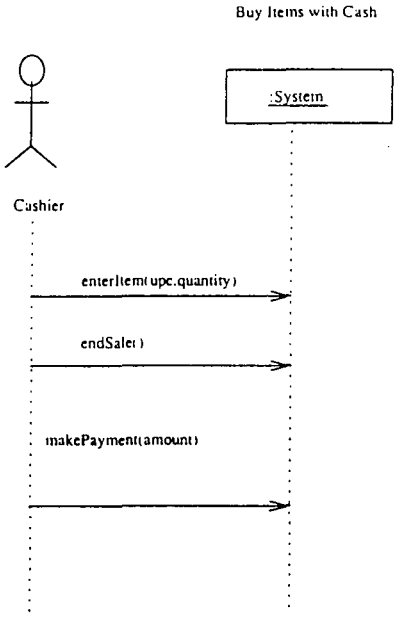


Figure 6.3: A POST system sequence diagram for the *Buy Items with Cash* use case

Both event trace and sequence diagrams are dynamic models which show the *time-dependent behaviour* of the system. They are used for dynamic analysis by looking for events which are externally – visible stimuli and responses. Algorithm execution is not relevant during this analysis if there are no externally-visible manifestations; algorithms are part of detailed design.

Dynamic models are insignificant for a purely static data repository, such as a database. They are important for interactive systems. For most problems, logical correctness depends on the sequence of interacting events, not the exact times when events occur. *Real-time systems*, however, do have specific timing requirements on interactions that must be considered during analysis. We do not address real-time analysis in this course.

6.1.3 Recording system operations

The set of all required system operations is determined by identifying the system input events. Each system input event $Event(x_1, ..., x_n)$ causes the execution of the system operation $Event(x_1, ..., x_n)$. The name of the input event and the name of the operation are identical; the distinction is that the input event is the named stimulus, the operation is the response.

For example, from the inspection of the use case *Buy Items with Cash*, we have identified the system operations:

- $enterItem(upc, quantity)$

- *endSale*()
- *makePayment*(*amount*)

Where should these operations be recorded? The UML includes notation to record operations of a type (or a class), as shown in in Figure 6.4. With this notation, the system operations can be grouped as

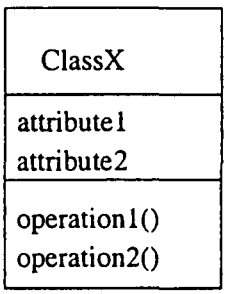


Figure 6.4: The UML notation for operations

operations of a type named *System* as shown in Figure 6.5. The parameter may optionally be ignored.

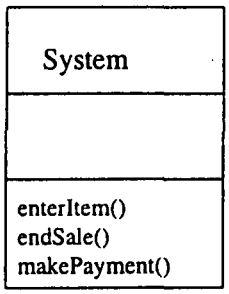


Figure 6.5: System operations recorded in a type named *System*

This scheme also works well when recording the system operations of multiple systems or processes in a distributed application; each system is given a unique name (*System*1, *System*2, ...), and assigned its own systems operations.

When naming system input events and their associated operations, they should be expressed at the level of intent rather than in terms of the physical input medium or interface widget level.

It is also improves clarity to start the name of a system input event with a verb (add..., enter..., end..., make...). For example, we should use *enterItem*() rather than *enterKeyPressed*(), *endSale*() rather than *enterReturnKey*.

6.2 Contracts for System Operations

A system sequence diagram does not describe the effect of the execution of an operation invoked. It is missing the details necessary to understand the system response – the system behaviour.

Part of the understanding of the system behaviour is to understand the system state changes carried out by system operations. A *system state* is a snapshot of the system at a particular moment of time which describes the objects of classes currently existing, current values of attributes of these objects, and current links between the objects at that time. The execution of a system operation changes the system state into another state: old objects may be removed, new objects and links may be created, and values for attributes of objects may be modified. This section introduces how we to write a *contract* for a system operation to describe what state changes that the operation commits to achieve.

The *contract* of an operation is defined mainly in terms of its *pre-conditions* and *post-conditions*:

- *Pre-conditions* are the conditions that the state of the system is assumed to satisfy *before* the execution of the operation.
- *Post-conditions* are the conditions that the system state has to satisfy *when* the execution operation *has finished.*

In formal specification literature, the contract for an operation $Op()$ can be written as a *Hoare-triple*

$$\{Pre\text{-}Conditions\}\, Op()\, \{Post\text{-}Conditions\}$$

roughly meaning that if the execution of $Op()$ starts from a state satisfying *Pre-Condition*, then when the execution has finished, system will be in a state satisfying *Post-Condition*.

Pre-Conditions and Post-condition are *Boolean* statements. A post-condition is a statement of what the world should look like after the execution of an operation. For instance, if we define the operation "getSquare" on a number, the post condition would take the form $result = this * this$, where result is the output and *this* is the object on which the operation is invoked.

A pre-condition is a statement of how we expect the world to be before the operation is executed. We might define a pre-condition for the "getSquare" operation as $this >= 0$. Such a pre-condition says that it is an error to invoke "getSquare" on a negative number and the consequences of is undefined.

Therefore, the pre-condition shall be checked either by the caller to ensure it is true before the execution, and/or checked the object executing the operation to flag an error when it is not true.

As an example, consider the operation $enterItem(upc : UPC, quantity : Integer)$:

- Its pre-condition can be stated as *UPC is known to the system.*

- its post-conditions include:
  - If a new sale, a *Sale* was created (*instance creation*).
  - If a new sale, the new *Sale* was associated with the *POST* (*association formed*).
  - A *SalesLineItem* was created (*instance creation*).
  - The *SalesLineItem.quantity* was set to *quantity* (*attribute modification*).
  - The *SalesLineItem* was associated with a *ProductSpecification*, based on *UPC* match (*association formed*).

It is recommended to express post-conditions in the past tense, in order to emphasise they are declarations about a past state change, rather than about actions. For example, we had better to write

> A *SalesLineItem* **was** created.

rather than

> Create a *SalesLineItem*.

There can be many possible pre- and post-conditions we can declare for an operation. Generating complete and accurate sets of pre- and post-conditions is not likely. For the post-conditions, we are suggested to focus on the following effects of the operation:

- Instances creation and deletion.
- Attributes modification.
- Associations formed and broken.

For pre-conditions, we are suggested to note things that are important to test in software at some point during the execution of the operation.

Contracts of operation are expressed in the context of the conceptual model: only instances of classes in the conceptual model can be created; only associations shown in the conceptual model can be formed.

6.2.1 Documenting Contracts

We are suggested to use the following schema for presenting a contract:

Contract

Name:	Name of operation, and parameters.
Responsibilities:	An informal description of the responsibility this operation must fulfill.
Type:	Name of type (concept, software class, interface).
Cross References:	System function reference numbers, use cases, etc.
Note:	Design notes, algorithms, and so on.
Exceptions:	Exceptional cases.
Output:	Non-UI outputs, such as messages or records that are sent outside of the system.
Pre-conditions:	As defined.
Post-conditions:	As defined

6.2.2 Contracts for some operations in POST system

Contract for *enterItem*

	Contract
Name:	enterItem(upc:UPC, quantity:Integer).
Responsibilities:	Enter (or record) sale of an item and add it to the sale. Display the item description and price.
Type:	System.
Cross References:	System Functions: R1.1, R1.3, R1.9 Use Cases: Buy Items
Note:	Use superfast database access.
Exceptions:	If the UPC is not valid, indicate that it was an error.
Output:	
Pre-conditions:	UPC is known to the system.
Post-conditions:	• If a new sale, a *Sale* was created (*instance creation*). • If a new sale, the new *Sale* was associated with the *POST* (*association formed*). • A *SalesLineItem* was created (*instance creation*). • The *SalesLineItem.quantity* was set to *quantity* (*attribute modification*). • The *SalesLineItem* was associated with the *Sale*. • The *SalesLineItem* was associated with a *ProductSpecification*, based on *UPC* match (*association formed*).

Contract for *endSale*

	Contract
Name:	endsale().
Responsibilities:	Record that it is the end of entry of sale items, and display sale total.
Type:	System.
Cross References:	System Functions: R1.2 Use Cases: Buy Items
Note:	
Exceptions:	If a sale is not underway, indicate that it was an error.
Output:	
Pre-conditions:	UPC is known to the system.
Post-conditions:	• *Sale.isComplete* was set to *true* (attribute modification).

Here, we notice that we have discovered a new attribute for class Sale which is isComplete, so we should add it to the conceptual model that we created in Chapter 3.

Contract for *makePayment*

	Contract
Name:	makePayment(amount: Quantity).
Responsibilities:	Record the payment, calculate balance and print receipt.
Type:	System.
Cross References:	System Functions: R2.1 Use Cases: Buy Items
Note:	
Exceptions:	If sale is not complete, indicate that it was an error. If the amount is less than the sale total, indicate an error.
Output:	
Pre-conditions:	
Post-conditions:	• A *Payment* was created (instance creation).

- *Payment.amountTendered* was set to *amount* (attribute medication).
- The *Payment* was associated with the *Sale* (association formed).
- The *Sale* was associated with the *Store*, to add it to the historical log of completed sales (association formed).

Contract for startUp

It is easy to identify a *StartUp* use case for the POST system, and thus identify a *startUp*() operation. And in fact, every system should have such an operation. However, although the *StartUp* use case and the *startUp*() operation are always carried out first when the system start to work, they can be finalized only after it is clear about what are needed for carrying out the other use cases and operations. The contract of *startUp*() for the POST system can be given as follows.

	Contract
Name:	startUp().
Responsibilities:	Initialise the system
Type:	System.
Cross References:	
Note:	
Exceptions:	
Output:	
Pre-conditions:	
Post-conditions:	• A *Sore, POST, ProductCatalog and ProductSpecification* were created (instance creation). • *ProductCatalog* was associated with *ProductSpecification* (association formed). • *Store* was associated with *ProductCatalog* (association formed). • *Store* was associated with *POST* (association formed). • *POST* was associated with *ProductCatalog* (association formed)

6.2.3 How to make a contract

Apply the following advice to create contract:

1. Identify the system operations from the system sequence diagram.
2. For each system operation, construct a contract.
3. Start by writing the *Responsibilities* section, informally describing the purpose of the operation.
4. Then complete the *Post-conditions* section, declaratively describing the state changes that occur to objects in the conceptual model.
5. To describe the post-conditions, use the following categories:
    - Instances creation and deletion.
    - Attributes modification.
    - Associations formed and broken.

6.2.4 Contracts and other Documents

- Use cases suggest the system input events and system sequence diagram.
- The system operations are then identified from system sequence diagrams.
- The effect of the system operation is described in contract within the context of the conceptual model.

6.3 Analysis Phase Conclusion

The analysis phase of development emphasises on understanding of the requirements, concepts, and operations related to the system. Investigation and analysis are often characterised as focusing on questions of *what* – what are the processes, concepts, associations, attributes, operations.

In Chapter 4 and Chapter 5, we have explored the following minimal but useful set of artifacts that can be used to capture the results of an investigation and analysis:

Analysis Artifact	Questions Answered
Use Cases	What are the domain processes?
Conceptual Model	What are the concepts, associations and attributes?
System Sequence Diagrams	What are the system input events and operations?
Contracts	What do the system operations do?

Questions

1. Discuss how post conditions of an operation can be generated from the use case of the operation and the conceptual model. What are the relationships among the three main kinds of effects of an operation described in the post-conditions?

2. Discuss the diferences and relationships between contracts of operations here and the VDM-style specification of pre- and post conditions of an operation that you have learnt from other modules.

3. We discussed the use cases, conceptual model, system sequence diagrams, and contracts regarding to the whole system to be developed. Discuss the use of the techniques of use cases, conceptual model, system sequence diagrams, and contracts in a component-based development.

Chapter 7

Design Phase: Collaboration Diagrams

Topics of Chapter 7

- Notion of collaboration diagrams
- UML notation for collaboration diagram
- The nature of the design phase
- Patterns for assigning responsibilities to objects
- Use Patterns to create collaboration diagrams

A contract for system operations describes *what* the system operation does. but it does not show a solution of *how* software objects are going work collectively to fulfill the contract of the operation. The later is specified by an *interaction diagrams* in UML. Therefore, a major task of the design phase is to create the interaction diagrams for the system operations.

The UML defines two kinds of interaction diagrams, either of which can be used to express similar or identical messages interactions[1]:

1. collaboration diagrams
2. object sequence diagrams

[1]Given an underlying class model, some CASE tools can generate one from another.

7.1 Object Sequence Diagrams

Sequence diagrams illustrate interactions in a kind of fence format as shown in Figure 7.1

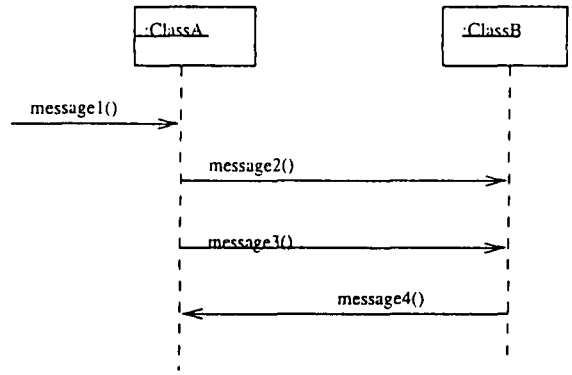


Figure 7.1: Object Sequence Diagram

Notice here the differences between a system sequence diagram and an object diagram include:

- A system sequence diagram illustrates the interaction between the whole system and external actors, while an object sequence diagram shows the interactions between objects of the system.
- A system sequence diagram shows only system's external events and thus identifies system operations, while an object sequence diagram identifies operations of objects.
- System sequence diagrams are created during the analysis phase, while object sequence diagrams are models created and used in the design phase.

If we treat an object as a subsystem, and other objects as external actors, then an object diagram is a sequence diagram of this sub-system.

7.2 Collaboration Diagrams

A *collaboration diagram* is a graph (as shown in Figure 7.2) showing a number of objects and the links between them, which in addition shows the messages that are passed from one object to another.

As both collaboration diagrams and object sequence diagrams can express similar constructs, different people may prefer one to another. We shall mainly discuss and use collaboration diagrams.

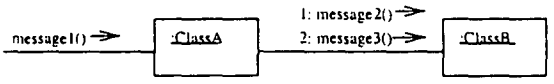


Figure 7.2: Collaboration Diagram

7.2.1 Examples of collaboration diagrams

Suppose after receiving the cash payment from the Customer, the Cashier records the amount of the cash received by sending the message *makePayment(cashTendered)* to the *POST*. The *POST* carries out this request by sending the message *makePayment(cashTendered)* to the (current) *Sale*, which carries out the task by creating a *Payment*. This interaction is shown by the collaboration diagram in Figure 7.3. From a programming point of view, Figure 7.3 represents the following pseudo program:

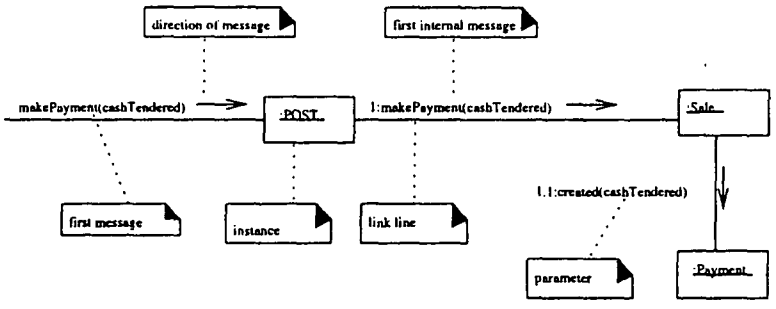


Figure 7.3: An example of a collaboration diagram

```
:POST accepts a call of it method makePayment()
                    (from an Interface Object);

:POST calls for the method makePayment() of :Sale;

:Sale calls the constructor of class Payment to create a new :Payment
```

Therefore, a collaboration diagram for an operation represents an algorithm that implement the operation.

We now explain some basic UML notation used in collaboration diagrams:

1. *Instances:* An instance (or object) uses the same graphic symbol as its type (or class), but the designator string is underlined. Additionally, the class name should always be preceded by a colon. An value for an attribute of an object is represented as $attributeName = attributeValue$.

2. *Links:* A *link* between two objects is a connection path between the two objects; it indicates that some form of navigation and visibility between the instances is possible. More formally, a link

is an instance of an association. Viewing two instances in a client/server relationship, a path of navigation from the client to the server means that messages may be sent from the client to the server.

3. *Messages:* Messages between objects are represented via a labelled arrow on a link line. Any number of messages may be sent along this link. A *sequence number* is added to show the sequential order in which the message are sent. This is shown in Figure 7.4. A message can be

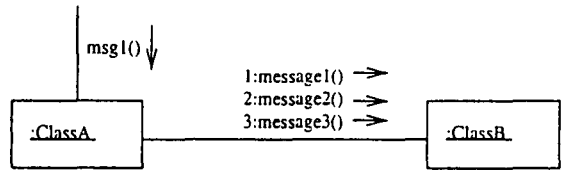


Figure 7.4: Messages

passed between two objects only when there is an association between the classes of objects, such as *Captured-by* association between *Sale* and *POST*, **and** the two objects are linked by this association.

4. *Parameters:* Parameters of a message may be shown within the parentheses following the message name. The type of the parameter may optionally be shown, such as *makePayment*(*cashTendered* : *Quantity*).

7.2.2 More notational issues

Representing a return value Some message sent to an object requires a value to be returned to the sending object. A return value may be shown by preceding the message with a return value variable name and an assignment operator (':='). The type of the return value may operationally be shown. This is illustrated in Figure 7.5. The standard syntax for messages is:

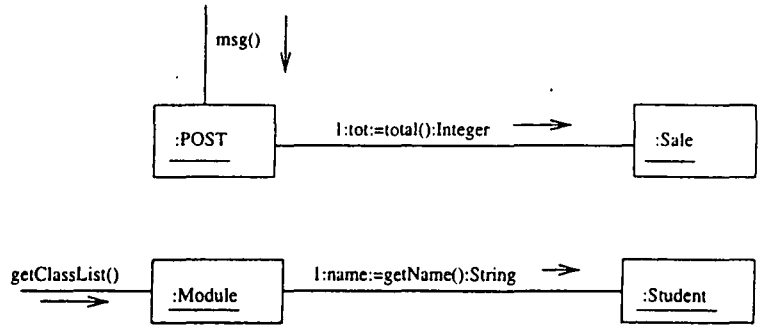


Figure 7.5: Return values

```
return := message(parameter : parameterType) : returnType
```

Representing iteration An object may repeatedly send a message to another object a number of times. This is indicated by prefixing the message with a start ('*'). This is shown in Figure 7.6.

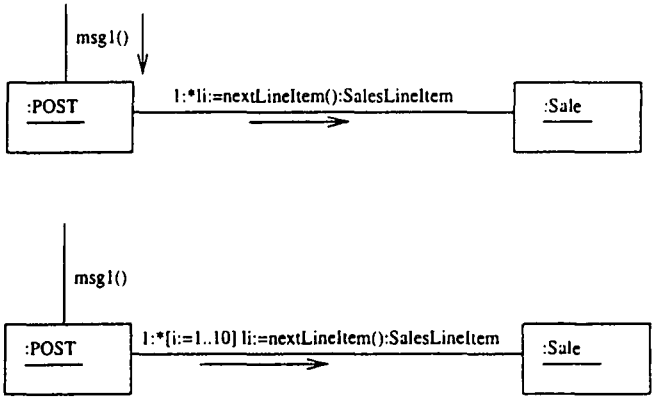


Figure 7.6: Iteration with and without recurrence values

For any other kind of iterations we may combine the asterisk with an *iteration clause* in square brackets, such as

- *[$x < 10$] — the message will be sent repeatedly, **until** x becomes less than 10,
- *[item not found] — the message will be sent repeatedly, **until** the *item* is found.

To express more than one message happening within the same iteration clause (for example, a set of messages within a *for loop*), repeat the iteration clause on each message. This is illustrated in Figure 7.7.

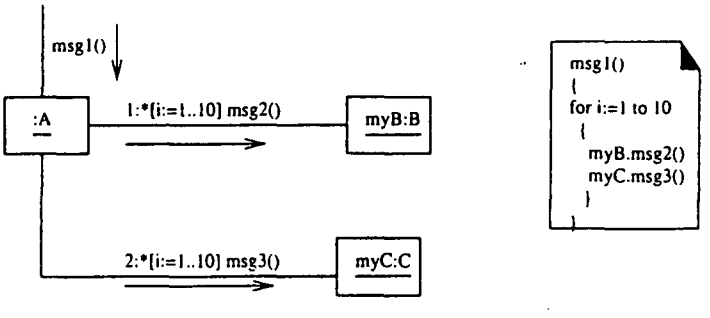


Figure 7.7: Multiple message within the same iteration clause

Representing creation of instances The UML creation message is *create* which is independent of programming languages. shown being sent to the instance being created. Optionally, the newly created instance may include a << *new* >> symbol (See Figure 7.8).

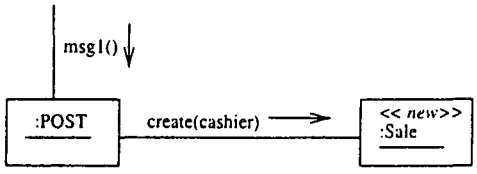


Figure 7.8: Instance Creation

A *create* message can optionally take parameters when some attributes of the object to be created need to be set an initial value.

Representing message number sequencing The order of messages is illustrated with *sequence numbers*, as shown in Figure 7.9. The numbering scheme is:

1. The first message is not numbered. Thus, *msg*1() is unnumbered.
2. The order and nesting of subsequent messages is shown with a legal numbering scheme in which nested messages have appended to them a number. Nesting is denoted by prepending the incoming message number to the outgoing message number.

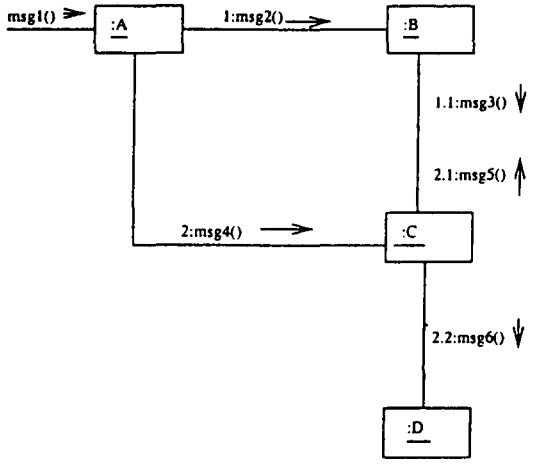


Figure 7.9: Sequence numbering

Representing conditional messages Sometimes, a message may be *guarded* and can be sent from one object to another only when a condition holds. This is illustrated in Figure 7.10.

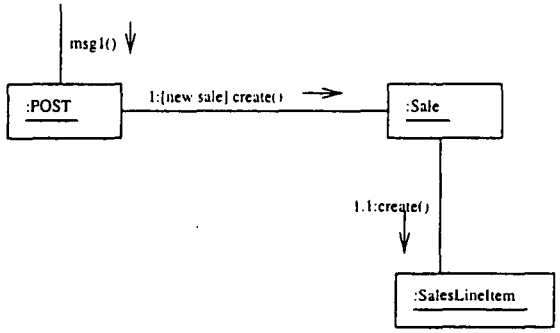


Figure 7.10: Conditional message

At a point during the execution of an object, a choice of several messages, guarded by different conditions, will be sent. In a sequential system, an object can send one message at a time and thus these conditions must be mutually exclusive. When we have mutually exclusive conditional messages, it is necessary to modify the sequence expressions with a conditional path letter. This is illustrated in Figure 7.11. Note that

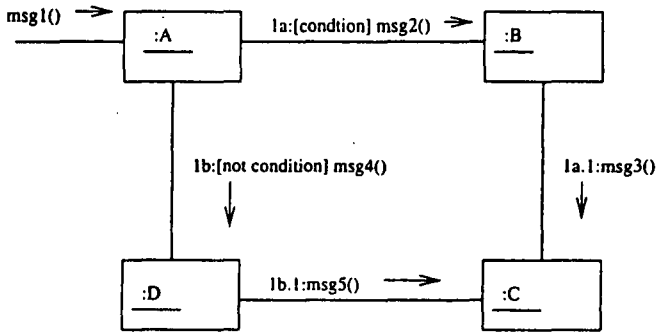


Figure 7.11: Mutually exclusive messages

- Either $1a$ or $1b$ could execute after $msg1()$, depending on the conditions.
- The sequence number for both is 1, and a and b represent the two paths.
- This can be generalised to any number of mutually exclusive Conditional Messages.

Representing multiobjects We call a logical set of instances/objects as a *multiobject*. In general, a multiobject is an instance of a *container class* each instance of which is a set of instances of a given class (or type).

For example, *SetOfSales* is a class each instance of which is a set of sales. Many implementation data structures are examples of container classes: Array, List, Queue, Stack, Set, Tree, etc.

Therefore, each multiobject is usually implemented as a group of instances stored in a *container* or a *collection* object, such as C++ STL *vector*, Java vector or a Smalltalk *OrderedCollection*.

In a collaboration diagram, a multiobject represents a set of of objects at the "many" end of an association. In UML, a multiobject is represented as a stack icon as illustrated in Figure 7.12.

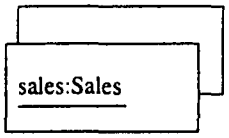


Figure 7.12: A multiobject

We represent a message sent to a multiobject in the standard way with the standard meaning, see Figure 7.13.

Notice that a message to a multiobject is *not* broadcast to each element in the multiobject; it is a message to the multiobject which is an object itself.

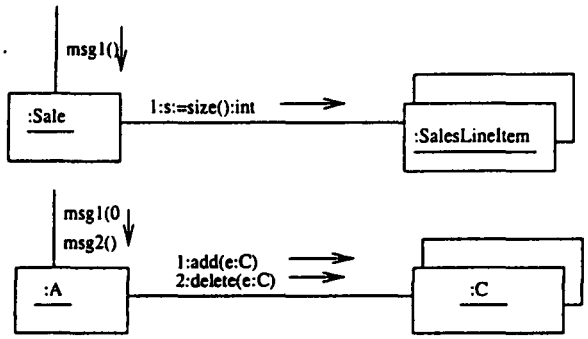


Figure 7.13: Message to multiobject

To perform an Operation on each object in a multiobject requires two messages: an iteration to the multiobject to extract links to the individual objects, then a message sent to each individual object using the (temporary) link. This is illustrated in Figure 7.14

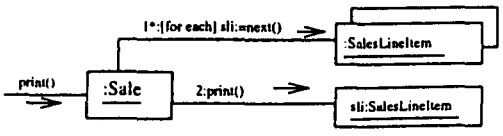


Figure 7.14: Performing an operation on each object of a multiobject

Typically the selection message in each iteration returns a reference to an individual object, to which the original sender sends a message.

Figure 7.14 indicates that a *Sale* object has received a message to print itself, which then requires each of the *SalesLineItem* objects contained in the *Sale* object to print itself.

7.3 Creating Collaboration Diagrams

7.3.1 Overview of Design Phase

During the design phase, there is a shift in emphasis from application domain concepts toward software objects, and from *what* toward *how*. The objects discovered during analysis serve as the skeleton of the design, but the designer must choose among different ways to implement them. In particular, the system operations identified during the analysis must be assigned to individual objects or classes, these operations must be expressed as algorithms. Also, complex operations must be decomposed into simpler internal operations which are assigned to (or carried out by) further objects.

The major task in the design is to create the collaboration diagrams for the system operations identified in the requirement analysis phase. The most important and difficult part in the generation of collaboration diagrams is the *assignment of responsibilities to objects*. Therefore, the rest of this chapter will mainly discuss the general principles for responsibility assignment, which are structured in a format called *patterns*.

The creation of collaboration diagrams is dependent upon the prior creation of the following artifacts:

- Conceptual model: from this, the designer may choose to define software classes corresponding to concepts. Objects of these classes participate in interactions illustrated in the interaction diagrams.
- System operations contracts: from these, the designer identifies the responsibilities and post-conditions that the interaction diagrams must fulfill.
- Essential (or real) use cases: from these, the designer may glean information about what tasks the interaction diagrams fulfill, in addition to what is in the contracts (remember that contracts do not say very much about the *UI-outputs*).

7.3.2 Real Use Cases

As we briefly said in Chapter 4, a *real use case* is the real or actual design of the use case in terms of concrete input and output technology and its overall implementation. For example, if a graphical user interface is involved, the real use case will include diagrams of the windows involved, and discussion of the low-level interaction with interface widgets.

For example, if we are to use window widgets in the POST system, we may describe the real use case for the **Buy Items with Cash** use case as follows:

Use case:	**Buy Items with Cash**
Actors:	Customer (initiator), Cashier.
Purpose:	Capture a sale and its cash payment.
Overview:	A Customer arrives at a checkout with items to purchase. The Cashier records the purchase items and collects a cash payment. On completion, the Customer leaves with the items.
Type	Real
Cross References:	*Functions:* R1.1, R1.2, R1.3, R1.7, R1.9, R2.1.

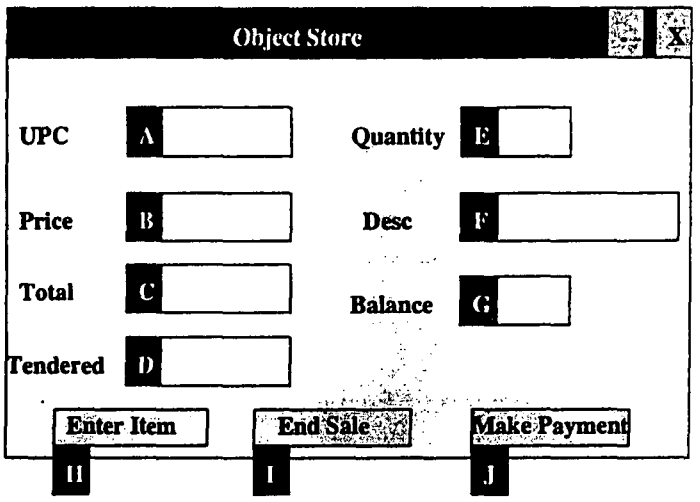


Figure 7.15: Window-1

Typical Course of Events			
	Actor Action		System Response
1.	This use case begins when a Customer arrives at a POST checkout with items to purchase.		
2.	For each item, the Cashier types in the UPC in A of Window-1 (shown in Figure 7.15). If there is more than one of an item, the quantity may optionally be entered in E. They press H after each item entry.	3.	Adds the item information to the running sales transaction. The description and price of the current item are displayed in B and F of Window-1.
4.	On completion of the item entry, the Cashier indicates to the POST that item entry is completed by pressing widget I.	5	Calculates and presents the sale total and display it in C.
6.	.....		

7.3.3 GRASP: Patterns for Assigning Responsibilities

Responsibilities

A *responsibility* is a contract or obligation of an object. Responsibilities are related to the obligations of objects in term of their behaviour.

There are in general two types of responsibilities:

1. **Doing** responsibilities: these are about the actions that an object can perform including
   - doing something itself
   - initiating an action or operation in other objects
   - controlling and coordinating activities in other objects
2. **Knowing** responsibilities: these are about the knowledge an object maintains:
   - knowing about private encapsulated data
   - knowing about related objects
   - Knowing about things it can derive or calculate

We notice that objects support the concept of *information hiding*, i.e. *data encapsulation*. All information in an object-oriented system is stored in its objects and can only be manipulated when the objects are ordered to perform actions. In order to use an object, we only need to know the interface of the object.

The general steps to create collaboration diagrams is described as follows. Start with the responsibilities which are identified from the use cases, conceptual model. and system operations' contracts. Assign these responsibilities to objects, then decide what the objects needs to do to fulfill these responsibilities in order to identify further responsibilities which are again assigned to objects. Repeat these steps until the identified responsibilities are fulfilled and a collaboration diagram is completed.

For example, we may assign the responsibility for print a *Sale* to the *Sale* instance (a doing responsibility), Or we assign the responsibility of knowing the date of a *Sale* to the *Sale* instance itself (a knowing responsibility). Usually, knowing responsibilities are often inferable from the conceptual model, because of the attributes and associations it illustrates.

Responsibilities of an object are implemented by using *methods* of the object which either act alone or collaborate with other methods and objects. For example, the *Sale* class might define a method that performing printing a *Sale* instance; say a method named *print*. To fulfill that responsibility, the *Sale* instance may have to collaborate with other objects, such as sending a message to *SalesLineItem* objects asking them to print themselves.

Within UML, responsibilities are assigned to objects when creation a collaboration diagram, and the collaboration diagram represents both of the assignment of responsibilities to objects and the collaboration between objects for their fulfilment.

For example, Figure 7.14 indicates that

1. *Sale* objects have been given a responsibility to print themselves, which is invoked with a *print* message.
2. The fulfilment of this responsibility requires collaboration with *SalesLineItem* objects asking them to print.

7.3.4 GRASP: Patterns of General Principles in Assigning Responsibilities

Experienced OO developers build up both general principles and idiomatic solutions called *patterns* that guide them in the creation of software. Most simply, a *pattern* is a named problem/solution pair that can be applied to new context, with advice on how to apply it in novel situations.

We shall present a pattern in the following format:

Pattern Name: the name given to the pattern

Solution: description of the solution of the problem

Problem: description of the problem that the pattern solves

Now we are ready to explore the GRASP patterns: General Responsibility Assignment Software Patterns. They include the following five pattern: *Expert, Creator, Low Coupling, High Cohesion, and Controller.*

Expert

Pattern Name: Expert

Solution: **Assign a responsibility to the information expert – the class that has *information* necessary to fulfill the responsibility.**

Problem: What is the most basic principle by which responsibilities are assigned in OOD?

Example

In the POST application, some class needs to know the grand total of a sale. When we assign responsibilities, we had better to state the responsibility *clearly*:

> Who should be responsible for knowing the grand total of the sale?

By Expert, we should look for that class of objects that has the information

- all the *SalsLineItem* instances of the *Sale* instance, and
- the sum of their subtotal.

that is needed to determine the total. Consider the partial conceptual model in Figure 7.16.

Only the *Sale* knows this this information; therefore, by Expert, *Sale* is the correct class of objects for this responsibility. Now we can start working on the collaboration diagram related to the assignment of responsibility for determining the grand total to *Sale*, by drawing Figure 7.17.

After assigning the responsibility of getting the grand total to *Sale*, it needs to get the subtotal of each line item. The information needed to determine the line item subtotal is *SalesLineItem.quantity* and *ProductSpecification.price*. As the *SalesLineItem* knows its *quantity* and its associated *ProductSpecificati* by Expert, *SalesLineItem* should *be responsible for* returning its subtotal to the *Sale*. In terms of a collaboration diagram, the *Sale* needs to send *subtotal* message to each of the *SalesLineItem* and sum the results; the design is shown in Figure 7.18.

In order to fulfill the responsibility of *knowing and returning* its subtotal, a *SalesLineItem* needs to know where to get the product price. The *ProductSpecification* is an information expert on answering its price; therefore, a message must be sent to it asking for its price. The design is shown in Figure 7.19, which is the complete collaboration diagram representing the design for operation *total*().

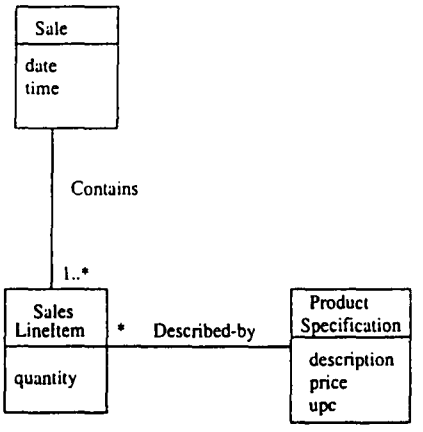


Figure 7.16: Associations of the Sale

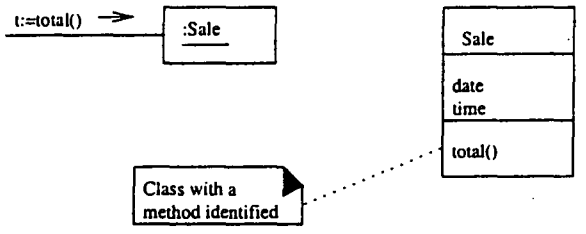


Figure 7.17: Partial collaboration diagram

Remarks on the Expert Pattern The Expert pattern is used more than any other pattern in the assignment of responsibilities; it is a basic guiding principle used again and again in OOD. It expresses the common intuition that objects do things related to the information they have.

The fulfilment of a responsibility often requires information spread across different classes of objects. This means that there are often many "partial" experts who will collaborate in the task. Whenever information is spread across different objects, the objects will need to interact via message passing in order to exchange information and to share the work. Information which is spread across different objects is exchanged via the links between associated objects.

Expert leads to designs where a software object does those operations which are normally done to the real-world (or domain) object it represents; this is called the " DO it Myself" strategy. Fundamentally, objects do things related to information they know.

The use of the Expert pattern also *maintain encapsulation*, since objects use their own information to fulfill responsibilities. This also implies **low coupling**, which means no extra links needed to be formed apart from those which have to be there. Low coupling implies high independency of objects that leads to more robust and maintainable systems.

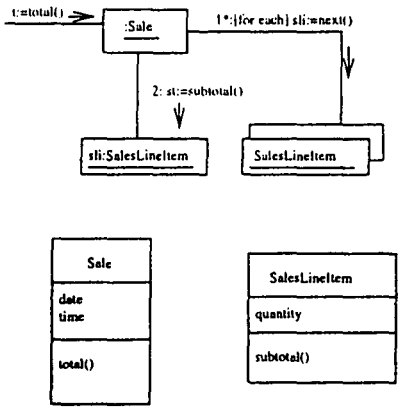


Figure 7.18: Getting the subtotal

Creator

The creation of objects is one of the most common activities in an OO system. Consequently, it is important to have a general principle for the assignment of creation responsibilities.

Pattern Name: Creator

Solution: **Assign class B the responsibility to create an instance of a class A (B is a *creator* of A objects) if one of the following is true:**

- **B *aggregates* A objects.**
- **B *contains* A objects.**
- **B *records* instances of A objects.**
- **B *closely uses* A objects.**
- **B *has the initialising data* that will be passed to A when it is created (thus B is an expert with respect to creating A objects.**

Problem: What should be responsible for creating a new instance of some class?

Example

In the POST application, consider the part of the conceptual model shown in Figure 7.20.

Since a *Sale* contains (in fact, aggregates) many *SalesLineItem* objects, the Creator pattern suggests *Sale* is a good candidate to be responsible for creating *SalesLineItem* objects. This leads to a design of object interactions in a collaboration diagram, as shown in Figure 7.21.

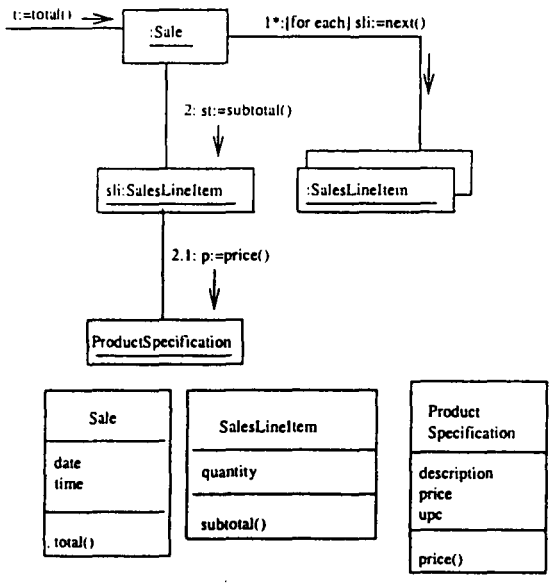


Figure 7.19: Getting the price of a product

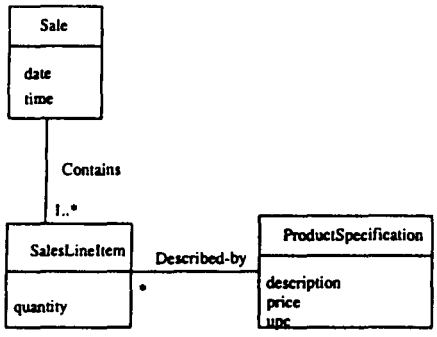


Figure 7.20: Partial conceptual model

The assignment of this responsibility requires that a *makeLineItem* method be defined in *Sale*. In general, a method has to be defined in a class to fulfill each responsibility assigned to that class.

Sometimes a creator is found by looking for the class that has the initialising data that will be passed in during the creation of an object. This is actually an example of the Expert pattern. For example, assume that a *Payment* instance needs to be initialised, when created, with the *Sale* total. Since *Sale* knows the total, *Sale* is a candidate creator of the *Payment* object.

Low Coupling

Coupling is a measure of how strongly one class is connected to, has knowledge of, or relies upon other

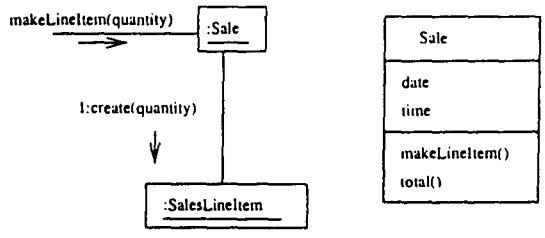


Figure 7.21: Create a SalesLineItem

classes. A class with low (or weak) coupling is not dependent on too many other classes.

A class with high (or strong) coupling relies upon many other classes. Such classes are undesirable; they suffer from the following problems:

- Changes in related classes force local changes, i.e. changes in this class.
- Changes in this class force the changes in many other related classes.
- Harder to understand in isolation.
- Harder to reuse as its reuse requires the use of the related class.

When we assign a responsibility to a class, we would like to assign responsibilities in a way so that coupling between classes remains low. This is captured by the **Low Coupling Pattern** described below.

Pattern Name: Low Coupling

Solution: **Assign a responsibility so that coupling remains low**

Problem: How to support low dependency an increased reuse?

Example

Consider the classes *Payment*, *POST* and *Sale* in the POST system. Assume we need to create a *Payment* instance and associate it with the *Sale* (see the contract for *makePayment* given in Chapter6). Since *POST "records" a Payment*, the creator pattern suggests *POST* as a candidate for creating the *Payment*. This would lead to a design shown in Figure 7.22.

This assignment of responsibilities *couples* the *POST* class to the knowledge of the *Payment* class. An alternative assignment of the responsibility to creation of the *Payment* and to associate it with the *Sale* allows us to do so and is shown in Figure 7.23.

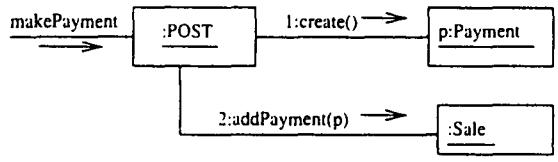


Figure 7.22: POST creates Payment

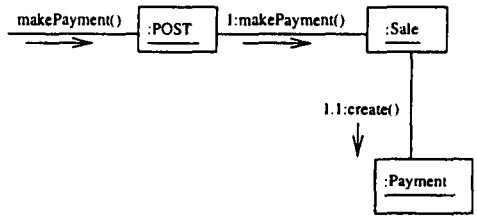


Figure 7.23: Sale creates Payment

Assume that *Sale* will be eventually coupled with *Payment*. The second design is preferable because it does not need an extra link formed between *POST* and *Payment*. This assignment of responsibility is also justifiable as it is reasonable to think that a *Sale* closely uses a *Payment*.

In object-oriented programming languages such as C++, Java, and Smalltalk, common forms of coupling from *ClassX* to *ClassY* include:

- *ClassX* has an attribute that refers to a *ClassY* instance, or *ClassY* itself.
- *ClassX* has a method which references an instance of *ClassY*, or *ClassY* itself, by any means. These typically include a parameter or local variable of type *ClassY*, or the object returned from a message being an instance of *ClassY*.
- *ClassX* is a direct or an indirect subclass of *ClassY*.

Coupling may not be that important if reuse is not a goal. And there is not absolute measure of when coupling is too high. While high coupling makes it difficult to reuse components of the system, a system in which there is very little or no coupling between classes is rare and not interesting.

High Cohesion

Cohesion is a measure of how strongly related and focused the responsibilities of a class are. A class with high cohesion has highly related functional responsibilities, and does not do a tremendous amount of work. Such classes has a small number of methods with simple but highly related functionality.

A class with low cohesion is undesirable as it suffers from the following problems:

- hard to comprehend;
- hard to reuse;
- hard to maintain;

One important nature of a good OOD is to assign responsibilities to classes that are naturally and strongly related to the responsibilities, and every class has something to do but does not have too much to do. It is often the case that a class takes on a responsibility and delegates parts of the work to fulfill the responsibility to other classes. This nature is captured by the high cohesion pattern.

Pattern Name: High Cohesion

Solution: **Assign a responsibility so that cohesion remains high**

Problem: How to keep complexity manageable?

Example

The same example used for Low Coupling can be used for High Cohesion. Assume we have a need to create a *Payment* instance and associate it with the *Sale* (see the contract for *makePayment* given in Chapter 5). Since *POST "records" a Payment*, the creator pattern suggests *POST* as a candidate for creating the *Payment*. This is the design shown in Figure 7.22. However, this design places the responsibility for making a payment in the *POST*. The *POST* takes on part of the responsibility for carrying out the *makePayment* system operation. Imagine that there were fifty system operations, all received by *POST*. If it did the work related to each, it would become "bloated" incohesive object.

In contrast, as shown in Figure 7.23, the second design delegates the payment creation responsibility to the *Sale*, which support higher cohesion in the POST. Since the second design support both high cohesion and low coupling, it is desirable.

The benefits from the use of the High Cohesion Pattern include:

- Clarity and ease of comprehension of the design is increased.
- Maintenance and enhancements are simplified.
- Low coupling is often supported.
- Supports reuse.

Controller

Controller pattern deals with the identified system operations.

Controller Pattern.

Pattern Name : Controller

Solution: **Assign the responsibility for handling a system (input) event to a class representing one of the following choices:**

- **Represents the "overall system" (*facade controller*).**
- **Represents the overall business or organisation (*facade controller*).**
- **Represents something in the real-world that is active (for example, the role of a person) that might be involved in the task (*role controller*).**
- **Represents an artificial handler of all system (input) events of a use case, usually named "< *UseCaseName* > *Handler*" (*use-case controller*).**

Problem: Who should be responsible for handling a a system input event?

This pattern also suggests to *use the same controller class for all system input events in the same use case.* It also implies that a controller is a **non-user** interface object responsible handling a system input event, and that controller defines the method for the system operation corresponding to the system input event. An interface object may accept a system input event, but it does not carry out any of the responsibilities (or any part of the work) of the system operation rather than simply passing (or delegating) it onto the controller.

Example

In the POST application, we have identified several system operations, including *enterItem*(), *endSale*(), and *makePayment*().

During system behaviour analysis (see Chapter 6), system operations are *recorded* to the class *System* as shown in Figure 7.24

However, this does *not* mean that a class named *System* fulfils them during the design. Rather, during design, a Controller class is assigned the responsibility for system operations.

By the Controller pattern, here are our choices;

1. *POST:* represents the overall system.

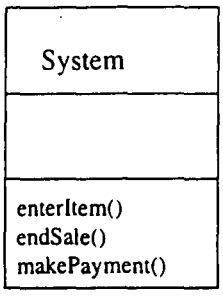


Figure 7.24: System operations recorded in a type named *System*

2. *Store:* represents the overall business or organisation.
3. *Cashier:* represents something in the real-world (such as the role of a person) that is active and might be involved in the task.
4. *BuyItemsHandler:* represents an artificial handler of all system operations of a use case.

In terms of collaboration diagrams, this means that one of the diagrams in Figure 7.25 will be used.

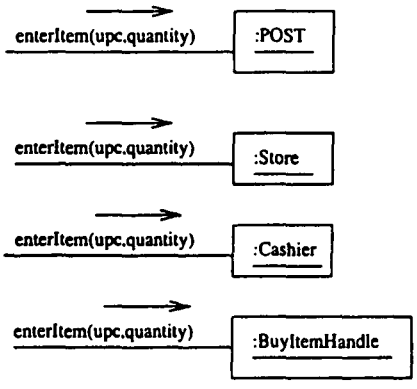


Figure 7.25: Choices of Controllers

The final decision on which of these four is the most appropriate controller is influenced by other factors, such as cohesion and coupling.

In the design, the system operations identified during the system behaviour analysis are assigned to one or more controller classes, such as *POST* (see Figure 7.26).

Discussion about Controller

- *Controller and Use Cases:* The same controller class should be used for all the system input events of one use case so that it is possible to maintain information about the state of the use case,

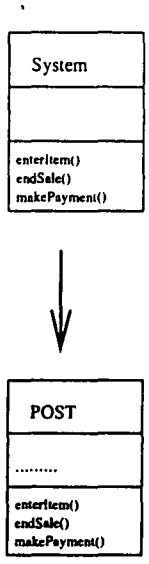


Figure 7.26: Allocation of system operation to a controller class

for example, to identify out-of-sequence system input events (e.g. a *makePayment* operation before an *endSale* operation). Different controllers may be used for different use cases.

- *Use-Case Controller:* If use-case controller are used, then there is a different controller for each use case. Such a controller is not a domain object; it is an artificial construct to support the system.

- *Controller and the other Patterns:* Poorly designed controllers classes will have low cohesion – unfocused and handling too many loosely responsibilities. Examples of poorly designed controllers include:

  - There is only a *single* controller class receiving *all* system input events, and there are many of them.

  - The controller itself performs many of the tasks necessary to fulfill the system operations, without delegating the work.

  The solutions to solve these problems include

  1. Add more controllers.

  2. Design the controller so that it is primarily delegates the fulfilment of each system operation responsibility on to objects, rather than doing all or most of the work itself.

- *Role controllers:* Assign a responsibility to a human-role object in a way that mimics what that role does in the real world is acceptable. However, if a role controller is chosen, avoid the trap of designing person-like objects to do all the work.

7.3.5 A Design of POST

This section applies the GRASP patterns to assign responsibilities to classes, and to create collaboration diagrams for POST. We consider **Buy Items with Cash** and **Start Up** use cases in this design. Before we start the work on the design, we first state the following guidelines for making a collaboration diagram:

1. Create a separate diagram for each system operation which have identified and whose contracts are defined.
2. If the diagram gets complex, split it into smaller diagrams.
3. Using the contract responsibilities and post-conditions, and use case description as a starting point, design a system of interacting objects to fulfill the tasks. Apply the GRASP to develop a good design.

This is partly illustrated in Figure 7.27.

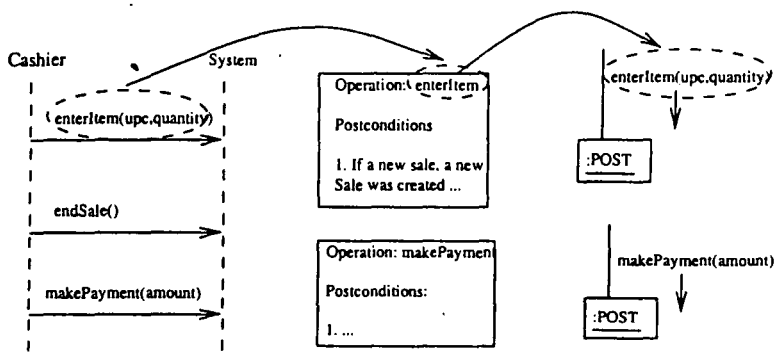


Figure 7.27: Guidelines for design

With **Buy Items with Cash** and **Start Up** use cases, we have identified four system operations *enterItem*, *endSale*, *makePayment*, and *StartUp*.

According to our guidelines we should construct at least four collaboration diagrams. According to the Controller pattern, the *POST* class could be chosen as controller for handling these operations. If so, we have the four starting diagrams shown in Figure 7.28 to complete.

Collaboration diagram for *enterItem*

First by Controller, *POST* is a candidate as the controller for handling this operation. And *POST* is satisfactory if there are only a few system operations. We make this choice in our example.

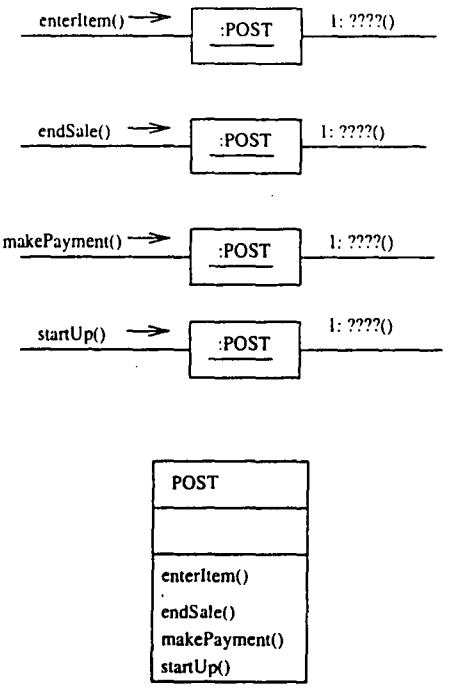


Figure 7.28: System operations

After assigning the responsibility for handling *enterItem* to *POST*, we need to look at the contract of *enterItem* what *POST* needs to do to fulfill this responsibility.

Displaying item description and price

In general, model objects are not responsible for display of information. These are the responsibilities of interface objects which can access to data known by model objects and can send messages to model objects. However, model objects do not send messages to interface objects in order to keep the system independent of the interface.

Creating a new *Sale*

The post-conditions of *enterItem* indicate a responsibility for creation an object *Sale*. The Creator pattern suggests *POST* is a reasonable candidate creator of the *Sale*, as *POST* records the *Sale*. Also by having *POST* create the *Sale*, the *POST* can easily be associated with it over time.

In addition to the above, when the *Sale* is created, it must create an *empty* collection to record all the future *SaleLineItem* that will be added. This collection will be maintained by the *Sale* instance, which implies by Creator that the *Sale* is a good candidate for creating it.

Therefore, we have the collaboration diagram shown in Figure 7.29.

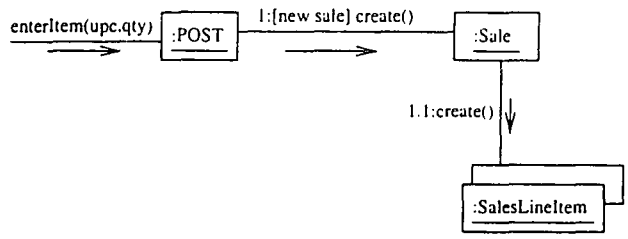


Figure 7.29: Sale creation

Creating a new *SalesLineItem*

The other post-conditions in the contract for *enterItem* indicate the responsibility to create a *SalesLineItem* instance. From the conceptual model and creator, *Sale* is an appropriate candidate creator for the *SalesLineItem* object. And by having the *Sale* create *SalesLineItem*, the *Sale* can be associated with the *SalesLineItem* object over time by storing the new instance in its collection of line items. The post conditions also indicate that the new *SalesLineItem* objects needs a quantity when it is created. Therefore, the *POST* must pass the quantity along to *Sale*, which must pass it along as a parameter in the creation message to the *SalesLineItem*.

Finding a *ProductSpectification* The postcondition in the contract of *enterItem* also requires the newly created *SalesLineItem* to be associated with the *ProductSpecification* that matches the *upc* of the item. This implies that the parameters to the *makeLineItem* message sent to the *Sale* include the *ProductSpecification* instance, denoted by *spec*, which matches the *upc*. This then requires to retrieve this *ProductSpectification* objects *before* the message *makeItem*(*spec*, *qty*) is sent to the *Sale*. From the conceptual model and the Expert pattern, *ProductCataLog* logically contains all the *ProductSpecifications*. Thus by Expert, *ProductCataLog* is a good candidate for looking up the *ProductSp*

Visibility to a ProductCatalog

After we have assigned the responsibility to *ProductCataLog* for looking up the *ProductSpecification*, we sould also decide the object that should send the message to the *ProductCatalog* to ask for a *ProductSpecification*.

In order for one object $:\underline{A}$ to send a message to another object $:\underline{B}$, object $:\underline{A}$ must be currently linked to object $:\underline{B}$. In OOD, the linkage between two objects is closely related to the concept of **visibility** of an object. The difference and relationship between the concepts of *association* and *visibility* are:

- If $:\underline{A}$ has visibility to $:\underline{B}$, there must be a link between $:\underline{A}$ and $:\underline{B}$, and thus there must be an association between class A and class B.
- If there is current a link $:\underline{A}$ and $:\underline{B}$, then one of the two objects has visibility to the other, though it is not necessarily they have visibility to each other.

We should now restate more precisely about the message passing between objects as

> *In order one object $:\underline{A}$ to send a message to another object $:\underline{B}$, object $:\underline{A}$ must currently have visibility to object $:\underline{B}$.*

There are the following four ways that visibility can be achieved from object $:\underline{A}$ to $:\underline{B}$:

1. *Attribute visibility* – $:\underline{B}$ is an attribute of $:\underline{A}$. This is a relatively permanent visibility because it persists as long as the two objects exists.
2. *Parameter visibility* – $:\underline{B}$ is a parameter of a method of $:\underline{A}$. This is a relatively temporary visibility because it persists within the scope of the method.
3. *Locally declared visibility* – $:\underline{B}$ is declared as a local object in a method of $:\underline{A}$. This is a relatively temporary visibility because it persists within the scope of the method.
4. *Global visibility* – $:\underline{B}$ is in some way globally visible. This is a relatively permanent visibility.

Now come back to our question about which object has visibility and can send a message *ProductCataLog* object to find a specification. This is not clear in the conceptual model. So we need add an association which links a class and *ProductCatalog*, and the links between these objects are established during the initial *StartUp* use case so that it can be used when creating a *SalesLineItem*. The postconditions of *StartUp* include the creation of the *POST* and the *ProductCatlog*, and the formation of the association between these two objects. This implies that we can safely assume that there is a permanent connection from the *POST* object to the *ProductCatalog* object. Therefore, it is possible for the *POST* to send the message, denote by *specification*, to the *ProductCatalog* object. Based on this discussion, we now can complete the collaboration diagram for *enterItem* shown in Figure 7.30.

From the collaboration diagram, we can write a pseudo-code for the method *enterItem*() of the *POST*:

```
POST--enterItem(upc,qty)
{
  if new sale then <<new>>Sale.create()
  spec:=ProductCataLog.specification(upc)
  Sale.makeLineItem(spec,qty)
}
```

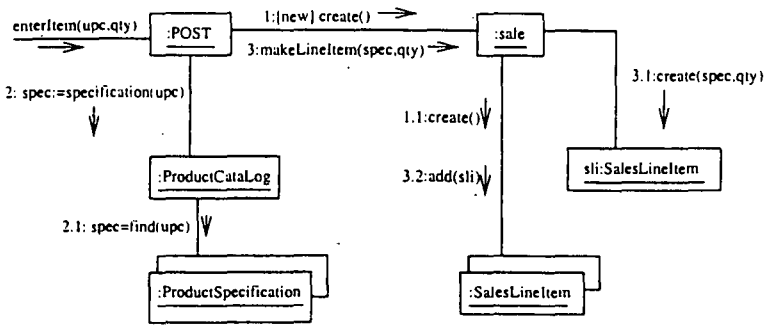


Figure 7.30: The *enterItem* collaboration diagram

Remarks about messages to multiobjects

Notice that in general, a message, such as *find* and *add*, sent to a multiobject is to the collection object itself, not to every object in the collection. The message 1.1 : *create*() sent to the *SalesLineItem* multiobject is for the creation of the collection data structure represented by the multiobject, it is not the creation of an instance of class *SalesLineItem*. The *add* message (3.2) sent to the *SalesLineItem* multiobject is to add element to the collection data structure represented by the multiobject.

Collaboration Diagram for *endSale*

The first design decision is to choose a controller to handle this system operation. We continue to use *POST* as the controller for this operation.

The contract of *endSale*() requires

- *Sale.isComplete* was set to *true* (*attribute modification*).

By Expert, it should be the *Sale* itself to carry out this responsibility, since it owns and maintains the *isComplete* attribute. Thus the *POST* will send a message, denoted by *becomeComplete*, to the *Sale* in order to set it to *true*. This assignment of responsibility is shown in Figure 7.31.

Calculating the *Sale* total Although we are not at the moment concerned ourselves with how the sale total will be displayed, we must *ensure that all information that must be displayed is known and available from the domain objects*. Note that no class current knows the sale total, so we need to create a design of the object interactions, using a collaboration diagram, to calculate the sale total.

In Section 7.4.1, we used the assignment of the responsibility for knowing and returning the total of sale to understand the use of the Expert Pattern. There we obtained the collaboration diagram illustrated

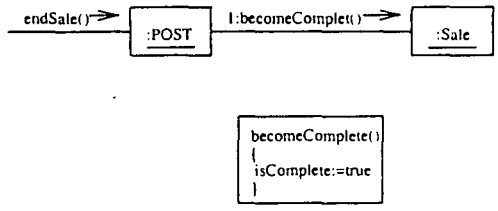


Figure 7.31: Completion of item entry

in Figure 7.19. In Figure 7.32, we only add some pseudo-codes to clarify the purposes of some of the operations.

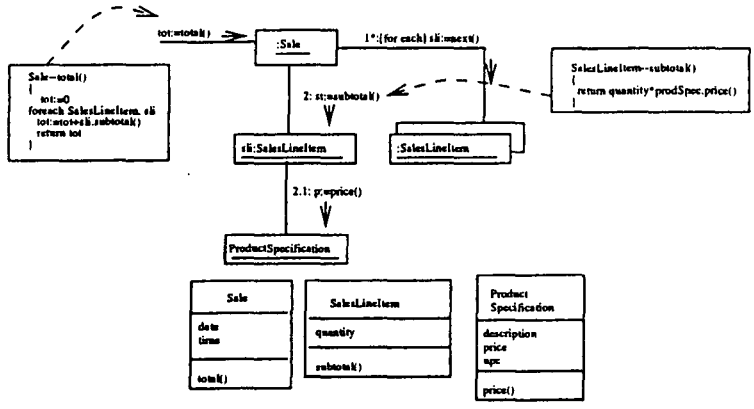


Figure 7.32: sale-total collaboration diagram

In general, since arithmetic is not usually illustrated via messages, the details of the calculations can be shown by attaching constraints to the diagram that defines the calculations.

We have one question unanswered: *who will send the total message to the Sale?* Most likely it will be an object in the interface, such as a Java applet.

Collaboration diagram for *makePayment*

Again, we shall use *POST* again as the controller. During our discussion about the patterns *Low Coupling* and *Cohesion*, we decided that we should assign the responsibility of the creation of the *Payment* to the *Sale* rather than to the *POST*. Additionally, it is reasonable to think that a *Sale* will closely use a *Payment*, and our choice is thus also justifiable by the Creator pattern. This discussion leads to the collaboration diagram in Figure 7.33

This collaboration diagram satisfies the postconditions that

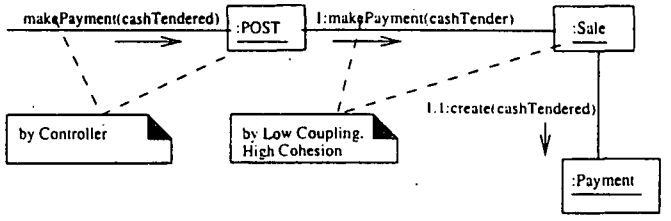


Figure 7.33: *makePayment* collaboration diagram

- the *Payment* was created,
- The *Payment* was associated with the *Sale*, and
- *Payment.amount* was set with *cashTendered*.

Logging the *Sale*

Once complete, the requirements state that the *Sale* should be placed in an historical log. By the Expert pattern. By expert, it is reasonable for the *Store* to know all the logged sales.

However, in some financial applications, the object representing the overall business may be come incohesive if we assign it with all the responsibilities of logging all the financial information. In such a case, other alternatives include the use of classic accounting concepts such as a *SalesLedger*. Notice that the postconditions of the contract of *makePayment* include *associating the Sale with the Store*. If a *SalesLedger*, rather than the *Store*, is used for logging the complete *Sale*, *SalesLedger* would ideally be added to the conceptual model, and the contract for *makePayment* would be adjusted accordingly as well. This kind of discovery and change during the design phase is perfectly reasonable and to be expected.

For our POST system, we will stick with the original plan of using the *Store* for logging the sales. This leads to the collaboration diagram in Figure 7.34.

Calculating the balance

The responsibilities section of the contract of *makePayment* requires that the balance is calculated; it will be printed on a receipt and displayed. Note that no class currently knows the balance, so we need to assign this responsibility to a class. To calculate the balance, the sale total and payment cash tendered are required. By Expert, *Sale* and *Payment* are partial expert on solving this problem.

If the *Payment* is primarily responsible for knowing the balance, it would need visibility to the *Sale*, in order to send the message to the *Sale* to ask for its total. Since the *Payment* does not currently have

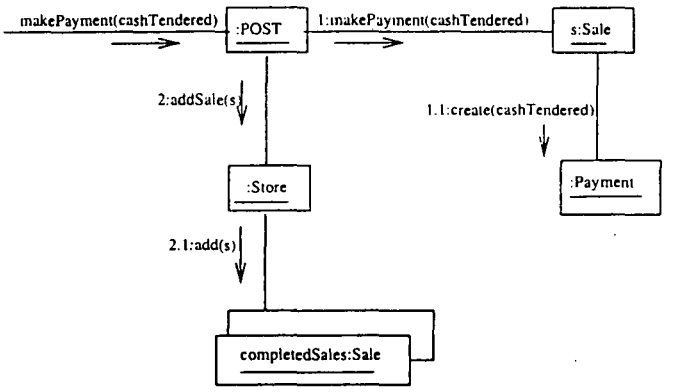


Figure 7.34: Logging a complete sale

visibility to the *Sale*, this approach would increase the overall coupling in the design.

In contrast, if the *Sale* takes the primary responsibility for knowing the balance, it needs the visibility to the *Payment*, in order to send it the message to ask for its cash tendered. Since the *Sales*, as the creator of the the *Payment* already has the visibility to the *Pyament*, this approach does not not increase the overall coupling, and it is a preferable design. This design is shown in Figure 7.35

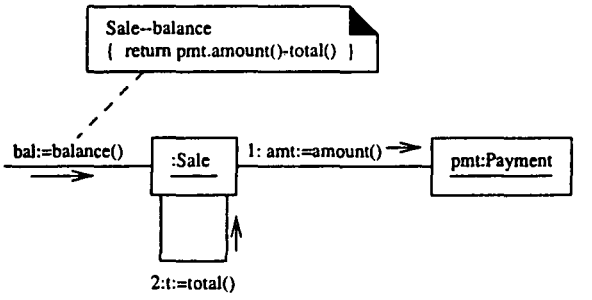


Figure 7.35: Collaboration diagram for the calculation of the sale balance

An interface object will be responsible for sending the message *balance* to the *Sale*.

Collaboration diagram for *StartUp*

Most, if not all, systems have a *StartUp* use case, and some initial system operation related to the starting up of the application. Although this *StartUp* system operation is the earliest one to execute, delay the development of a collaboration diagram for it until after all other system operations have been considered. This ensures that significant information has been discovered concerning what initialisation activities are required to support the later system operation interaction diagrams.

The *startUp* operation abstractly represents the initialisation phase of execution when an application is launched. How an application starts up and initialises is dependent upon the programming language and operating system.

In all cases, a common design idiom is to ultimately create an **initial domain object**, which is the first problem domain object created. In its initialisation method this objects is then responsible for the creation of other problem domain objects and for their initialisation.

The problem of *who should be responsible for creating the initial domain object* is dependent upon the object-oriented programming language and operating system. For example, if a graphical user interface is used, such as a Java applet, its execution may cause the attributes to be initialised. One of these attributes may be the initial domain object, such as *Store* which is then responsible for the creation of other objects, such as *POST*:

```
public class POSTApplet extends Applet
{
public void init()
{
  post = store.getPOST();
}
// Store is the initial domain object.
// The Store constructor creates other domain objects
private Store store = new Store();

private POST post;
private Sale sale;
}
```

The collaboration diagram for the *startUp* operation illustrates what happens when the initial problem domain object is created, rather than how the initial domain object is created. Thus, this collaboration diagram is language *independent*, and applicable to different languages. Therefore, the creation of a collaboration diagram for the *startUp* operation starts with sending a *create*() message to the initial domain object to create it. However, the question is how to choose this initial domain object. One of the following alternatives can be chosen to be the class of the initial domain object:

- A class representing the entire logical information system.
- A class representing the overall business or organisation.

Choosing between these alternatives may be influenced by High Cohesion and Low Coupling considerations.

Therefore, for the POST application, we can either choose *POST* or *Store* as the initial domain object.

We choose the *Store* as the *POST* is already chosen as the controller to be responsible for handling the system operations.

Persistent objects: *ProductSpecification*

In a realistic application the *ProductSpecification* objects will be maintained in a *persistent storage* medium, such as relational or object-oriented database. During the *startUp* operation, if there are only a few of these objects, they may all be loaded into the computer's direct memory. However, if there are many, loading them all would consume too much memory or time. More likely, individual objects will be loaded on-demand into the memory as they are required.

The design of how to dynamically on-demand load objects from a database into memory is not considered in this module. You may have to learn this when you come to do your Software Engineering Project in the next semester. Here, we simply assume that all the *ProductSpecification* can be created in the direct memory.

Create the *Store* and start up the system

From the contract of *startUp*, and the early discussion, the creation of a collaboration diagram

1. starts with a message *create*() being sent to the *Store*,
2. in order to create the *POST* and to allow it to have visibility and send messages to the the *ProductCatalog* (see the *enterItem* collaboration diagram), the *Store* needs to create the *ProductCatalog*, represented by *pc* first,
3. then the *Store* creates the *POST* and passes the *pc* as a parameter of the message *create*() to the *POST*.
4. when the *ProductCatalog* is created, it needs to create the empty collection of the *ProductSpecification* objects to which further specifications can be added.

This algorithm (or design) is illustrated in the collaboration diagram in Figure 7.36.

Notice in the figure that *pc* calls a method of itself:

```
Productcatalog -- loadProdSpecs()
{
Repeat
   ps=new ProductSpecification(upc,price,description);
   Collection(ProductSpecification).add(ps)
```

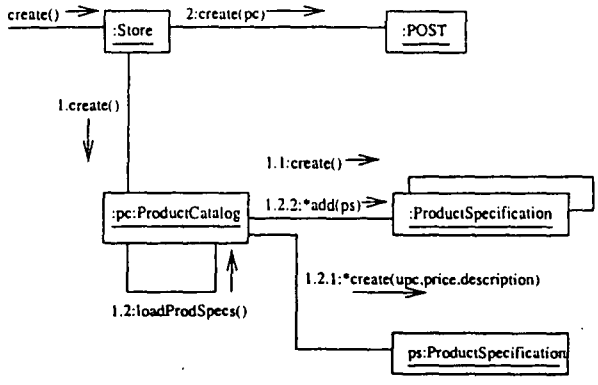


Figure 7.36: Collaboration diagram for *Startup*

```
Until finished
}
```

A deviation between analysis and design

The collaboration diagram for *StartUp* illustrates that the *Store* only create *one POST* object. However, the conceptual model that we created in Chapter 5 models a real store which may house *many* real point-of-sale terminals.

Regarding to this deviation, there are the following points to be made:

1. The *Store* in the collaboration diagram is not a real store; it is a software object.

2. The *Post* in the diagram is not a real terminal; it is a software object.

3. The collaboration diagram represents the interaction between one *Store* and one *POST* (certainly with other objects). This is what we are required to do by the requirement that we are current considering.

4. Generalisation from a single *POST* object to *many POST* objects requires that the *Store* to create *many POST* instances. And the interactions between these many *POST* objects with other objects would need to be synchronised in order to maintain the integrity of the shared information, such as logging complete *Sale* objects, and looking up the product specification. These would involve *multi-thread computation* or *concurrent computation*. Design and program concurrent systems is out of the scope of this module, but it is the main topic of MC306: Programming Concurrent Computer Systems.

7.3.6 Connecting User Interface Objects to Domain Objects

Although we are not concerned ourselves with the design of *user interface objects*, we are interested in knowing how an user interface object, once it is designed, is linked to a domain object.

As discussed during the creation of the *StartUp* collaboration diagram, if a graphical user interface is used, such as a Java applet, then the applet is responsible for initiating the creation of the initial domain object. The initial domain object in turn created other objects. Then the applet requests a reference to the *POST* object from the *Store*, and stores the reference in an attribute. With this attribute visibility to the *POST*, the applet may directly send messages to the *POST*. This is illustrated in Figure 7.37.

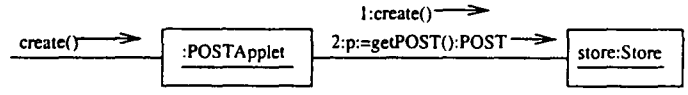


Figure 7.37: Interactions between the applet and a domain object

Once the applet has a connection with the controller *POST*, it can forward system input events message to it, such as the *enterItem*, *endSale*, and so on.

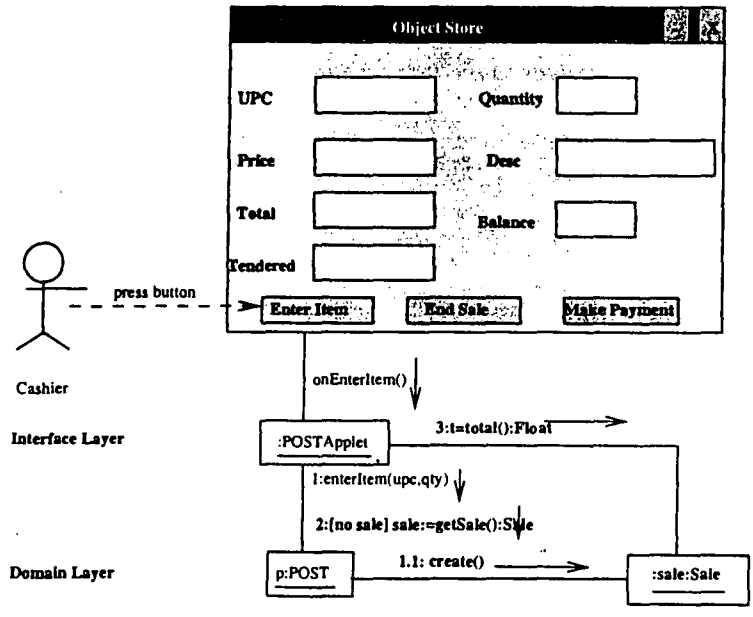


Figure 7.38: Connecting interface and domain objects

For example, consider the case of the *enterItem* message as illustrated in Figure 7.38:

1. The Cashier types in *upc* and *qty* (which are not shown in the Figure).
2. The Cashier requests *enterItem* by pressing the button "Enter Item".

3. The window forward the message by sending $onEnterItem()$ to the $Applet$.
4. The $Applet$ then sends the message $enterItem(upc, qty)$ to the $POST$.
5. The $Applet$ gets a reference to the $Sale$ (if it did not have one) by sending a message $getSale()$ to the $POST$.
6. $Applet$ sends a *total* message to the $Sale$ in order to get the information needed to display the running total.

Interface and domain layer responsibilities

We have to remember the following general principle:

> *The interface layer should not have any domain logic responsibilities. It should only be responsible for interface (presentation) tasks, such as updating widgets.*
>
> *The interface layer should forward requests for all domain-oriented tasks on to the domain layer, which is responsible for handling them.*
>
> *Interface layer has visibility to the domain layer and can directly send messages to the domain layer. However, the domain objects usually do not have visibility to the interface layer, and cannot send messages to the interface layer.*

This principle supports the separation between the design of the domain layer (application layer) and the design of the interface.

7.3.7 Design Class Diagrams

During the creation of a collaboration diagram, we record the *methods* corresponding to the responsibilities assigned to a class in the third section of the class box. For example, see Figure 7.39.

These classes with methods are *software classes* representing the conceptual classes in the conceptual models. Then based on these identified software classes, the collaboration diagrams and the original conceptual model, we can create a **design class diagram** which illustrate the following information:

- classes, associations and attributes
- methods,
- types of attributes
- navigability
- dependencies

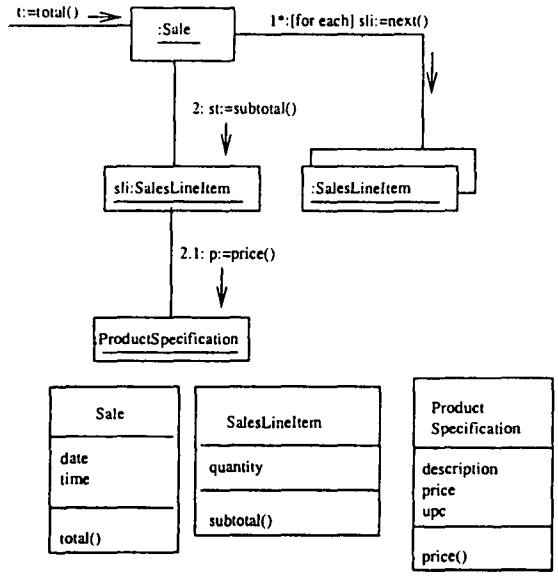


Figure 7.39: Recording methods of classes when create a collaboration diagram

Steps in making a design class diagram

Use the following strategy to create a design class diagram:

1. Identify all the classes participating in the object interactions. Do this by analysing the collaboration diagrams.
2. Draw them in a class diagram.
3. Copy the attributes from the associated concepts in the conceptual model.
4. Add method names by analysing the collaboration diagrams.
5. Add type information to the attributes and methods.
6. Add the associations necessary to support the required attribute visibility.
7. Add navigability arrows necessary to the associations to indicate the direction of attribute visibility.
8. Add dependency relationship lines to indicate non-attribute visibility.

These steps, except for steps 7&8, are mostly straightforward.

Add associations and navigability

Each end of an association is called a role. In a design class diagram, the role many be decorated with a navigability arrow. **Navigability** is a property of the role which indicates that it is possible to navigate uni-directionally across the association from objects of the source to the target class. An Example is shown in Figure 7.40.

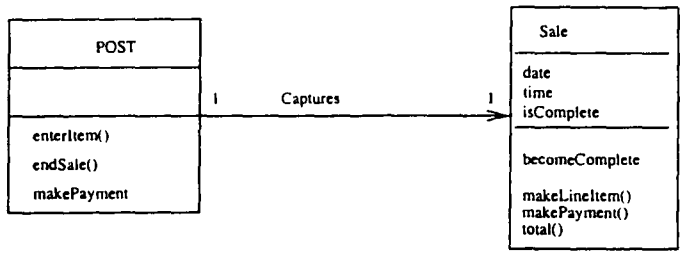


Figure 7.40: Showing navigability

Navigability is usually interpreted as attribute visibility from the source class to the target class. During the implementation in a OOP language it is usually translated as the source class having an attribute that refers to an instance of the target class. For example, the *POST* class will define an attribute that reference a *Sale* instance.

In a design class diagram, associations are chosen by a spartan software-oriented need-to-know criterion – what associations are needed in order to satisfy the visibility and ongoing memory needs indicated by the collaboration diagrams? This is in contrast with the associations in the conceptual model, which may be justified by intention to enhance comprehension of the problem domain.

The required visibility and associations between classes are indicated by the collaboration diagrams. Here are common situations suggesting a need to define an association with a navigability from A to B:

- A sends a message to B, e.g. the *POST* sends *makeLineItem* to the *Sale*.
- A create an instance of B, e.g. the *Store* creates the *POST*, the *POST* creates the *Sale*.
- A needs to maintain a connection to B by having an attribute whose value is of type B, e.g. the *Store* should have an ongoing connection with the *POST* and *ProductCatlog* objects it created.
- A receives a message with an object of B as a parameter.

Based on the above discussion, we have Figure 7.41 as a the design diagram with navigability information for the POST system.

Notice that this is not exactly the same set of associations that we generated for the conceptual model:

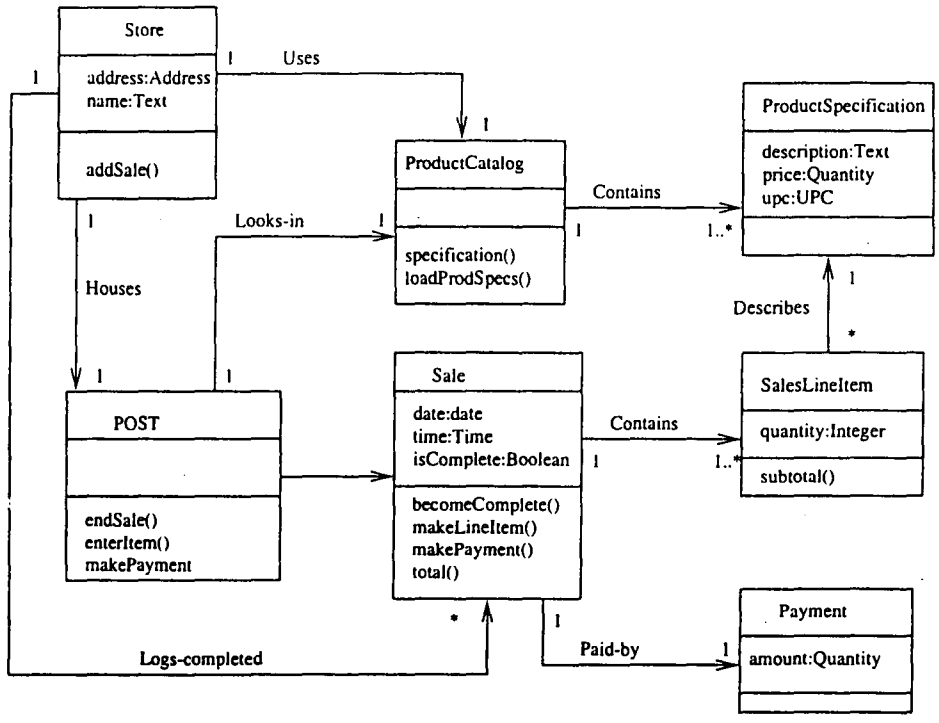


Figure 7.41: Design diagram with association and navigability

- The *Looks-in* association between *POST* and *ProductCatalog* was discovered during the design.
- The *Houses* association between *Store* and *POST* is now *one-to-one* association, rather than *one-to-many*.

Add dependency relationships

Apart from the attribute visibility, there are the other three kinds of visibility: parameter visibility, locally declared visibility, and global visibility. For examples:

1. The *POST* software object receives a returned object of type *ProductSpecification* from the specification message it sent to a *ProductCatalog*. Thus, *POST* has a short-term locally declared visibility to the *ProductSpecification*.
2. The *Sale* receives a *ProductSpecification* as a parameter in the *makeLineItem* message; it has parameter visibility to a *ProductSpecification*.

A non-attribute visibility may be illustrated with a dashed arrowed line indicating a *dependency relationship* from the source class to the target class. This is shown in Figure 7.42

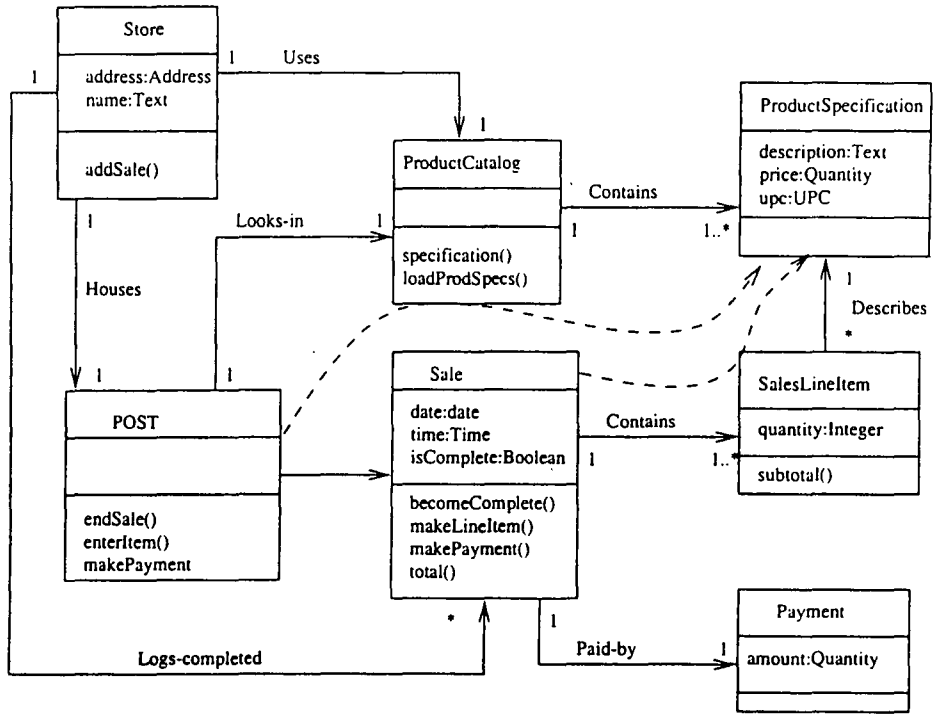


Figure 7.42: Design diagram with association and navigability

Some Remarks

- Some many of the concepts in the conceptual model, such as *Cashier*, *Manager*, *Customer*, and *Item*, are not present in the design. They are not required to be there by the requirement (the use case) that we are currently considering.
- Sometimes, it needs to add some classes which are not in the conceptual model but discovered later during the design into the design class diagram.
- We do not show the message *create*() sent to a class as a method of the class, because its interpretation is dependent on programming languages.
- *Accessing methods* are those which retrieve (*accessor method*) or set (*mutator methods*) attributes. It is a common idiom to have an accessor and mutator for each attribute, and to declare all attributes private (to enforce encapsulation). These methods are also usually excluded from the class diagram because of the high noise-to-value ratio they generate.

7.4 Questions

1. What is the difference between *msg(), *[true] msg(), and [true] msg() in a collaboration diagram?

2. You might think it would be more obvious just to number all the messages in a collaboration diagram using $1, 2, \ldots$, rather than using the nested scheme. What diference would it make? Can you construct a situation in which it would be ambiguous?

3. We have said that the UML creation message is denoted by $create()$ (sometimes with parameters) with the newly created object labelled with $<< new >>$. In a collaboration diagram, objects my be deleted/destroyed as wel as created. The UML notation for a message destroying an object is $destroy()$ with the deleted objects labelled with $<< destroyed >>$.

   Can you think of a situation when we need to show in a collaboration diagram an object being destroyed?

4. Develop a collaboration diagram in which:

   (a) an object O receives message *meg* from an actor;

   (b) O create a new object P;

   (c) P sends a message to Q;

   (d) then O destroy P.

   We did not get into the precise syntax and detailed meaning of the UML sequence diagrams. Read about the syntax of a sequence diagrams from Rational Rose tool or from a book to understand how *activations of objects* and *flow of control* are described in a sequence diagram. Draw a sequence diagram which models the above interactions.

5. If there is a message $x := foo()$ from object O to object P, should be x the name of an attribute of O or P?

   Discuss when it would and would not be meaningful and useful to name the return value from a message?

6. What is the difference between the two sequence diagram fragments shown in Figure 7.43?

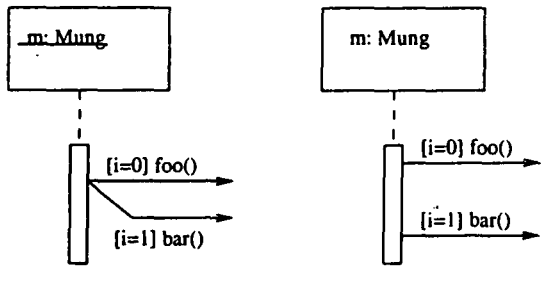


Figure 7.43: Two sequence diagram fragments

7. What is the difference between the two collaboration diagrams shown in Figure 7.44?

8. Discuss the relationship between the conceptual model and the design class model in terms of their classes and associations. What are the possible ways to associate two classes in a design class diagram?

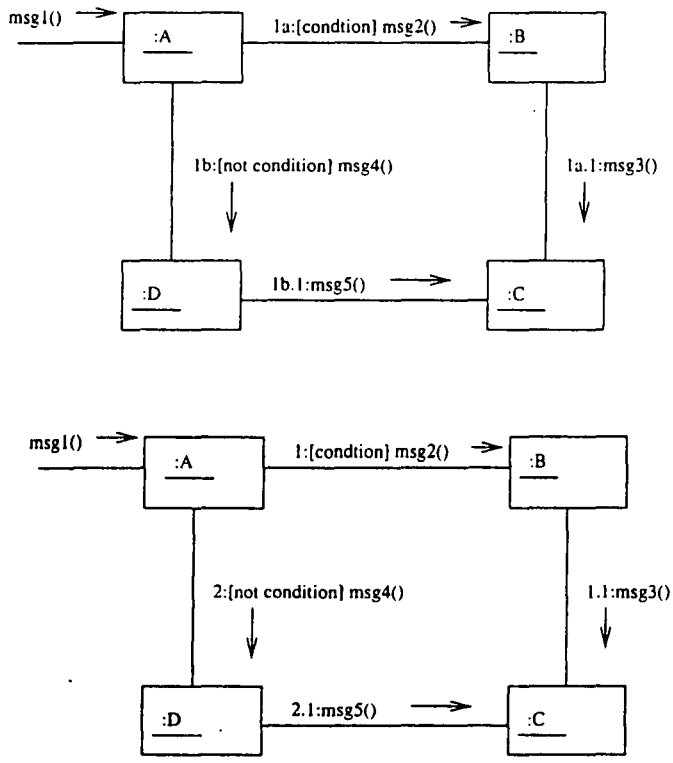


Figure 7.44: Two sequence diagram fragments

9. Consider the following method.

```
msg1()
{
for i:=1 to 10 myB.msg2()
for i:=1 to 10 myC.msg3()
}
```

How do you, in a collaboration diagram, represent the above situation? Do you have any problem if we still use the collaboration diagram in Figure 7.7?

10. Figure ?? is not a complete/general collaboration diagram for operation *getStudentNames*() from an instance of class *Lecturer*, as the lecturer may teach more or less than two modules, and more or less than three students. For the same application of Figure ??, design a general collaboration diagram for *getStudentNames*().

11. If we introduce use-case handlers as controllers, what changes need to be made in the artifacts of the requirements analysis and the design?

Chapter 8

Implementing a Design

Topics of Chapter 8

- The notion of the interface of a class and its features
- Definition of a class in a programming language
- Definition of a method of a class in a programming language.

The end goal of an object-oriented development of a system is the creation of code in an object-oriented programming language. The artifacts created in the design phase provide a significant degree of the information necessary in order to generate the code. *Implementation* in an object-oriented programming language requires writing source code for:

- class definitions – define the classes in the design class diagram in terms of the programming notation.
- method definitions – define the methods of classes in the design class diagram in terms of the programming notation.

We shall discuss these definitions in Java. We first discuss some notations.

8.1 Notation for Class Interface Details

The *interface* of a class primarily consists of the declarations of all the methods (or operations) applicable to instances of this class, but it may also include the declaration of other classes, constants,

variables (attributes), and initial values. Therefore, the interface of a class provides its outside view and emphasises the abstraction while hiding its structure and the secretes of its behaviour. By contrast, the *implementation* of a class is its inside view, which encompasses the secretes of its behaviour. The implementation of a class primarily consists of the definitions of all the methods (or the implementation of all the operations) declared in the interface of the class.

The interface of a class can be in general divided into three parts

- *Public:* A declaration that is accessible to all the *clients* which are the classes that have attribute visibility to this class.
- *Protected:* A declaration that is accessible only to the class itself, its subclasses, and its friends[1].
- *Private:* A declaration that is accessible only to the class itself and its friends.

The UML provides a rich notation to describe features of the interface of a class. Attributes are assumed to be private by default. The notation for other kind of interface declarations is shown in Figure 8.1. The

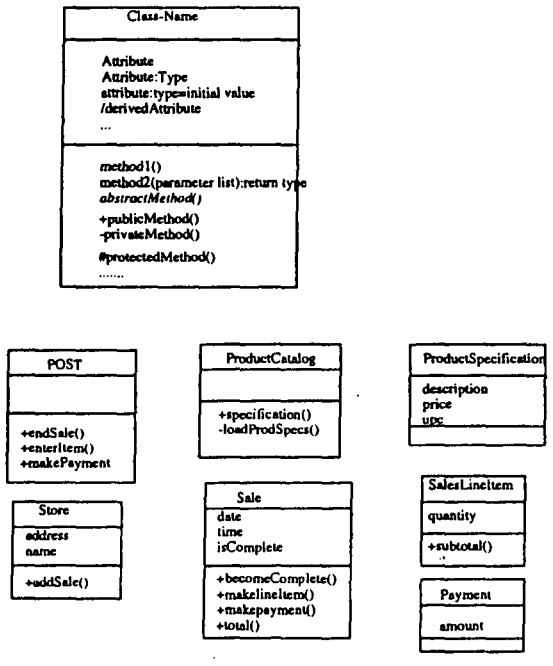


Figure 8.1: Interface details

figure also include some interface information about the classes in the POST system. Notice that, except for *loadProdSpecs*() which is a private method of *ProductCatalog*, all other methods are public.

[1] *Friends* are cooperative classes that are permitted to see each other's private parts.

8.2 Mapping a Design to Code

At the very least, design class diagrams depict the class name, superclasses, method signatures, and simple attributes of a class. This is sufficient to create a basic class definition in an object-oriented programming language. More information can be added later on after such a basic definition is obtained.

8.2.1 Defining a class with methods and simple attributes

Consider the class *SalesLinterItem* and the partial design class diagram in Figure 8.2. A mapping of

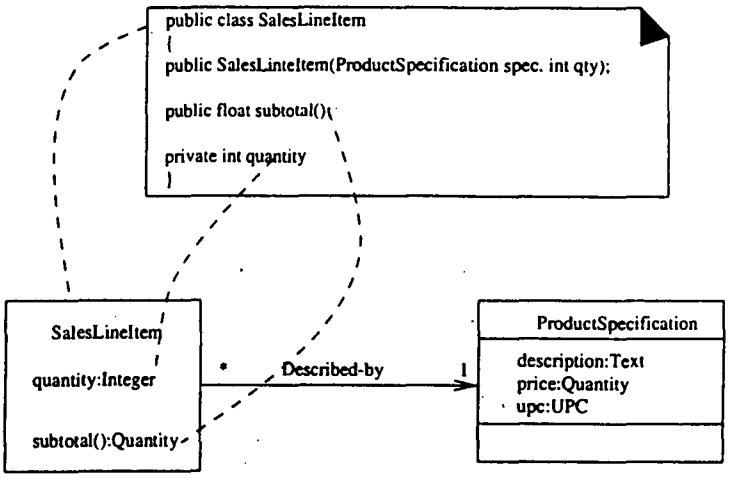


Figure 8.2: *SalesLineItem* in Java

the class box for *SalesLineItem* in the design diagram to the basic attribute definitions and method signatures for the Java definition of *SalesLineItem* is straightforward.

Note that, we had to add the Java *constructor SalesLineitem*(...) because of the fact that a *create*(*spec*, *qty*) message is sent to a *SalesLineItem* in the *enterItem* collaboration diagram. This indicates, in Java, that a constructor supporting these parameters is required.

Observe also that the return type for the *subtotal* method was changed from *Quantity* to a simple *float*. This implies the assumption that in the initial coding work, the programmer does not want to take the time to implement a *Quantity* class, and will defer that.

8.2.2 Add reference attributes

A **reference attribute** is an attribute that refers to another complex object, not to a primitive type such as a String, Number, and so on.

The reference attribute of a class are suggested by the associations and navigability in a design class diagram.

For example, a *SalesLineItem* has an association to a *ProductSpecification*, and with navigability to it. This navigability is needed for sending the message *price* to the *ProductSpecification* from the *SalesLineItem* in the collaboration diagram for *total* of the *Sale*. In Java, this means that an instance variable referring to a *ProductSpecification* instance is suggested.

Reference attributes are often implied, rather than explicit, in a class diagram. Sometimes, if a role name for an association is present in a class diagram, we can use it as the basis for the name of the reference attribute during code generation. The Java definition of the class *SalesLineItem* with a reference attribute *prodSpec* is shown in Figure 8.3.

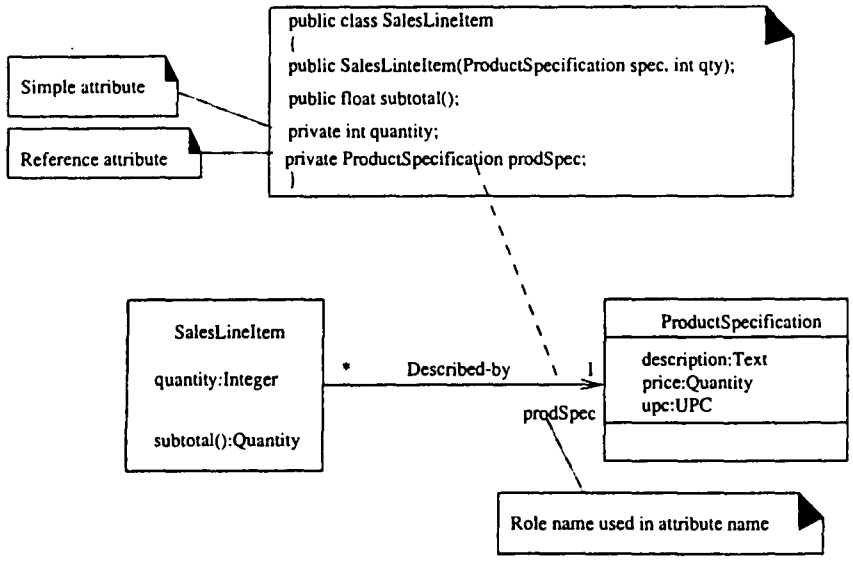


Figure 8.3: Add reference attributes

In the same way as we have defined the Java class for *SalesLineItem*, we can define the Java class for *POST*. This is shown in Figure 8.4.

8.2.3 Defining a method from a collaboration diagram

A collaboration diagram shows the messages that are sent in response to a method invocation. The sequence of these messages translates to a series of statements in a method definition.

For example, consider the collaboration diagram for the *enterItem* operation given in Figure 8.5.

We should declared *enterItem* as a method of the *POST* class:

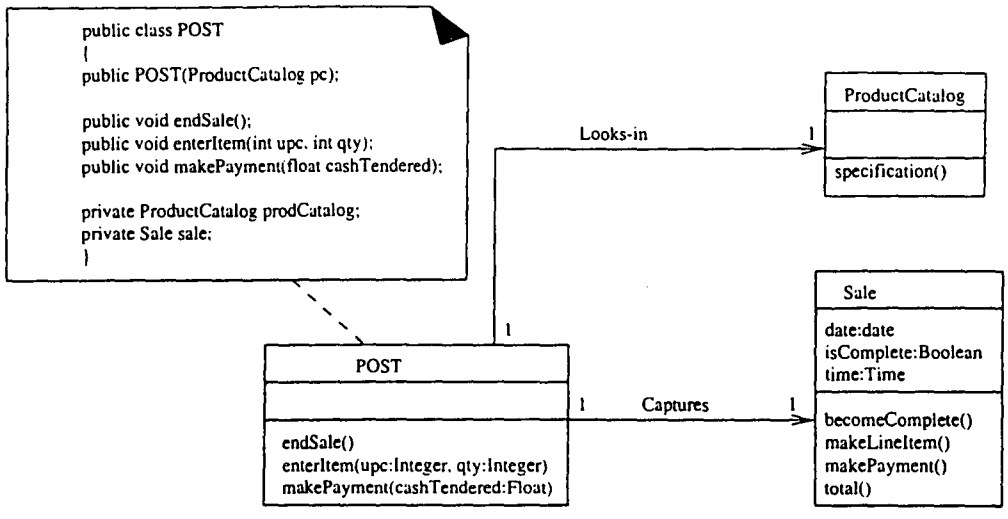


Figure 8.4: The POST class in Java

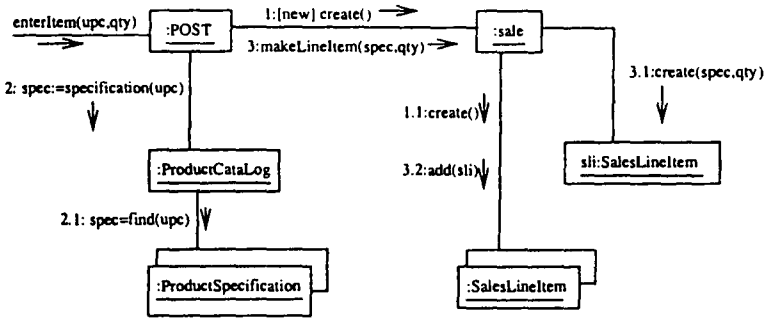


Figure 8.5: The *enterItem* collaboration diagram

```
public void enterItem(int upc, int qty)
```

This is also shown in Figure 8.4. Then we have to look at the messages sent by *POST* in response to the message *enterItem* received by *POST*.

Message 1: According to the collaboration diagram, in response to the *enterItem* message, the first statement is to conditionally create a new *Sale*.

```
if (isNewSale()) { sale = new Sale(); }
```

This indicates that the definition of the *POST – enterItem* method — needs the introduction of a new method to the *POST* class: *isNewSale*. This is a small example of how, during the coding phase,

changes from the design will appear. It is possible that this method could have been discovered during the earlier solution phase, but the point is that changes will inevitably arise while programming.

As a first attempt, this (private) method will perform a test based on whether or not the *sale* instance variable is *null* (i.e. *sale* does not point to anything yet):

```
private boolean isNewSale()
{
  return ( sale == null);
}
```

You may wonder why not simply hard-code this test into the *enterItem* method? The reason is that it relies on a design decision about the representation of information. In general, expressions that are dependent on representation decisions are best wrapped in methods so that if the representation changes, the change impact is minimised. Furthermore, to a reader of the code, the *isNewSale* test is more informative in terms of semantic intent that the expression

```
if (sale == null)
```

On reflection, you will realize that this test is not adequate in the general case. For example, what if one sale has completed, and a second sale is about to begin. In that case, the *sale* attribute will not be null; it will point to the last sale. Consequently, some additional test is required to determine if it is a new sale. To solve this problem, assume that if the current sale is in the *complete* state, then a new sale can begin. If it turns out later this is an inappropriate business rule, a change can be easily made. Therefore,

```
private boolean isNewSale()
{
  return ( sale == null ) || ( sale.isComplete() );
}
```

Based on the above coding decisions, the new method *isNewSale* needs to be added to the *POST* class definition given in Figure 8.4. The design class diagram depicting the *POST* class should be updated to reflect this code change.

Message 2: The second message sent by the *POST* is *specification* to the the *ProductCatalog* to retrieve a *ProductSpecification*:

```
ProductSpecification spec =
              prodCatalog.specification(upc);
```

Notice the use of the reference attribute *prodCatalog*.

Message 3: The third message sent by *POST* is the *makeLineItem* to the *Sale*:

```
sale.makeLineItem(spec, qty);
```

In summary, each sequenced message within a method, as shown on the collaboration diagram, is mapped to a statement in the Java method.

Therefore, the complete *enterItem* of *POST* method is given as:

```
public void enterItem(int upc, int qty)
{
if (isNewsale()){sale=new Sale();}

ProductSpecification spec = prodCatalog.specification(upc);

sale.makeLineItem(spec,qty);
}
```

8.2.4 Container/collection classes in code

It is often necessary for an object to maintain visibility to a group of other objects; the need for this is usually evident from the multiplicity value in a class diagram – it may be greater than one. For example, a *Sale* must maintain visibility to a group of *SalesLineItem* instances, as shown in Figure 8.6.

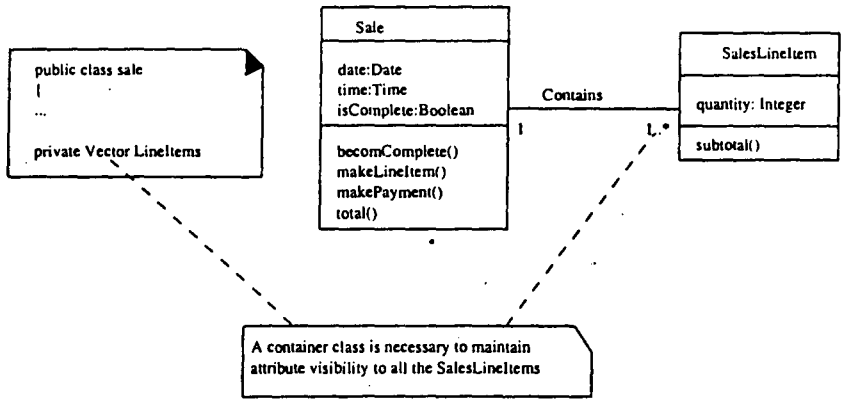


Figure 8.6: Reference to a container

In object-oriented programming languages, these relationships are often implemented with the introduction of an intermediate container or collection. The one-side class in the association define a reference attribute pointing a container/collection instance, which contains instances of the many-side class.

The choice of a container class is influenced by the requirements; key-based lookup requires the use of a *Hashtable*, a growing ordered list requires a *Vector*, and so on.

With these discussions, we can now define the *makeLineItem* method of the *Sale* class as shown in Figure 8.7.

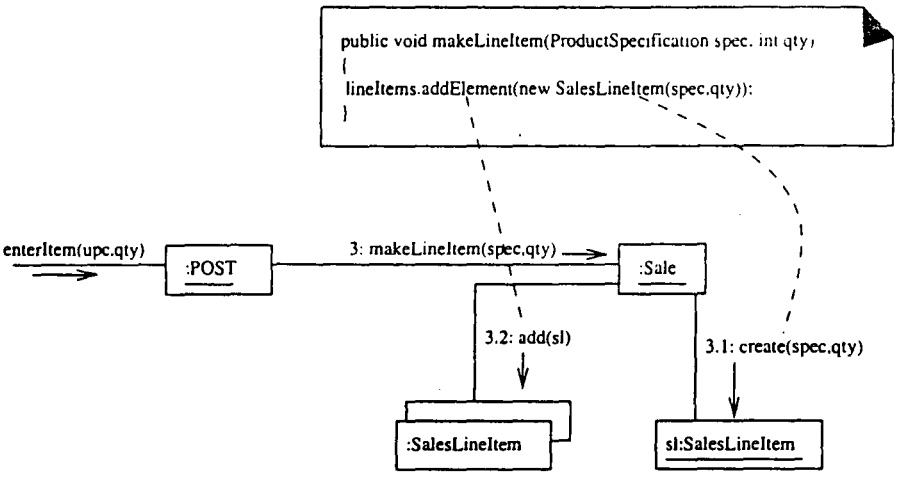


Figure 8.7: Sale – makelineItem method

Note that there is another example of code deviating from the collaboration diagram in this method: the generic *add* message has been translated into the *Java-specific addElement* message.

8.2.5 Exceptions and error handling

Error handling has been ignored so far in the development of a solution, as we would like to focus on the basic questions of responsibility assignment and object-oriented design. However, in real application development, it is wise to consider error handling during the design phase. For example, the contracts can be annotated with a brief discussion of typical error situations and the general plan of response.

The UML does not have a special notation to illustrate exceptions. Rather, the message notation of collaboration diagrams is used to illustrate exceptions. A collaboration diagram may start with a message representing an exception handling.

8.2.6 Order of implementation

Classes need to be implemented (and ideally, fully unit tested) from least coupled and most coupled. For example, in the POST system, possible first classes to implement are either *Payment* or *ProductSpecification*. Next are classes only dependent on the prior implementations – *ProductCatalog* or *SalesLineItem*

8.3 Questions

1. Define a Java class for *ProductcataLog*.
2. Define a Java class for *Store*.
3. Define a Java class for *Sale*.

Chapter 9

Advanced Modelling Concepts and Design Techniques

Topics of Chapter 9

- Iterative development process
- Generalization-specialization
- More about associations
- UML Packages
- State diagrams

9.1 Iterative Development Process

In the case study of the POST system, we have carried out three major steps of development: requirement analysis and specification, design, and implementation, by mainly considering only one use case, **Buy Items with Cash**. For this use case, we have also made assumptions to significantly simplify the problem (recall the five attributes of complex systems).

However, for some complex applications, an *iterative development process* or an *iterative life-cycle* is more effective compared with the waterfall life-cycle, in which each macro-step is done once for the entire system requirements.

An **iterative life-cycle** is based on successive enlargement and refinement of a system through *multiple* development cycles of analysis, design, implementation (and testing). Each cycle tackles a relatively

small set of requirements, proceeding through analysis, design, implementation and testing. The system grows incrementally as each cycle is completed. This is illustrated in Figure 9.1.

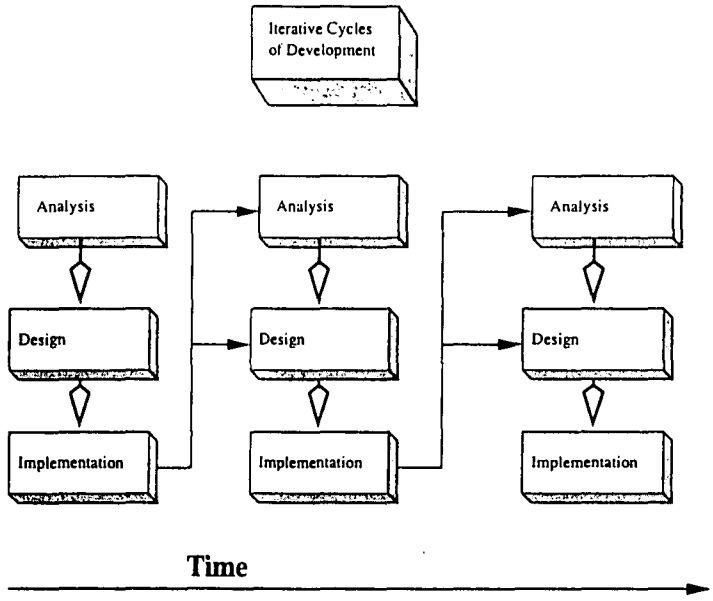


Figure 9.1: Iterative development cycles

Notice that the implementation result of a prior cycle can feed into the beginning of the next cycle. Thus the subsequent analysis and design results are continually being refined and informed from prior implementation work.

9.1.1 Use case and iterative development

As for the POST system, a development cycle is assigned to implement one or more use cases, or simplified versions of use cases. This is shown in Figure 9.2.

During the project planning stage, use cases should be ranked, and high ranking use cases should be tackled in early development cycles. The most useful strategy is to first pick use cases that significantly influence the core architecture, i.e. the use cases that concerns the most of the important domain concepts.

9.1.2 Development cycle 2 of POST

To illustrate the iterative development, we consider the second development cycle for the POST application. In this cycle, we consider the general **Buy Items** use case by adding more functionality to the **Buy Item with Cash** use case, so that it approaches the final complete use case.

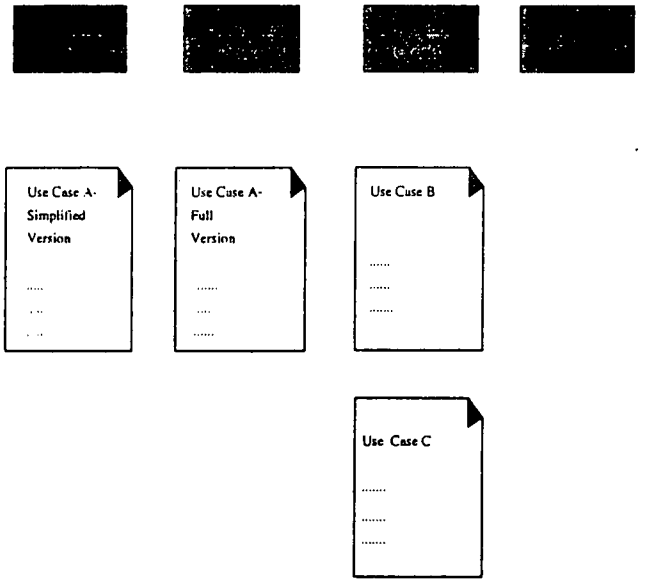


Figure 9.2: Use case driven development cycles

The first cycle made many simplification so that the problem was not overly complex. Once again–for the same reason–a relatively small amount of additional functionality will be considered:

The additional functionality is that cash, credit, and check payments will be supported.

Also, most of the simplifications that were made for the first cycle apply to this cycle, such as that there is no inventory maintenance, the system is not part of a larger organization, and so on.

Now we revisit the **Buy Items** use cases. This time we shall also give the sub-use-cases **Pay by Cash, Pay by Credit**, and **Pay by Check**.

Use case: **Buy Items**

Actors: Customer (initiator), Cashier.
Overview: A Customer arrives at a checkout with items to purchase. The Cashier records the purchase items and collects payment. On completion, the Customer leaves with the items.

Typical Course of Events

Actor Action	**System Response**

1. This use case begins when a Customer arrives at a POST checkout with items to purchase.
2. The Cashier records the identifier from each item.

   If there is more than one of the same item, the Cashier can enter the quantity as well.
3. Determines the item price and adds the item information to the running sales transaction.
   The description and price of the current item are presented.
4. On completion of the item entry, the Cashier indicates to the POST that item entry is completed.
5. Calculates and presents the sale total.
6. The Cashier tells the Customer the total
7. The Customer chooses payment method:

   (a) If cash payment, **initiate** *Pay by Cash.*

   (b) If credit payment, **initiate** *Pay by Credit.*

   (c) If check payment, **initiate** *Pay by Check.*
8. Logs the completed sale.
9. Prints a receipt.
10. The Cashier gives the printed receipt to the Customer.
11. The Customer leaves with the items purchased.

Alternative Courses

- Line 2: Invalid identifier entered. Indicate error.
- Line 7: Customer could not pay. Cancel sales transaction.

Use case: **Pay by Cash**

Actors: Customer (initiator), Cashier.
Overview: A Customer pays for a sale by cash at a point-of-sale terminal.

Typical Course of Events

Actor Action	System Response

1. This use case begins when a Customer chooses to pay by cash, after being informed of the sale total
2. The Customer gives a cash payment–the "cash tendered"–possibly greater than the sale total.
3. The Cashier records the cash tendered.
4. Shows the balance due back to the customer.
5. The Cashier deposits the cash received and extracts the balance owing.
   The Cashier gives the balance owing to the Customer

Alternative Courses

- Line 2: Customer does not have sufficient cash. May cancel sale or initiate another payment method.
- Line 7: Cash drawer does not contain sufficient cash to pay balance. Cashier requests additional cash from supervisor or asks Customer for different payment method.

Use case: **Pay by Credit**

Actors: Customer (initiator), Cashier, Credit Authorization Service (CAS), Accounts Receivable.

Overview: A Customer pays for a sale by credit at a point-of-sale terminal. The payment is validated by an external credit authorization service, and is posted to an accounts receivable system.

Typical Course of Events

Actor Action	System Response
1. This use case begins when a Customer chooses to pay by credit, after being informed of the sale total.	
2. The Customer gives the Cashier the credit card.	
3. The Cashier records the information of the credit card and requests a credit payment authorization.	4. Generates a credit payment request and sends it an external CAS.
5. CAS authorises the payment.	6. Receives a credit approval reply from the CAS
	7 Posts (records) the credit payment and approval reply information to the Accounts Receivable system (The CAS owes money to the Store, hence A/R must track it.)
	8. Display authorization success message.

Alternative Courses

- Line 4: Credit request denied by CAS. Suggest different payment method.

Use case: **Pay by Check**

Actors: Customer (initiator), Cashier, Check Authorization Service (CAS).
Overview: A Customer pays for a sale by check at a point-of-sale terminal. The payment is validated by an external check authorization service.

Typical Course of Events

Actor Action	System Response
1. This use case begins when a Customer chooses to pay by check, after being informed of the sale total	
2. The Customer writes a check and gives it to the Cashier together with the bank card.	
3. The Cashier records bank card information and requests check payment authorization.	4. Generates a check payment request and sends it an external Check Authorization Service.
5. Check Authorization Service authorizes the payment.	6. Receives a check approval reply from the Check Authorisation Service
	7. Display authorization success message.

Alternative Courses

- Line 4: Credit request denied by Check Authorization Service. Suggest different payment method.

9.1.3 Extending the conceptual model

Going through the *Concept Category List* and using the noun phrase identification, we can have a draft conceptual model shown in Figure 9.3.

9.2 Generalisation

The concepts *CashPayment*, *CreditPayment*, and *CheckPayment* are all very similar. A *Sale* can be paid by any of these methods of payment. A partial class diagram modeling this is given in Figure 9.4.

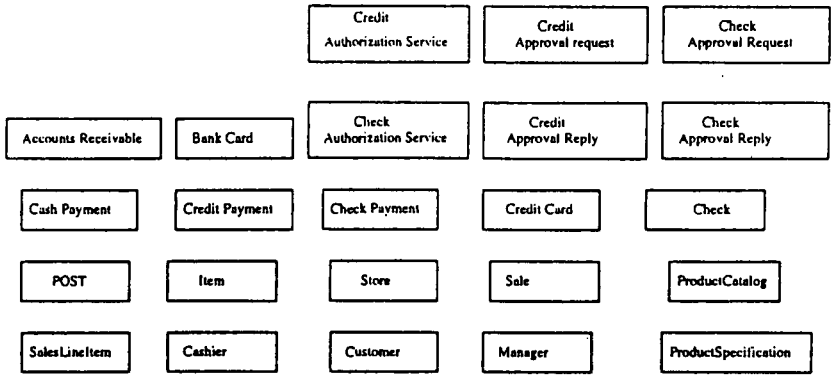


Figure 9.3: Draft conceptual model for cycle 2 of the development of POST

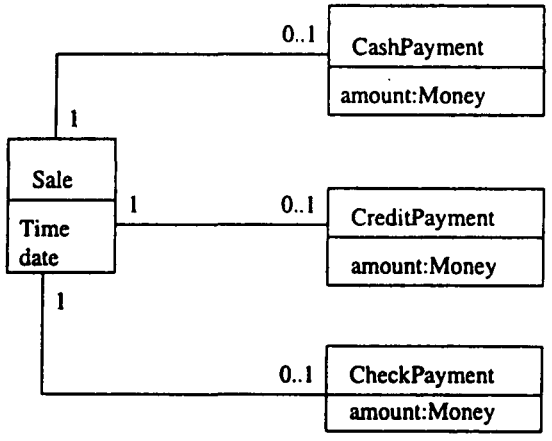


Figure 9.4: A *Sale* can be paid by any of the payment methods

There are the following problems with the model in Figure 9.4:

1. There are too many distinct association (though they have the same name), one between each payment method and the *Sale*.

2. The multiplicities in the model do not correctly capture the fact that a *Sale* is paid by one and only one of the payment methods; and an instance of a *Payment* of any kind is to pay a *Sale* instance.

3. If a new kind of payment were introduced, a new association would also have to be added to the model.

4. The model does not explicitly show the similarities of the three concepts of payment methods which probably have a large amount of common structure.

9.2.1 The notion of generalization

These problems can be overcome by making use of the notion of *generalization*. **Generalization** is the activity of identifying commonality among concepts and defining supertype and subtype relationships. In a **generalization-specialization** relationship, one class is identified as a 'general' class and a number of others as 'specializations' of it.

In UML, a generalization relationship is represented by a large hollow triangle pointing to the 'general concept' from a 'specialized concept'. With the notion of generalization and the UML modelling notation

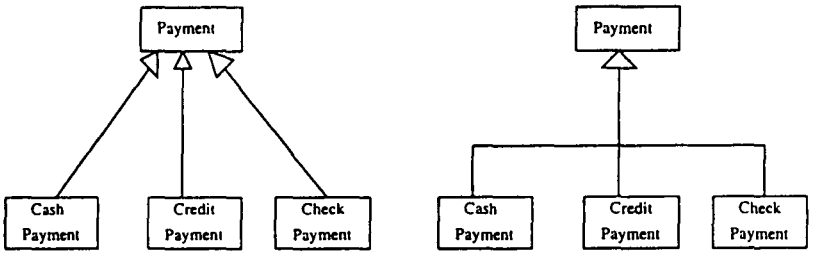


Figure 9.5: Generalization-specialization hierarchy with separate and shared triangle notations

for it, the associations between the *Sale* and the payment method can be clearly described by the model in Figure 9.6.

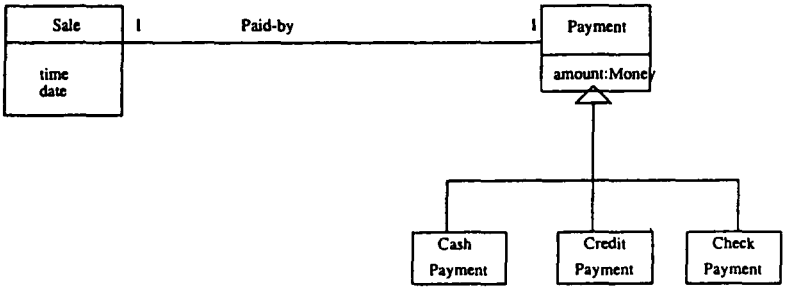


Figure 9.6: A *Sale* can be paid by one and only of the payment methods

9.2.2 The meaning of the generalization-specialization relationship

What is the relationship of a supertype to a subtype?

A supertype definition is more general or encompassing than a subtype definition.

For example, consider the supertype *Payment* and its subtypes (*CashPayment* and so on). Assume the definition of *Payment* is that it represents the transaction of transferring money (not necessarily cash)

for a purchase from one party to another, and that all payments have an amount of money transferred. A *CreditPayment* is a transfer of money via a credit institution which needs to be authorized. Thus, the definition of *Payment* encompasses and is more general than the definition of *CreditPayment*.

If we define a class as a set of objects, then subtypes and supertypes are related in terms of set membership:

> *All members of a subtype set are members of their supertype set:*
>
> $SubtypeSet \subseteq SupertypeSet$

This is also termed as the **Is-a-Rule** for testing a correct subtype:

Subtype **is a** *Supertype*

Inheritance of attributes and associations

When a generalization-specialization hierarchy is created, statements about a supertype that apply to its subtypes are made. For example, Figure 9.6 states that all *Payments* have an amount and are associated with a *Sale*. All *Payment* subtypes must conform to having an amount and paying for a *Sale*. In general, this rule of inheritance from (conformance to) a supertype is the 100% *Rule*:

> 100% *of the supertype's definition should be applicable to the subtype. The subtype must conform to* 100% *of the supertypes's:*
>
> - *attributes*
> - *associations*

The rules to ensure that a subtype is correct are the 100% and Is-a Rules.

When to define a subtype

A **type partition** is a division of a type into disjoint subtypes. *When is it useful to show a type partition?*

The following are strong motivations to partition a type into subtypes:

1. The subtypes have additional attributes of interest.
2. The subtypes have additional associations of interest.

3. The subtype concept is operated upon, handled, reacted to or manipulated differently than the supertype or other subtypes, in ways that are of interest.

4. The subtype concept represents an animate thing (for example, animal, robot) that behaves differently than the supertype or other subtypes, in ways that are of interest.

In a design class diagram, in addition to adding new features to a subclass, it sometimes happens that a class requires a customized version of a method (operation) that it has inherited from its superclass.

For example, assume that the *withdraw* operation in *Account* simply adjust the balance to take into account the money withdrawn. A high-interest account, on the other hand, might want to deter investors from making withdrawals by imposing a fixed charge for each withdrawal. Whenever a withdrawal is made, the balance should additionally be decremented by the amount of the fixed charge. This means that *HighInterestAccount* class needs to *refine* the *withdraw* operation of of *Account* class to provide this extended functionality.

9.2.3 Abstract classes

There is a particular kind superclasses, which are called *abstract classes*. When the system is running, no concrete instance of an abstract class is ever created. It is useful to identify abstract types in a conceptual model because they constrain what types it is possible to have concrete instances of. A class T is called an **abstract class** if every member (or an instance) of T must also be a member of a subtype.

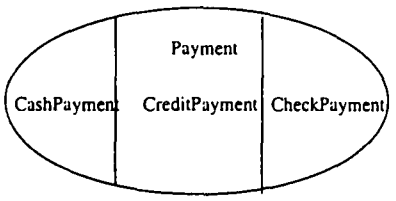
Therefore, an abstract class does not has instances of it own, and every instance of this type created during the execution of the system is created via the creation of an instance of a subtype of it. For example, assume every *Payment* instance must more specially be an instance of the subtype *CreditPayment*, *CashPayment*, or *CheckPayment*. Then *Payment* is an abstract type. In contrast, if there can exist *Payment* instances (in the system) which are not members of a subtype, it is not an abstract type. These two cases are illustrated in Figure 9.7.

In UML, the name of an abstract type in a class diagram is *italicized*, see Figure 9.8.

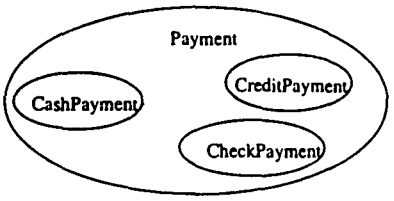
If an abstract type is implemented in software as a class during the design phase, it will usually be represented by an *abstract class*, meaning that no instances may be created for the class. An **abstract method** is one that is declared in an abstract class, but not implemented; in the UML it is also noted with italics.

9.2.4 Type hierarchies in the POST application

This section presents the the generalizations between types which can be justified by the four motivations discussed earlier.



(a): Every payment is made by one of the three methods



(b) There are payments which do not belong to any of the three kinds

Figure 9.7: (a): *Payment* is an abstract type; (b): *Payment* is not an abstract type

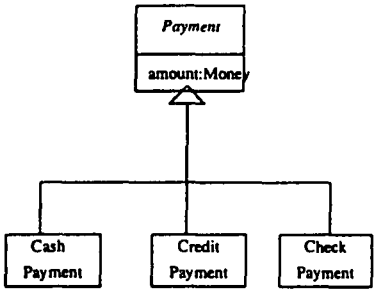


Figure 9.8: The name of an abstract type is *italicized*

Subtypes of *Payment*

The subtypes of payment have additional associations that are of interest, see Figure 9.9.

Subtypes of *AuthorizationService*

The subtypes of *AuthorizationService* have additional associations that are of interest, see Figure 9.10.

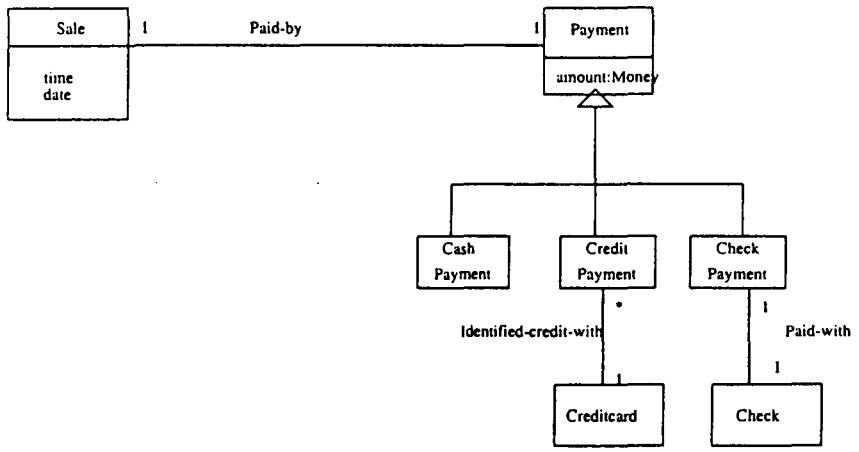


Figure 9.9: Justifying the subtypes of *Payment*

Subtypes of *Authorization Transaction*

The subtypes of *AuthorizationTransaction* have additional associations that are of interest, see Figure 9.11.

Modelling changing states

Assume that a payment can either be in an unauthorized or authorized state, and it is meaningful to show this in the conceptual model (it may not really be, but assume so for discussion). One approach to modelling this is to define subtypes of *Payment*: *UnauthorizedPayment* and *AuthorizedPayment*. However, note that a payment does not stay in one of these states; it typically transitions from unauthorized to authorized. This leads to the following guideline:

Do not model the states of a concept X as a subtype of X. Rather:

1. *Define a state hierarchy and associate the states with X, or*
2. *Ignore showing the states of a concept in the conceptual model; show the states in a state diagram instead.*

The first choice is shown in Figure 9.12.

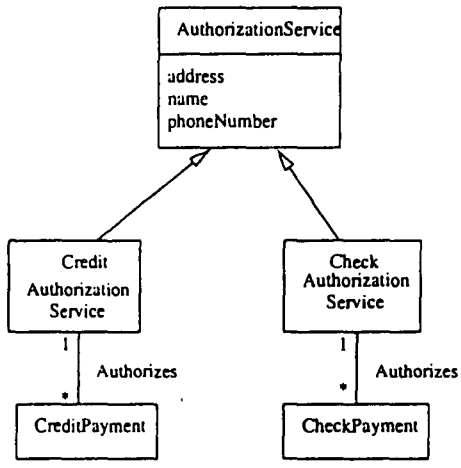


Figure 9.10: Justifying the subtypes of *AuthorizationService*

9.3 More about Associations

9.3.1 Associative types

As we said before, that attributes of class describe properties of the instances of a class. A *Student* class might have an attribute *name*, for example, and every *Student* instance would contain a data value giving the name of the student.

Sometimes, however, it is necessary to record information that is more naturally associated with the association between two classes than with either of the roles of the the association individually. In other words, an association between two classes can have interesting properties that need to be modelled, and these properties cannot be naturally described as attributes of either of the classes that the association links.

For example, consider the association *takes* between the classes *Student* and *Module*. Assume that the system needs to record all the marks gained by students on all the modules that they are taking. The UML notation that we have introduced so far for class diagrams does not enable us to model this situation easily:

- It is not sufficient to add a *mark* attribute to the *Student* class, as a student in general takes many modules and therefore it will be necessary to record more than one *mark* for each student. An attribute which allows a set of marks to be stored would get round this problem, but would not preserve the information about which mark was gained for which module. It would not be desirable to get round the problem by having the student instances explicitly store some kind of module identifiers. This would duplicate some of the information modelled by the association *Takes*.

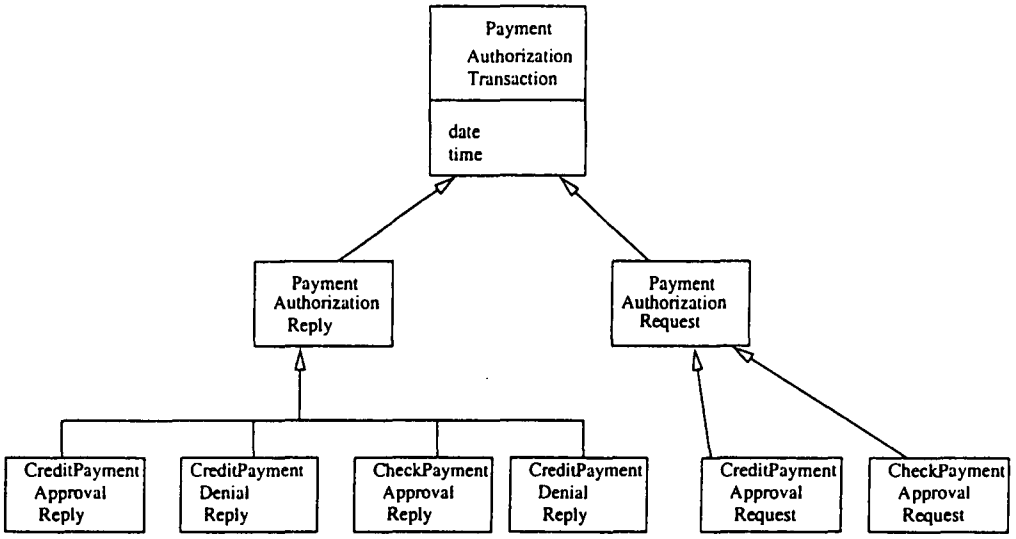


Figure 9.11: Justifying the subtypes of *AuthorizationTransaction*

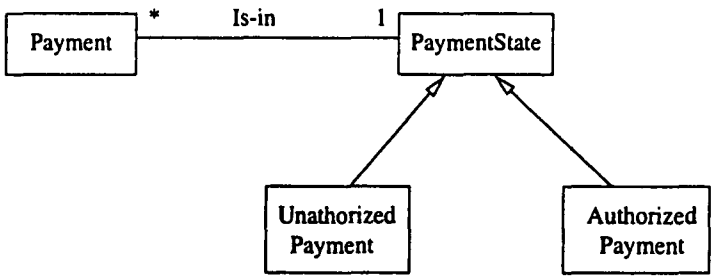


Figure 9.12: Modelling changing states

- The same problems arise if we consider to add a *mark* attribute to the *Module* class, as each module can be taken by many students.

Intuitively, an exam mark is neither a property that a student has in isolation nor a property that a module has in isolation. It is rather a property of the association *Takes* that linked the student and the module. What is required is some way of describing a property of an association. The notion of an *Associative class* is introduced for this purpose.

In UML, the *Takes* association between *Student* and *Module* is further described by an associative class in the way shown in Figure 9.13.

Like other types, an associative type can be associated with other class. For example, a class (in the sense of a group of students who are taught together) might be defined as a set of registrations, namely those for all the students registered to take a particular module in a particular semester. This situation is modelled by the class diagram in Figure 9.14.

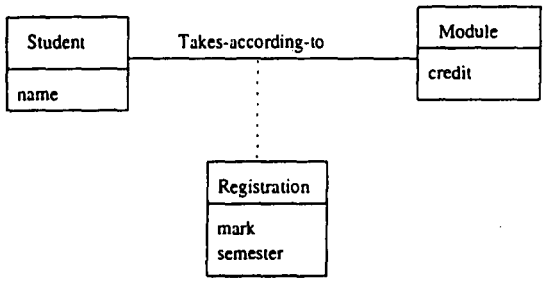


Figure 9.13: Associative class

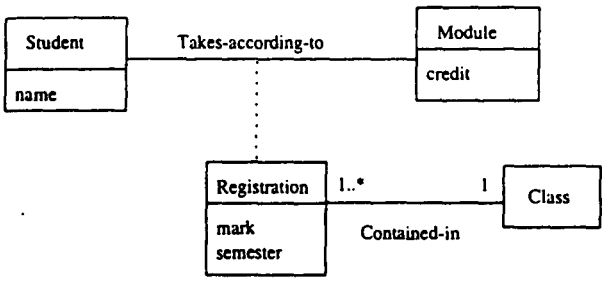


Figure 9.14: Associative class can be linked to another class

For the POST application, the following domain requirements need the use of an associative type:

- Authorization services assign a merchant ID to each store for identification during communications.
- A payment authorization request from the store to an authorization service requires the inclusion of the merchant ID that identifies the store to the service.
- Furthermore, a store has a different merchant ID for each service.

This is modelled by the class diagram in Figure 9.15.

Clues that an associative type might be useful in a conceptual model

- An attribute is related to an association.
- Instances of the associative type have a life-time dependency on the association.
- There is a many-to-many association between two concepts, and information associated with the association itself.

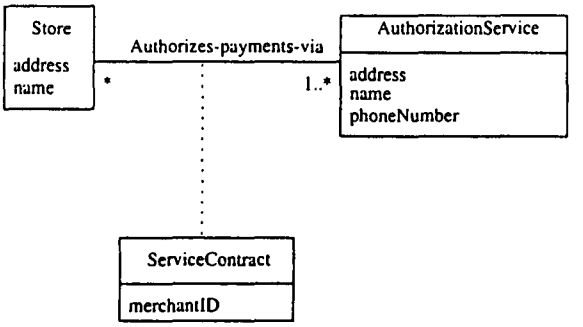


Figure 9.15: An associative class in the POST application

- Only one instance of the associative type exists between two objects participating in the association.

The presence of many-to-many association is a common clue that a useful associative type is lurking in the background somewhere; when you see one, consider an associative type.

9.3.2 Qualified associations

In Figure 9.16, a **qualifier** is used in an association to distinguish the set of objects at the far end of the association based upon the qualifier value. An association with a qualifier is a **qualified association.**

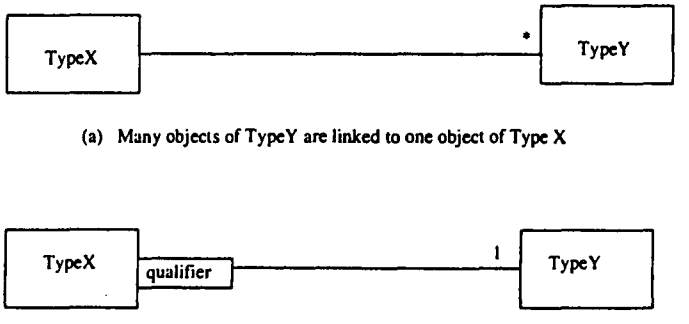


Figure 9.16: UML notation ualified association

For example, *ProductSpectifications* may be distinguished in a *ProductCatalog* by their UPC, as illustrated in Figure 9.17.

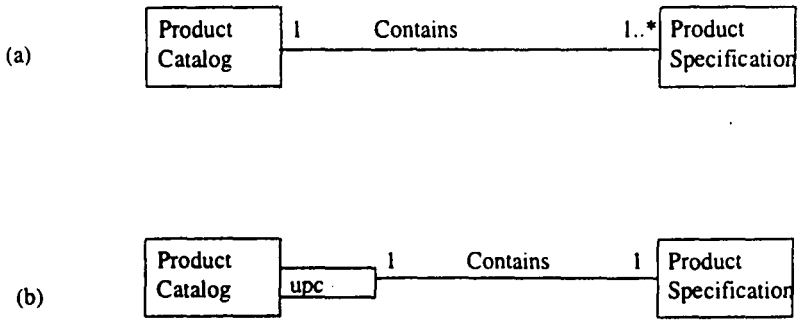


Figure 9.17: A qualifier in the POST application

We can see that the use of a qualifier in an association reduces the multiplicity at the far end from the qualifier, usually down from many to one. Depicting a qualifier in a conceptual model communicates how, in the domain, things of one type are distinguished in relation to another type. They should not, in the conceptual model, be used to express design decisions about lookup keys, although that is suitable in later design diagrams.

9.4 Packages: Organizing Elements

The conceptual model for development cycle two of the POST application is approaching an unwieldy size; it indicates the need to partition the model elements into smaller subsets.

Further, we would like to group the elements into packages to support a higher-level view.

In UML, a *package* is shown as a tabbed folder as shown in Figure 9.18.

- Subordinate packages may be shown within it.
- The package name is within the tab if the package depicts its elements, otherwise it is centered within the folder itself.

An element, such as a type or class, is *owned* by the package within which it is defined, but may be *referenced* in other packages. When an element is being referenced, the element name is qualified by the package name using the pathname format

PackageName :: *ElementName*

A type or a class in a foreign package may be modified with new associations, but must otherwise remain unchanged. This illustrated in Figure 9.19.

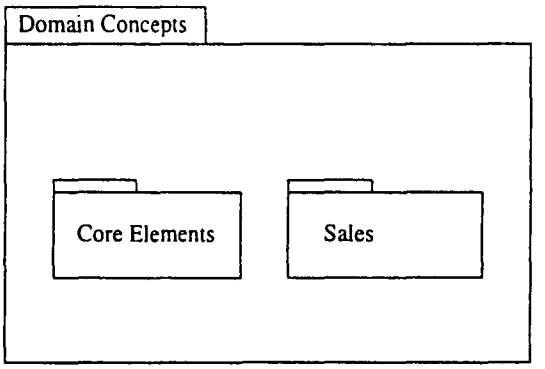


Figure 9.18: A UML package

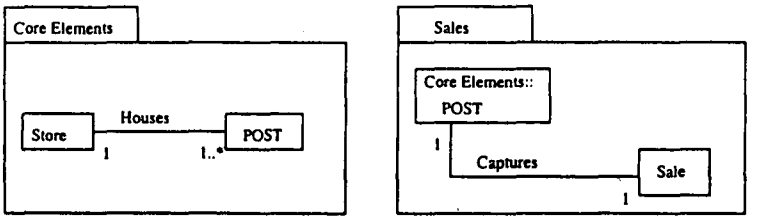


Figure 9.19: A referenced type in a package

If a model element is in some way dependent on another, the dependency may be shown with a dependency relationship, depicted with an arrowed line. A package dependency indicates that elements of the dependent package in some way know about or are coupled to elements in the target package.

For example, if a package references an element owned by another, a dependency exists. Thus *Sales* package has a dependency on the *Core Elements* package, see Figure 9.20.

How to partition the conceptual model into packages To partition the conceptual model into packages, place elements together that:

- are in the same subject area–closely related by concept and purpose
- are in a generalization-specialization hierarchy together
- participate in the same use cases
- are strongly associated.

It is useful that all elements related to the conceptual model be rooted in a package called *Domain Concepts*, and that widely shared, common, core concepts be defined in a package named something like *Core Elements* or *Common Concepts*.

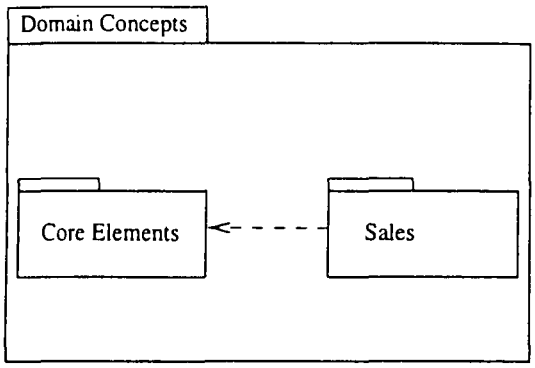


Figure 9.20: A package dependency

9.4.1 POST conceptual model packages

This section gives a package organization for the POST application.

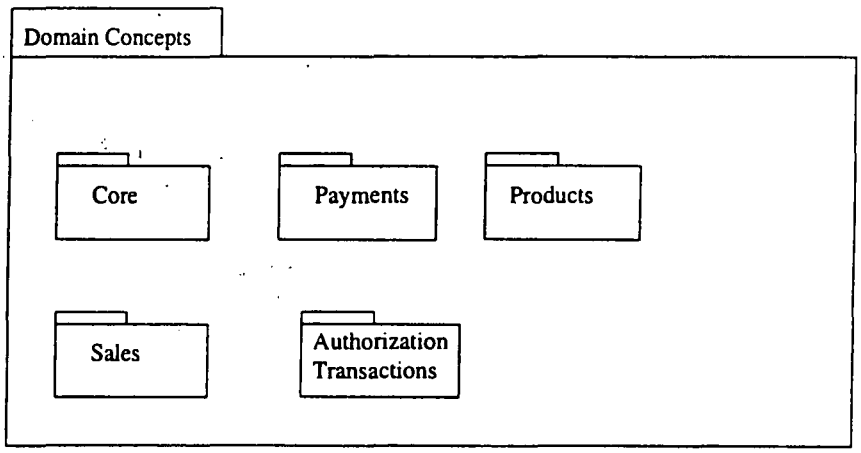


Figure 9.21: Domain concept package

Core package

Put the widely shared concepts or those without an obvious owner (or home) into the *Core/Misc* package.

Sales package

Except for the use of composite aggregation, there is no new concept, attribute or association added.

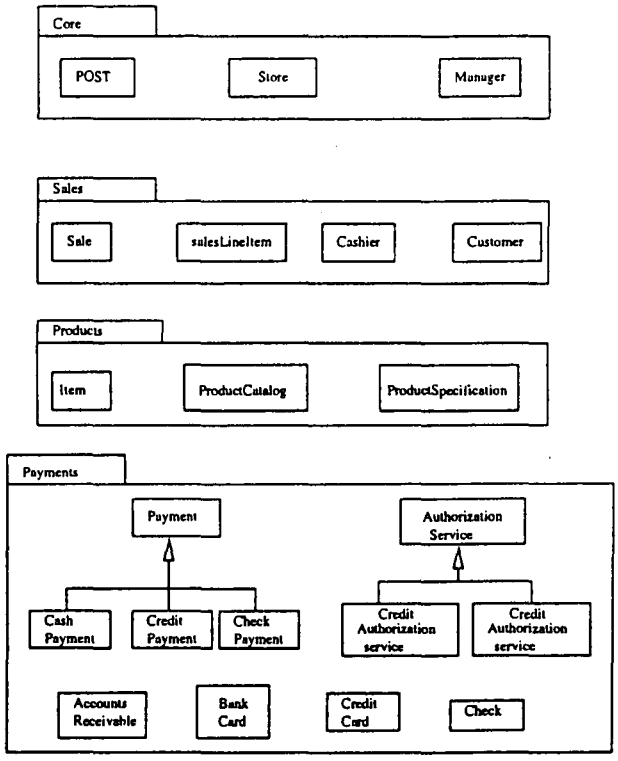


Figure 9.22: Domain package

Products package

With except of composite aggregation, there are no new concepts, associations or attributes added to this package.

Payments package

New concepts are itentified, new associations are primarily motivated by a need-to-know criterion. For example, there is a need to remember the relationship between a *CreditPayment* and a *CreditCard*. Some associations are also added for comprehension, such as *BankCard Identifies Customer*.

Note the use of the associative type *PaymentAuthorizationReply* - a reply arises out of the association between a payment and its authorization servatice.

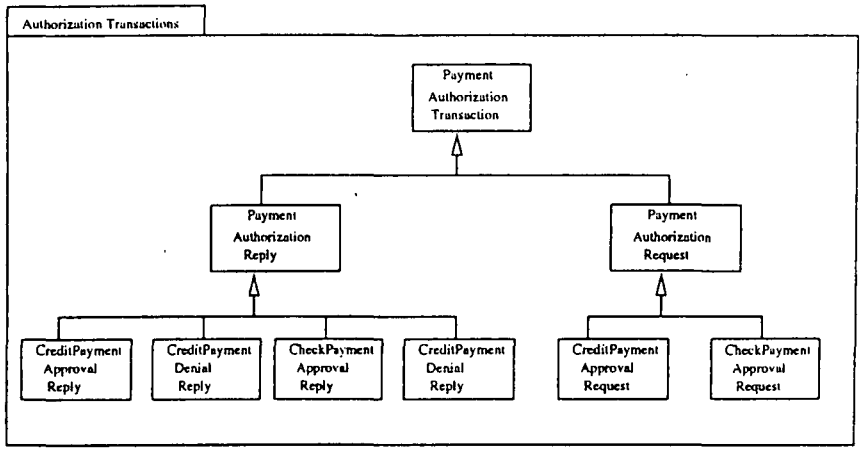


Figure 9.23: Another domain package

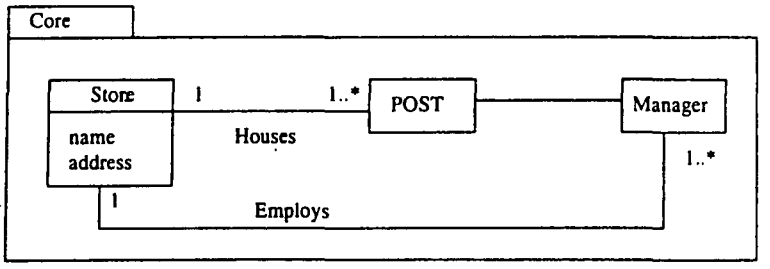


Figure 9.24: Core package

Authorization Transactions Package

Notice that the names of some associations are left unspecified as they can be easily understood: transactions are for the payment.

9.5 Modelling Behaviour in State Diagrams

A *state diagram* of an object shows

- all the messages that can be sent to the object,
- the order in which must be received,
- the effect of messages – what actions the object performs or what messages it sends to other objects after receiving a particular message.

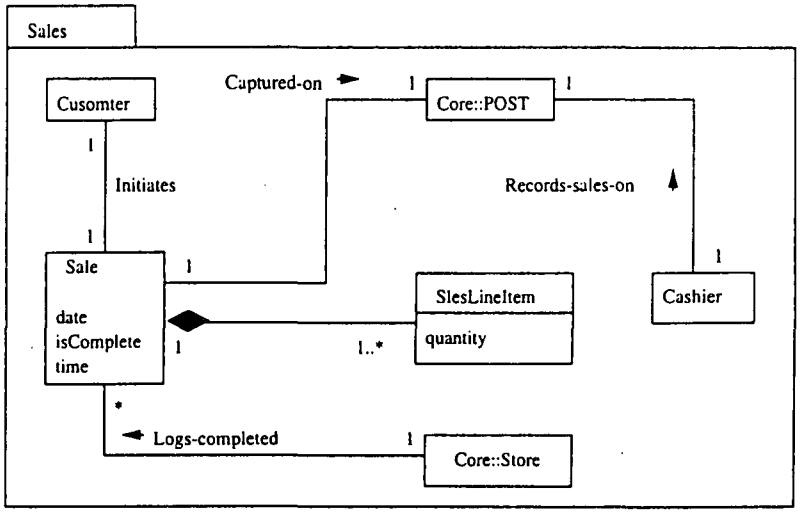


Figure 9.25: Sales package

A state diagram can also be used to describe the legal sequence of system input events in the context of a use case; and to define the behaviour, all the state transitions for the input events across all the use cases, of the system.

Events

- An **event** is a significant or noteworthy occurrence.
- In a software system, an object must be able to detect and respond to events.
- The events detected by an object are simply the messages sent to it.

Examples

1. Consider a CD player as a real-world object, which contains a drawer to hold the CD. Suppose the player provides buttons labelled 'load', 'play' and 'stop'.

   The events that the player can detect are simply the pressing of the three buttons.

   To respond to an event, such as 'load', the player will open the drawer if it is closed, and will close the drawer if it is open.

2. Consider a telephone. It can detect the event 'the receiver is taken off the hook'.

Sates

- A **state** of an object is the condition of the object at a moment in time – the time between events.

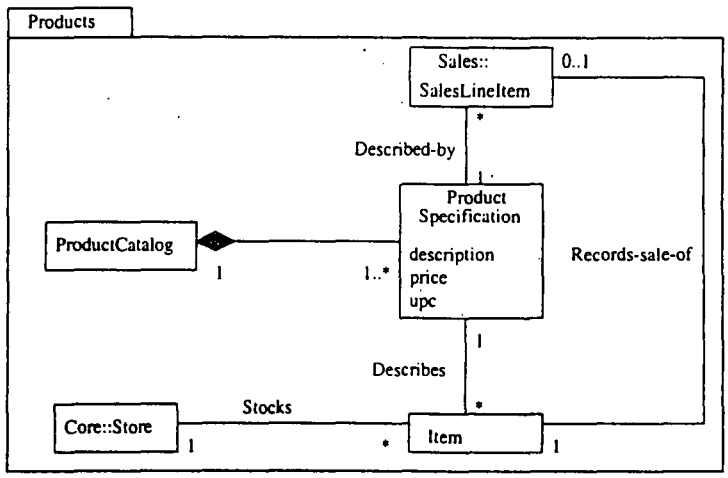


Figure 9.26: Products package

- The object being modelled is thought of as being in different states at different time.
- The defining property of states is that the state that an object is in can affect how it responds to particular events.
- The fact that an object is in different states can be identified by observing it responding differently to the same event at different time.

Examples of States

1. At any time, a telephone can be in one of the two states: *idle* and *active*:
   - It is in state *idle* after the receiver is placed on the hook until it is taken off the hook.
   - It is in state *active* after the receiver is taken off the hook until it is placed on the hook.
2. The CD player can be in one of at least two states: *drawerOpen* and *drawerClosed*:
   - When it is in *drawerOpen*, it closes the drawer after it detects 'load'.
   - When it is in *drawerClosed*, it opens the drawer after it detects 'load'.

   A third state *Playing* can be identified by considering the effect of the 'stop' event.

   Here, we need to clarify the meaning of *drawerClosed* and *Playing*.

State Transitions

- In general, detecting (receiving) an event can cause an object to move from one state to another. Such a move is called a **state transition**.

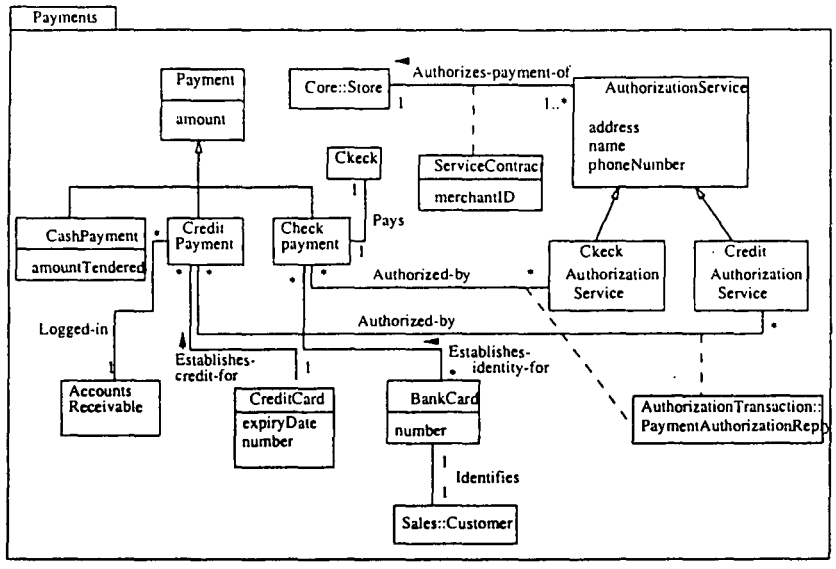


Figure 9.27: Payments package

- A **state transition** is a relationship between two states that indicates that when a event occurs, the object changes from the prior state to the subsequent state.
- The behaviors of an object is described in terms of it state transitions.

Examples of State Transitions

1. When the event "off hook" occurs, the state transition from state *idle* to state *active* is taken place.
2. If event "load" occurs when the CD is in *drawerOpen* state, the state transition from *drawerOpen* to *drawerClosed* is taken place.

UML Notation for State Diagram

- A UML **state diagram** illustrates the interesting events and states of an object, and the behaviour of an object in reaction to an event.
- State transitions are shown as arrows, labelled with their events.
- States are shown in rounded rectangles.
- It is common to include an initial pseudo-state which automatically transitions to another state when the instance is created.

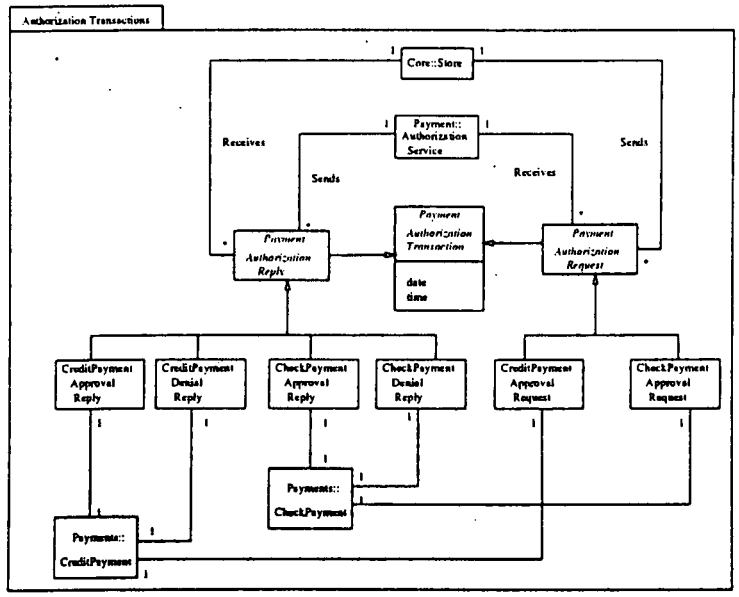


Figure 9.28: Payments package

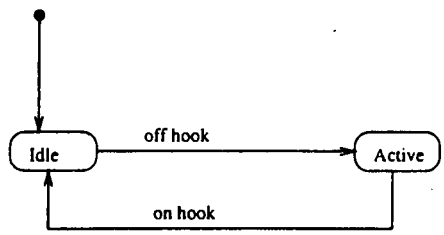


Figure 9.29: UML notation for state diagrams

This is shown in Figure 9.29

Example: CD Player Figure 9.30 gives a state diagram for the CD player.

A **final state** represents the state reached when an object is destroyed, or stops reacting to events.

Guards

- For the CD player, it is **not** correct that the player always goes into playing state when the play button is pressed – even without a CD in the drawer.
- It should do so only when there is a CD in the drawer. More precisely:
  - When the event 'play' occurs, the player goes into the *Playing* state if there is a CD in the drawer.

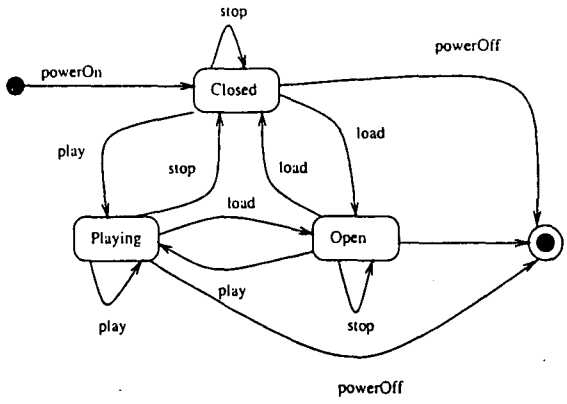


Figure 9.30: A sate diagram of the CD player

  - When the event 'play' occurs, they play goes to *drawerClosed* state if there is no CD in the drawer.

- This indicates the need of a *conditional guard* – or boolean test for a state transition.
- A guarded state transition can be taken only when the guard is true.

Examples of Guarded Transitions

Diagrams in Figure 9.31 show the UML notation for guarded transitions.

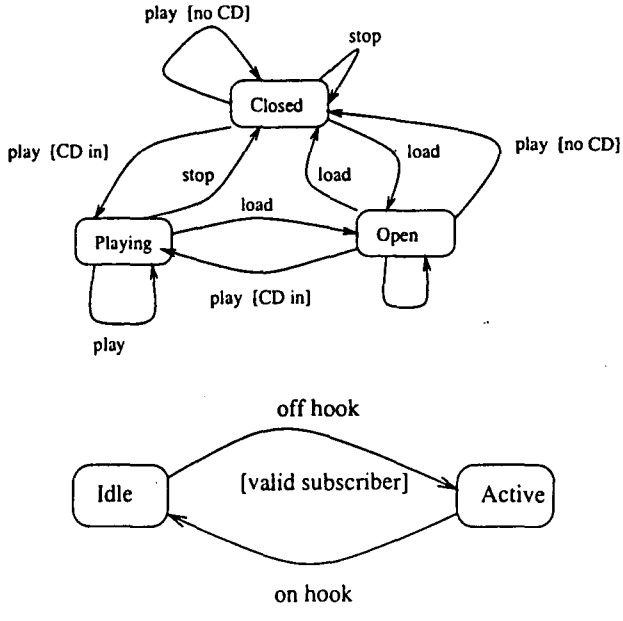


Figure 9.31: Guarded transitions

Actions

- A transition can cause an **action** to fire.
- In a software implementation, firing an action may represent the the invocation of a method of the class of the object of the state diagram.
- An action must be completed before the transition reaches the new state.
- An action cannot be interrupted by any other event that might be detected by the object – *actions are atomic.*

Examples of Actions are shown in Figure 9.32.

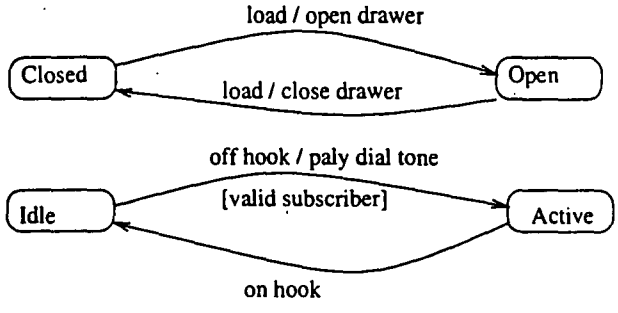


Figure 9.32: Examples of actions

Nested States

- A state allows nesting to contain *substates.*
- A substate inherits the transitions of its superstate (the enclosing state).
- Nested state can be used to refine a state diagram at a higher level of abstract into one with finer states.
- Nested states can be used for better modelling of the behaviour of an object.

Examples of Nested States are shown in the diagrams in Figure 9.33.

Use Case State Diagram

- A useful application of state diagrams is to describe the legal sequence of system input events in a use case.
- For example, consider the **Buy Items** use case

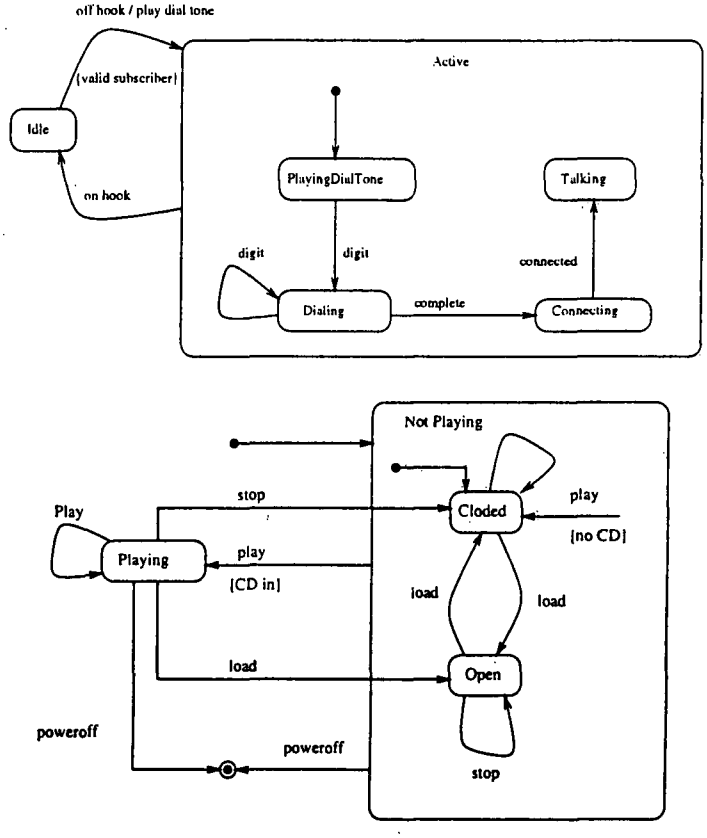


Figure 9.33: Nested states

- It is not legal to perform the *makeCreditPayment* operation until the *endSale* event has happened.

This can be illustrated in Figure 9.34.

A State Diagram for *POST*

- A *POST* instance reacts differently to the *EnterItem* message, depending on its stat as shown in Figure 9.35

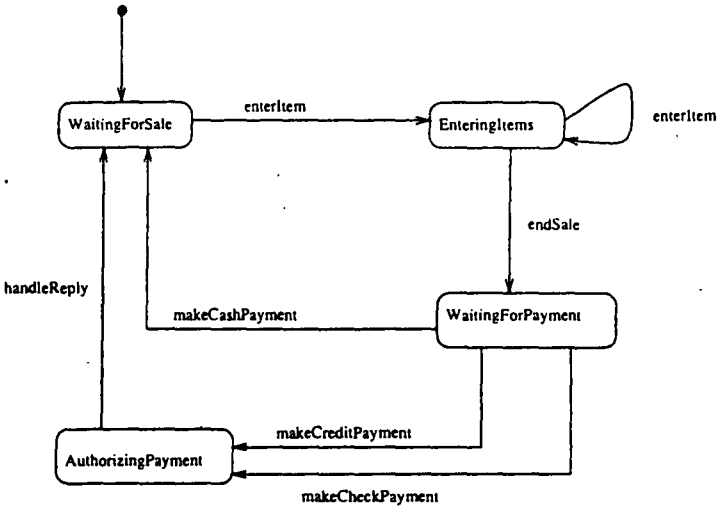


Figure 9.34: A sate diagram of Buy Items use case

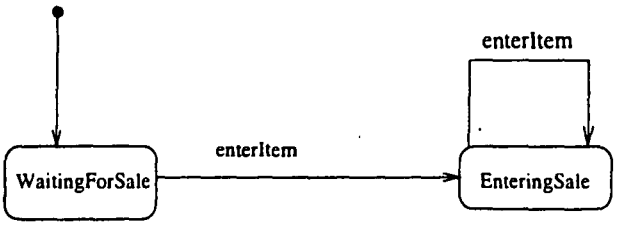


Figure 9.35: A state diagram for the *POST* object

Chapter 10

Summing Up

Topics of Chapter 10

- Summing up the materials
- The scope of the exam
- About the revision

We use the UML notation for package to summerize the materials of this course.

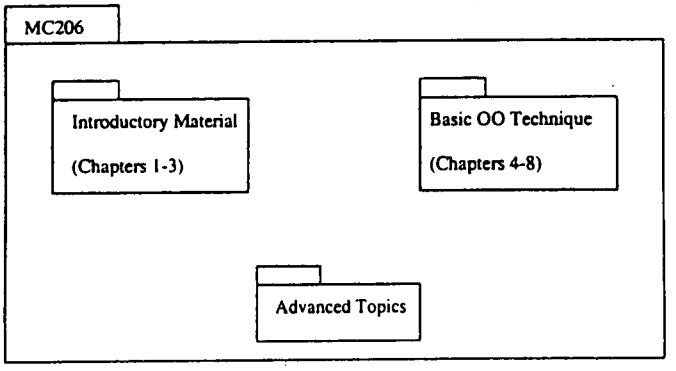


Figure 10.1: Three Parts of the Course

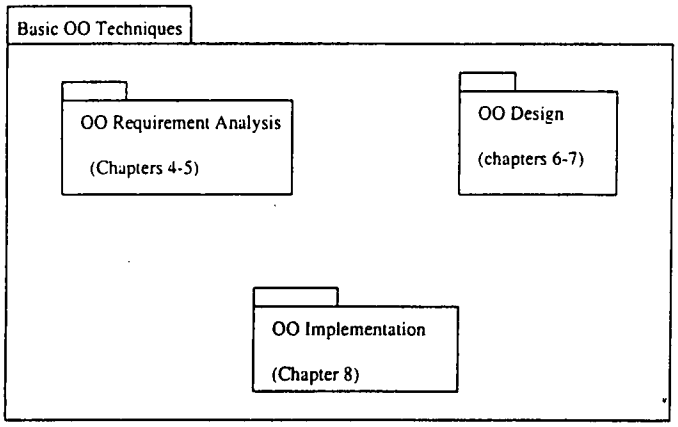


Figure 10.2: The Basic OO Techniques

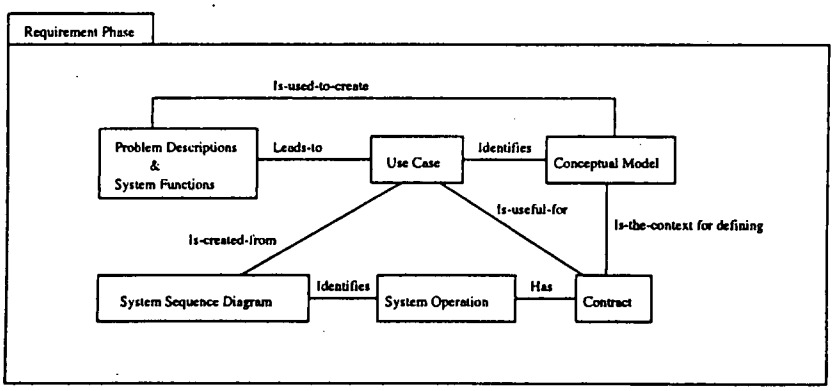


Figure 10.3: Requirement phase

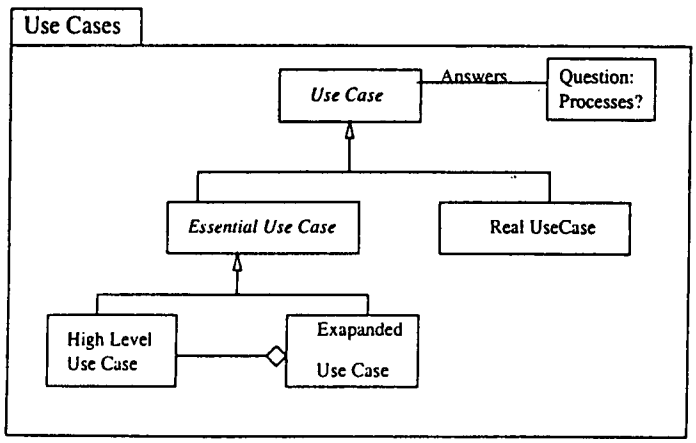


Figure 10.4: Use Cases

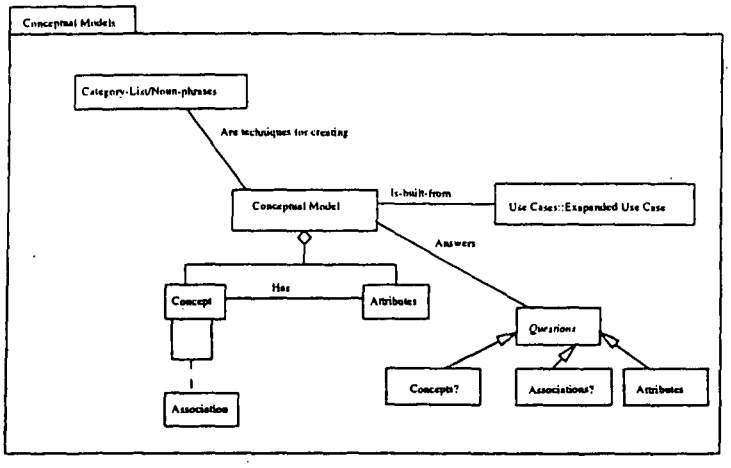


Figure 10.5: Conceptual Model

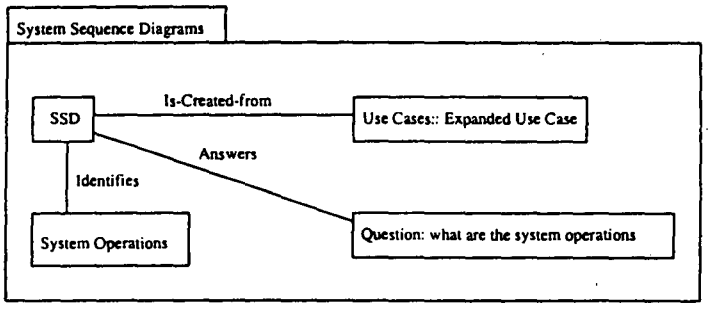


Figure 10.6: System sequence diagrams

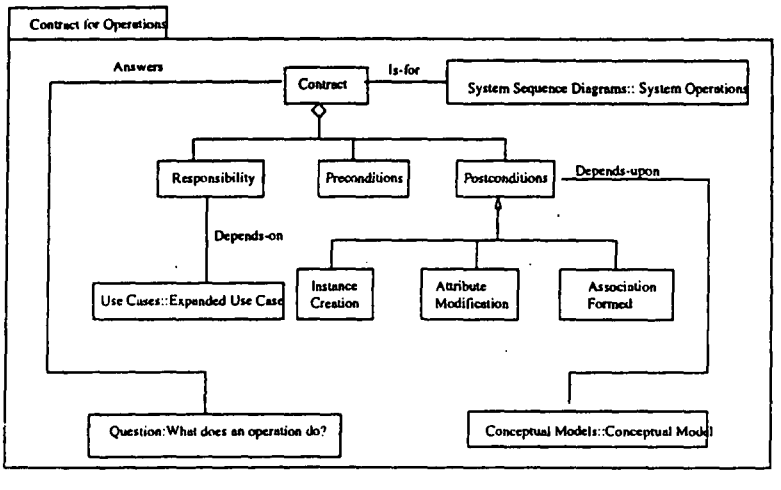


Figure 10.7: Contract

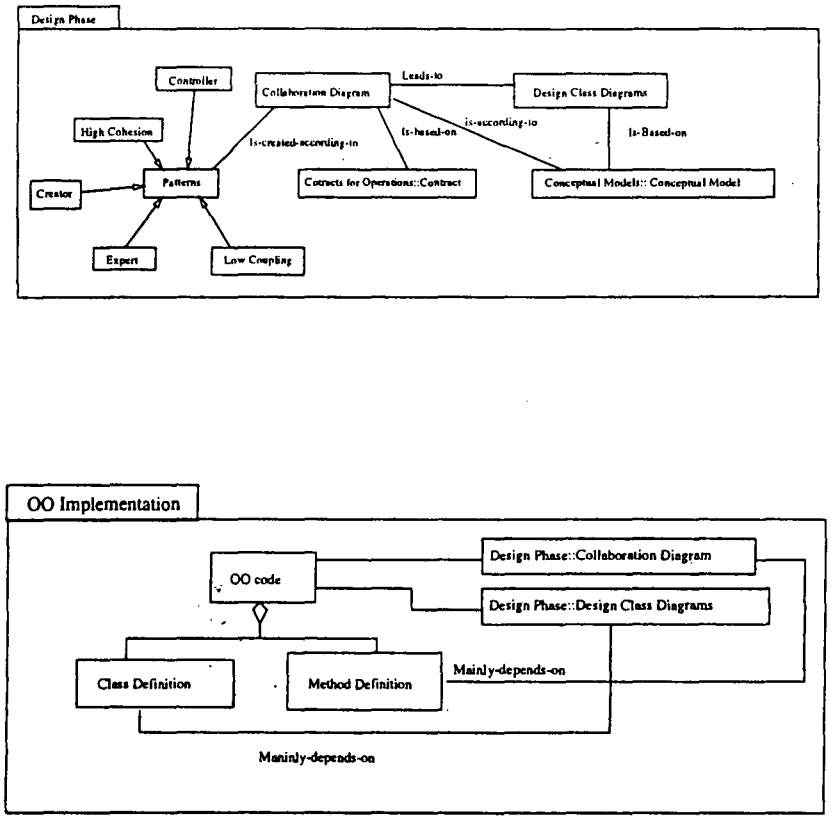


Figure 10.8: Design Phase

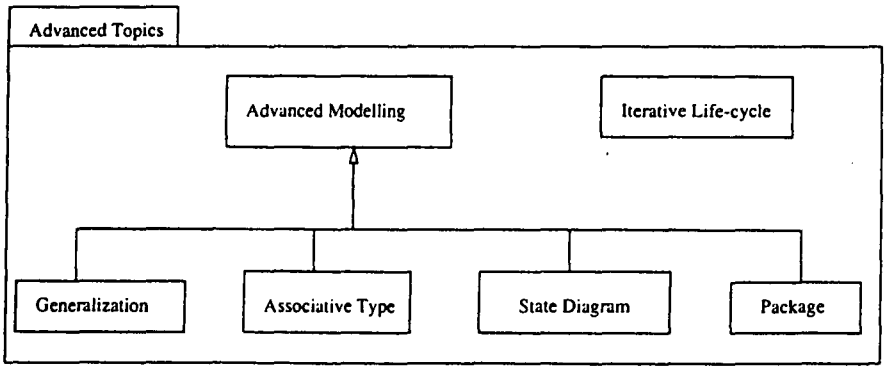


Figure 10.9: Advanced Topics