Object-Oriented Technology
By Tsang, Lau & Leung 2005 Mcgraw-Hill 2005

Chapter 2

What is a object	16
What is a Class and what are instances	18
Summary of UML notation	34
Structural analysis techniques	35
Domian modeling and Analysis process	39
Tricks and Tips in Structural analysis modeling and Analysis	54
Domain modeling and Analysis with VP-UML	55
Summary	70
Exercise	71

Chapter

2

Structural Modeling and Analysis

Overview

Structural modeling is concerned with describing "things" in a system and how these things are related to each other. A "thing" can be an object, a class, an interface, a package or a subsystem, which is part of the system being developed. For example, a class diagram can be used to describe the objects and classes inside a system and the relationships between them. The software components of a system in a component diagram can be described by providing details as to how these software components are deployed in terms of computing resources, such as a workstation.

Structural modeling is a very important process because it is employed throughout the entire system development life cycle. At the early analysis stage, a structural model is developed to describe the objects identified from the problem domain. As time progresses, the structural model is refined and new ones created in the process. Early versions of a structural model are usually incomplete, and as such are refined iteratively and incrementally. System implementation commences only when the structural model contains sufficient details.

What You Will Learn

On completing the study of this chapter, you should be able to:

- describe and apply the fundamental object-oriented concepts
- use the standard Unified Modeling Language (UML) notation to represent classes and their properties

- model the structural aspects of problems with the class model
- perform domain analysis to develop domain class models

What Is an Object?

An object is a self-contained entity with well-defined characteristics (properties or attributes) and behaviors (operations). For example, the real-life environment consists of *objects* such as schools, students, teachers and courses which are related in one way or another. A student has a name and an address as its characteristics. Similarly, a subject has a title and a medium of instruction as its characteristics.

An object generally has many *states*, but it can only be in one state at a time. The state of an object is one of the possible conditions in which an object may exist. The state is represented by the values of the properties (attributes) of an object. In different states, an object may exhibit different *behaviors*. For example, in the awake state, a person may have behaviors such as standing, walking or running, while in the sleeping state, the person may have behaviors such as snoring or sleepwalking. For objects such as a human being or an automobile, a complete description of all the states of these objects can be very complex. Fortunately, when objects are used to model a system, we typically focus on all the possible states of the objects that are relevant only to and are within the scope of that system.

The behavior of an object relates to how an object acts and reacts. An object's behaviors are also known as *functions* or *methods*. The behavior is determined by a set of operations that the object can perform. For example, through the physical interface of the VCR system, functions like play, rewind and record can be performed, while simultaneously changing the state of the system.

Types of Objects

Physical and Conceptual Objects

Objects can be broadly classified as *physical* or *conceptual objects*, and they are things that we find around us in the real world. We interact with physical and conceptual objects all the time. In software development, real-life objects are naturally mapped onto objects of a software system.

Physical (tangible) objects are visible and touchable things such as a book, a bus, a computer or a Christmas tree. In an automated teller machine (ATM), the card reader and the receipt printer are examples of physical objects.

Conceptual objects are intangibles such as a bank account and a time schedule. Very often, conceptual objects are thought of as physical objects. For example, we would normally say we pay the mortgage (conceptual object) every month, instead of saying we pay the bankbook (where the money is deposited). We mix conceptual objects and physical objects all the time as they are well understood within the context. Some of these concepts may only be understood within a small society or even within a group of domain experts. The object designer, therefore, needs to talk to the domain experts to gain the necessary domain knowledge so that they can use the objects, concepts and terminologies that are well understood by the people working in that domain.

Domain and Implementation Objects

The beauty of object-orientation is that different software engineers are likely to identify similar sets of domain objects for the same area of application because of the natural mapping of real-world entities to objects. The objects identified from the real world are *domain objects*. Collectively, we call all objects which are not related to real-world entities as *implementation objects*. For example, bank accounts, tellers and customers are domain objects that we come across daily. On the other hand, the transaction log which provides information for error recovery is obviously an implementation object.

Domain objects tend to be more stable throughout the development life cycle as the latter is unlikely to incur a major change in the specification of the domain objects since these objects form the foundation (architecture) of a software system. On the other hand, implementation objects are more likely to change when the requirements are altered. For example, bank accounts, customers and banks are domain objects in an ATM system. Most software designers can identify a similar set of domain objects. In contrast, they have greater flexibility in choosing the implementation objects in order to satisfy the implementation constraints, such as performance and usability.

Active and Passive Objects

An object can be *active* or *passive*. It is necessary to distinguish between active objects and passive objects because they require different strategies for implementation. An active object is an object that can change its state. For example, timers and clocks can change their states without an external stimulus. Active objects are usually implemented as processes or threads, which are also referred to as “objects with life.” With a passive object, the state of an object will not change unless the object receives a message. For example, the properties of a bank account will not change unless the bank account receives a message such as set balance (an operation for updating the balance of an

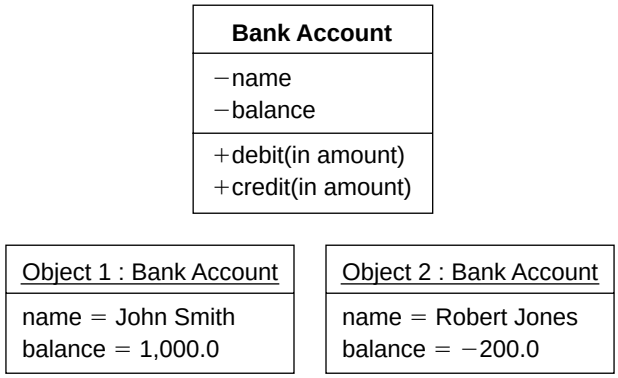
account). Because the majority of objects are passive, sometimes it is automatically assumed that all objects are strictly passive.

What Is a Class and What Are Instances?

A *class* is a generic definition for a set of similar objects. It is an abstraction of a real-world entity that captures and specifies the properties and behaviors that are essential to the system but hides those that are irrelevant. The class also determines the structure and capabilities of its *instances* (objects). Thus, a class is a template or blueprint for a category of structurally identical items (objects). Objects are instances of a class. In other words, a class is like a mold and an instance of a class is like a molded object.

It is very important to understand the differences between classes and instances in order to get to grips with this chapter. A class has methods and attributes while object instances have behaviors and states. This concept is illustrated in Figure 2.1. In this example, *bank account* is a class. *Bank account* is a generic term that covers many different account types. John's and Robert's accounts are instances of the *bank account* class. Although their accounts are of a type of *bank account* and are not generic.

Figure 2.1. UML notation for objects and classes



The *bank account* class specifies that a *bank account* object has *name* and *balance* as its private properties (indicated by a "−" sign) and public credit and debit operations (indicated by a "+" sign). It is noteworthy that the two instances are in different states. John's account is in the credit state (positive balance), while Robert's account is in the overdrawn state (negative balance). The state of the objects can be changed by calling the *credit* or *debit* operations, e.g. Robert's account can be changed to the credit state if a *credit* operation is invoked with a parameter of, say, 300.

Attributes

Things in the real world have *properties*. An *attribute* is a property of a class. Other words for attribute include "property," "characteristic" and "member data." For example, a book can be described in terms of its author, ISBN (International Serial Book Number), publisher, among others. More properties can be associated with the class *book* such as the number of pages, its weight, physical dimensions and so on. The abstraction of a book is limited to a specific problem domain so that the number of required properties can be reduced. For example, information on the weight and dimensions may be required for a delivery company but totally irrelevant to an information system of a bookstore.

From a human perspective, a property is a characteristic that describes an object. From a technical perspective, an attribute is a data item where an object holds its own state information. In summary, attributes have a name and a value, and attributes may also have a type, e.g. "integer," "Boolean."

Operations

Each object can perform a set of *functions* in order to provide a number of services in a software system. This is similar to the situation in a company where each member of staff provides a set of services to other members and customers. An object calls another object's service by sending it a message. A *service* is defined by one or more operations, and an *operation* is a function or a procedure which can access the object's data. An operation consists of two parts: a name and argument(s). Thus, every object must define its operations for each of its services. The collection of operations is the object's interface. Other objects only need to know the interface of an object in order to invoke the operations provided by the object.

An operation is sometimes called a *method* or a *member function*. These two terms are more widely used by programmers than designers. To a programmer, an operation is like a function (or procedure). The *return value* is the result that an operation "brings back" on completion. This is a useful way of allowing other objects to find out a piece of information about an object. In programming language, operations are similar to functions in that they have parameters and return values. For example, the savings account class in an ATM banking system may have the following operations:

- *withdraw*(*amount*)
- *deposit*(*amount*)
- *getBalance*()

Encapsulation: Information Hiding

Objects are like black boxes. Specifically, the underlying implementations of objects are hidden from those that use them. This is a powerful concept called information hiding, better known as the *encapsulation* principle. In object-oriented systems, it is only the producer (creator, designer or developer) of an object that knows the details of the internal construction of that object. The consumers (users) of an object are denied knowledge of the inner workings of the object and must deal with an object via one of its three distinct interfaces:

- Public interface which is open (visible) to everybody.
- Protected interface which is accessible only by objects that have inherited the properties and operations of the object. In class-based, object-oriented systems, only classes can provide an inheritance interface. (Inheritance and specialization will be discussed later).
- Parameter interface. In the case of parameterized classes, the parameter interface defines the parameters that must be supplied to create an instance. For example, a linked list of objects may have a parameter that specifies the type of object contained in the linked list. When the linked list is used, the actual type of object can be provided.

Structural Modeling Techniques

In UML, a class is simply represented by a rectangle divided into three compartments, containing, from top to bottom, the class name, a list of attributes and a list of operations (see Figure 2.2). Each attribute name may be followed by optional details such as a type and a default value. Each operation may be followed by optional details such as an argument list and a result type. In most cases, the bottom two compartments are omitted, and even when they are present, they typically do not show every single attribute and operation. Typically, only those attributes and operations that are relevant to the current context will be shown in a diagram. We can also specify the accessibility of an element (an attribute or an operation) by prefixing its name by a "−," "+," or "#" sign. The "−," "+," and "#" signs respectively indicate that an element is private, public or protected.

Figure 2.2. Classes providing different levels of details

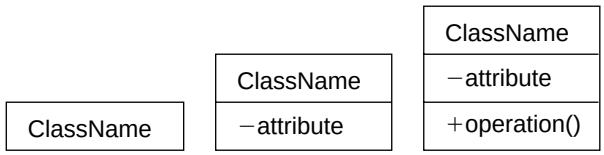


Figure 2.3 shows how a class is represented in the UML notation. Classes and objects are distinguished by underlining the object name and optionally followed by the class name.

Figure 2.3. UML notation for classes

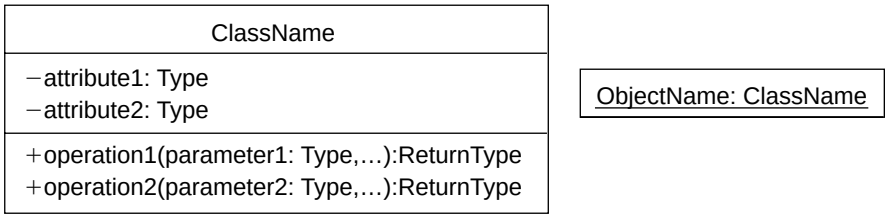
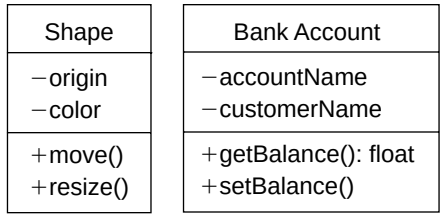


Figure 2.4 shows two examples of classes. In the first example, the *Shape* class has *origin* and *color* as attributes, with *move* and *resize* as operations. In the second example, the *bank account* class has *account number* and *customer name* as attributes and performs *get balance* and *set balance* operations.

Figure 2.4. Examples of classes in UML



Naming Classes

It is common practice to name a class with a noun or a noun phrase, but there are no firm rules on naming the elements (classes, attributes, etc.) of class models. The system development team should decide when and where upper case letters and spaces should be used, and it is important that all members of the team stick to the team's decision. When using name phrases, a widely used convention is to eliminate spaces and concatenate the words with their first letters in upper case, e.g. *SavingsAccount* and *BankAccount*.

Relationships between Classes

Relationships exist among real-life objects. For example, friendship and partnership are common examples of relationships among people. Similarly, a relationship specifies the type of *link* that exists between objects. Through the

links of an object, it is possible to discover the other objects that are related to it. For example, all the friends of a person John can be determined through the links to John.

Finding relationships between classes is an important part of object-oriented modeling because relationships increase the usefulness of a model. Identifying relationships can help find new classes and eliminate bad ones. Furthermore, it may lead to the discovery of relevant attributes and operations.

There are essentially three important types of relationships between classes: *generalization/specialization* ("type-of"), *aggregation* ("part-of") and *association* relationships.

Inheritance

Object-oriented programming languages facilitate *inheritance* that allows the implementation of generalization-specialization associations in a very elegant manner. Attributes and operations common to a group of subclasses are attached to a superclass and inherited by its subclasses; each subclass may also include new features (methods or attributes) of its own. Generalization is sometimes called the "is-a" relationship. For example, *checking accounts* and *savings accounts* can be defined as specializations of *bank accounts*. Another way of saying this is that both a *checking account* and a *savings account* "is-a" kind of a *bank account*; everything that is true for a *bank account* is also true for a *savings account* and a *checking account*.

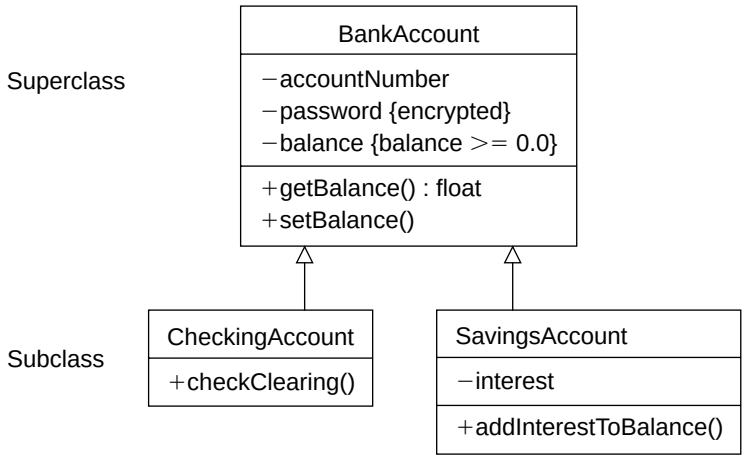
Properties of Inheritance

Generalization

The purpose of this property is to distribute the commonalities from the superclass among a group of similar subclasses. The subclass inherits all the superclass's (base class) operations and attributes. That is, whatever the superclass possesses, the derived class (subclass) does as well. Taking the *BankAccount* example from above, if *BankAccount* (superclass) has an *account_number* attribute, the *CheckingAccount* (subclass) will also have the same attribute, *account_number*, as it is a subclass of *BankAccount*. It would be unnecessary and inappropriate to show the superclass attributes in the subclasses. Similarly, suppose there is a bank application for an ATM machine. If *BankAccount* has the operation *setBalance*, then *SavingsAccount* will automatically inherit this operation as well. It would be a mistake to duplicate the attributes and operations in the superclass in its subclasses as well unless those operations have different implementations of their own. Figure 2.5 illustrates the concept of generalization, with *CheckingAccount* and

SavingsAccount inheriting their superclass's (*BankAccount*) attributes and operations.

Figure 2.5. *BankAccount* and its subclasses



Specialization

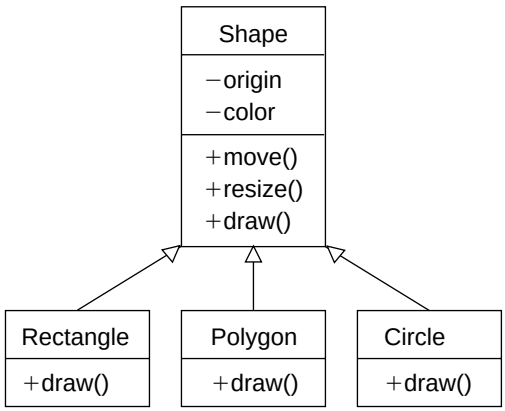
Specialization allows subclasses to extend the functionalities of their superclass. A subclass can introduce new operations and attributes of its own. For example, in Figure 2.5, *SavingsAccount* inherits attributes *account_number*, *password* and *balance* from *BankAccount* and extends the functionalities of *BankAccount* with an additional attribute, *interest*, and an additional operation, *addInterestToBalance*. A *SavingsAccount* has the attribute *interest* that *BankAccount* does not because not all bank accounts earn interest.

Abstract Classes

An *abstract* class is used to specify the required behaviors (operations) of a class without having to provide their actual implementation. An operation without the implementation (body) is called an *abstract operation*. A class with one or more abstract operations is an abstract class. An abstract class cannot be instantiated because it does not have the required implementation of the abstract operations. An abstract class can act as a repository of shared operation signatures (function prototypes) for its subclasses and so those methods must be implemented by subclasses according to the signatures. A class (or an operation) can be specified as abstract in the UML by writing its name in italics, such as for the class *Shape*. Here, the class *Shape* is abstract because we cannot draw a shape; we can only draw its subclasses such as

rectangles, circles, etc. Figure 2.6 shows an example of the abstract class *Shape* and its subclasses. The subclasses provide the actual implementations of their draw operations since *Rectangle*, *Polygon* and *Circle* can be drawn in different ways. A subclass can override the implementation of an operation inherited from a superclass by declaring another implementation (body of the operation). In the example, the draw operation in the *Rectangle* class overrides the implementation of the draw operation inherited from the *Shape* class.

Figure 2.6. *Shape* as an example of an abstract class



Polymorphism

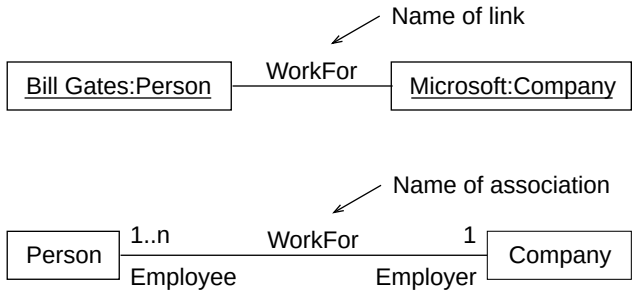
Polymorphism is the ability for a variable to hold objects of its own class and subclasses at runtime. The variable can be used to invoke an operation of the object held. The actual operation being invoked depends on the actual class of the object that is referenced by the variable. For example, suppose the variable "shape" is of type *Shape*. If shape references a *Rectangle* object, then *shape.draw*() invokes the *draw*() method of the *Rectangle* class. Similarly, if shape references a *Polygon* object, the *draw*() method of the *Polygon* class is invoked by *shape.draw*().

Association

Object-oriented systems are made up of objects of many classes. Associations represent binary relationships among classes. An *association* is represented by a line drawn between the associated classes involved with an optional role name attached to either end. The role name is used to specify the role of an associated class in the association. If an association connects between two objects instead of classes, it is called a *link*. A link is an instance of an association. For example, Figure 2.7 illustrates the *WorkFor* relationship

between the *Person* and *Company* classes. The relationship carries the meaning of "a person works for one company." Figure 2.7 illustrates that Bill Gates works for Microsoft and that many people can work for a company.

Figure 2.7. Association and link



Links provide a convenient way to trace the relationship between objects. However, do not spend too much time trying to identify all possible relationships between classes, as the implementation of these association relationships adds to the overheads in your system. Only specify those relationships that are necessary for the requirements of the system, and focus on questions such as: "While you are operating on one object, do you need to know the information of another object(s)?"

Role

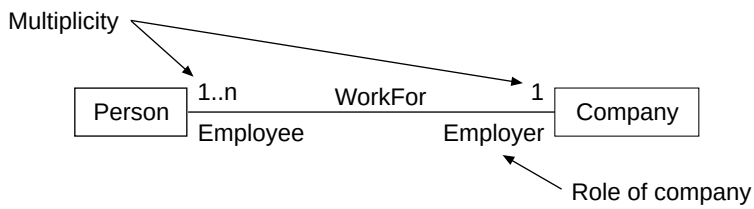
Each end of an association has a *role*. You may optionally attach a role name at the end of an association line. A role name uniquely identifies one end of an association. For example, the role of a person in the *WorkFor* relationship is employee and the role of a company is employer (See Figure 2.7). From the object's point of view, tracing the association is an operation that yields an object or a set of related objects at the other end of the association. For example, the employees of Microsoft can be determined by following the *WorkFor* association. During the analysis stage, an association is often considered to be bi-directional, that is, tracing can be done from either end of the association. However, during the design stage, only one direction may be needed to implement the requirements of the system.

Multiplicity

Multiplicity refers to the number of objects associated with a given object. For the *WorkFor* association in Figure 2.8, a person works for one and only one company since the multiplicity on the *Company* side is 1. On the other hand,

a company may have one or more persons working in it. If the multiplicity is not explicitly specified, the default value of 1 is assumed.

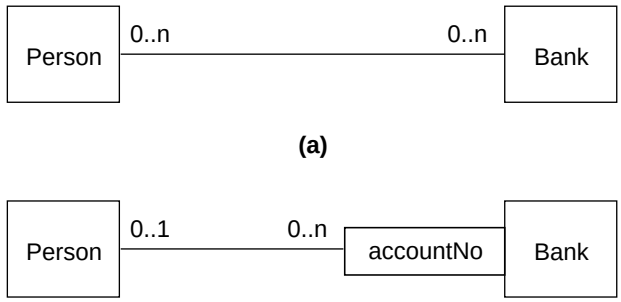
Figure 2.8. Association and role



Qualification

Qualification serves as names or keys that are part of the association and are used to select objects at the other end of the association. Qualification reduces the effective multiplicity of the association from one-to-many to one-to-one. In UML, a qualifier is used to model this association semantic, that is, an association attribute whose value determines a unique object or a subset of objects at the other end of the association. For example, a bank is associated with many customers. The *account number* (qualifier) specifies a unique person of a bank (a customer) (see Figure 2.9).

Figure 2.9. (a) Many-to-many association between *Person* and *Bank* and (b) reduced to a one-to-many association

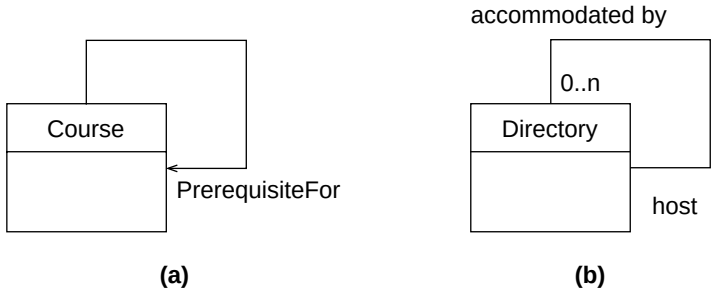


Reflexive Association and Roles

A *reflexive* association is an association that relates one object of a class to another object of the same class. In other words, a class can be associated with itself. There are two types of reflexive associations, namely, directional and bi-directional.

Figure 2.10a a shows an example of a directional reflexive association where the class *Course* is associated with itself. Here, a course may be a prerequisite for another course. Figure 2.10b shows an example of a bi-directional reflexive association where a parent directory (role: host) contains zero or more subdirectories (role: accommodated by).

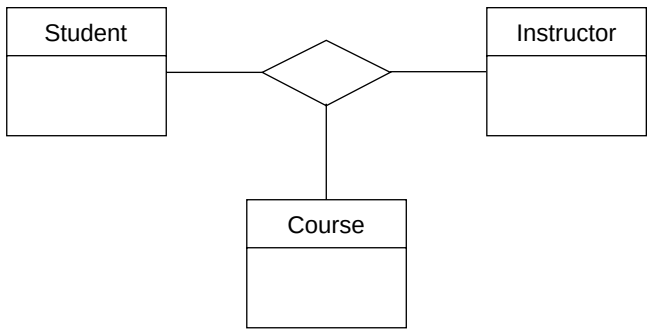
Figure 2.10 (a) A directional reflexive association and (b) a bi-directional reflexive association



N-ary *Association*

Associations are often binary, but higher order associations are also possible. A relationship involving three classes is referred to as a *ternary* relationship, and one involving many classes is referred to as an *n*-ary relationship. An *n*-ary association is represented by a diamond connecting the associated classes. In Figure 2.11, for example, a *Student* that takes a *Course* taught by a particular *Instructor* exhibits a ternary relationship.

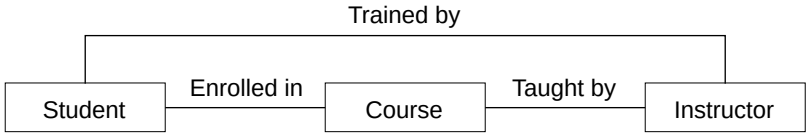
Figure 2.11. A ternary association



When modeling association relationships among classes, binary association is the most preferred form. A higher-order association can always be decomposed into a corresponding number of binary associations, and it is possible to convert some of the bi-directional relationships into unidirectional relationships in our class model during the design phase. For example, we can represent the ternary association in Figure 2.12 as three binary associations instead:

- a *Student* enrolls a *Course*
- an *Instructor* teaches a *Course*
- an *Instructor* trains a *Student*

Figure 2.12. Three binary associations replacing a ternary association



Association Classes

It is sometimes necessary to describe an *association* by including additional attributes which do not naturally belong to the objects involved in the association. In Figure 2.13, for example, the year of the enrollment of a student in a course does not belong to the *student* or *course* classes. In this case, an association class *Enrollment* is added to hold the attribute year.

There are situations where an association is complex enough to be a class in its own right. The association has its own class name and may have operations just like any other ordinary class. In the example of the association between a *Person* and a *Company* (Figure 2.14), the *Position* class contains the attributes of the association between the *Person* and the *Company*. The *Position* class has attributes of its own that do not naturally belong to *Person* or *Company*. Therefore, it is only natural or beneficial that the information belonging only to the object is contained in a separate class so as to maximize the level of module cohesion. It may sometimes be possible to transfer the attributes from the *Position* class to the *Person* or *Company* class. However, this move significantly affects the reusability of those classes, as the association attributes may be meaningful only in a specific context but not others. For example, in Figure 2.14, the *Person* class may well be suitable for other applications that do not need to know the *Position* information. Furthermore, if the *Position* class information in transferred to either the *Person* or *Company* class, it will rule out the possibility that a *Person* may have more than one *Position* with the same or another *Company*.

Figure 2.13. An association class

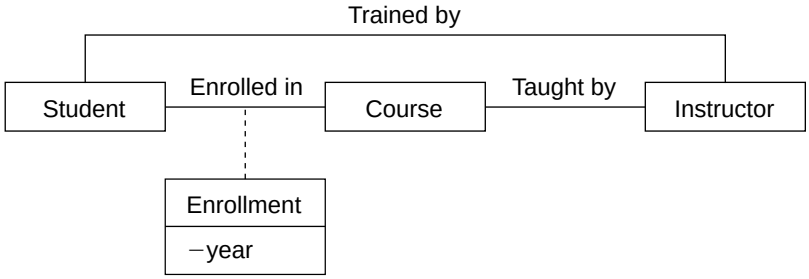
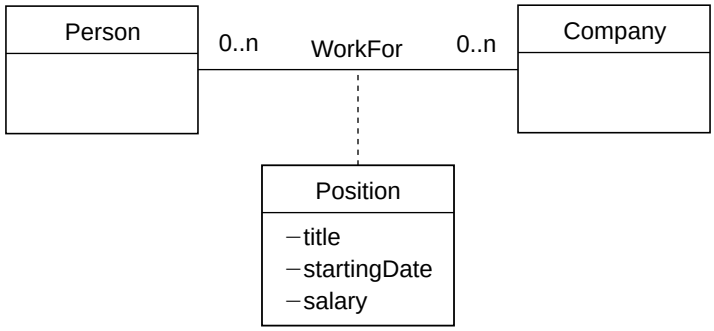


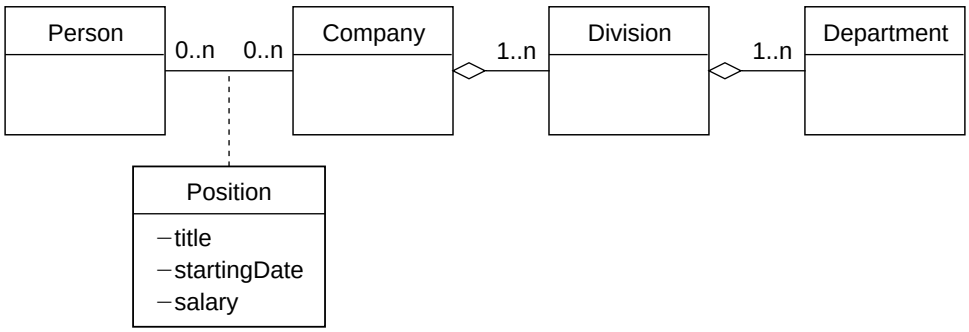
Figure 2.14. Using an association class



Aggregation

Aggregation is a stronger form of association. It represents the has-a or part-of relationship. In UML, a link is placed between the "whole" and "parts" classes, with a diamond head (see Figure 2.15) attached to the *whole* class to indicate that this association is an aggregation. Multiplicity can be specified at

Figure 2.15. Example of aggregation

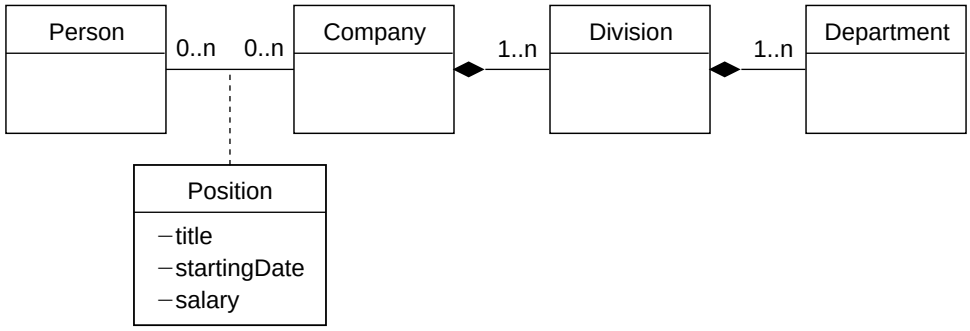


the end of the association for each of the "part-of" classes to indicate the quantity of the constituent parts. Typically, aggregations are not named, and the keywords used to identify aggregations are "consists of," "contains" or "is part of."

Composition

A stronger form of aggregation is called *composition*, which implies exclusive ownership of the "part-of" classes by the "whole" class, i.e. a composite object has exclusive ownership of the parts objects. This means that parts may be created after a composite is created, but such parts will be explicitly removed before the destruction of the composite. In UML, a filled diamond (see Figure 2.16) indicates the composition relationship. In Figure 2.15, it is more natural (closely resembling scenarios in the real world) for *Division*(s) and *Department*(s) to be created after the *Company* is set up and they will not exist if the *Company* closes down.

Figure 2.16. Example of composition



Constraints and Notes

Constraints are an extension of the semantics of a UML element that allow the inclusion of new rules or the modification of existing ones. It is sometimes helpful to present an idea about restrictions on attributes and associations for which there is no specific notation. Simply write them in braces near the class concerned. Constraints are represented by a label in curly brackets ({constraintName} or {expression}) that are attached to the constrained element. In the ATM banking example (see Figure 2.17), the *password* of a *bank account* is encrypted and the *balance* is not less than $0.

You can specify constraints for two associations such as {for}, {or}, {subset}, etc. Such constraints are called *complex constraints*. The {or} constraint

indicates that only one of the associations can exist at any given time. The {subset} constraint indicates that an association is a subset of another.

Figure 2.17. Example of constraints

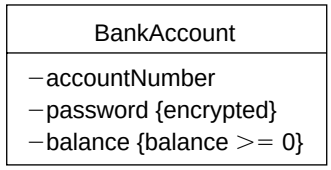
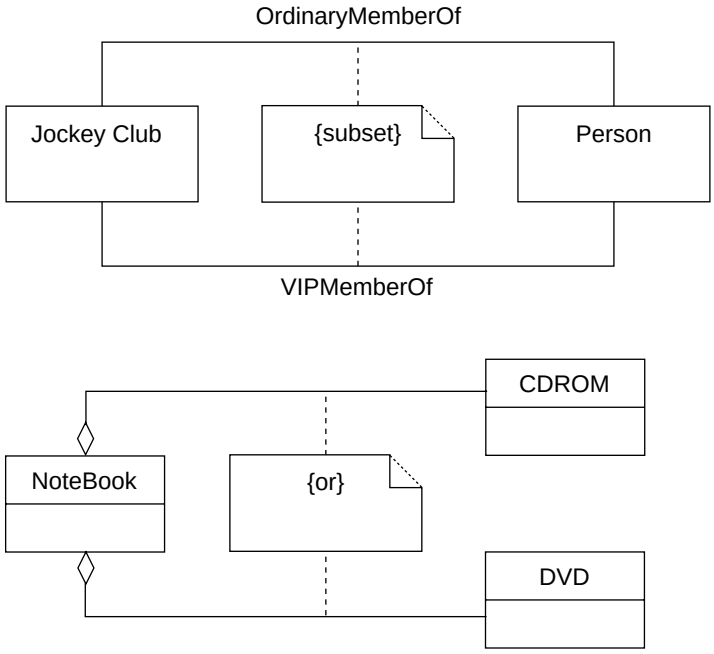


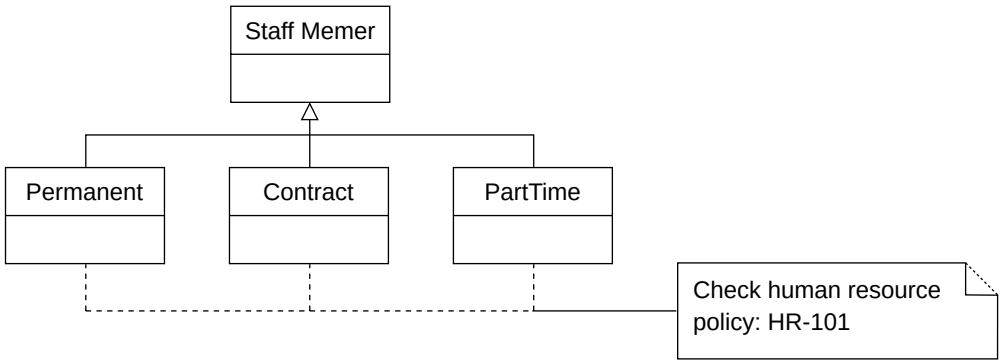
Figure 2.18 shows two examples of complex constraints. In the first example, the Jockey Club has two kinds of members: ordinary members and VIP members. The *VIPMemberOf* association is a subset of the *OrdinaryMemberOf* association. In other words, a VIP member is also an Ordinary member. In the second example, a *Notebook* computer has either a *CDROM* or *DVD* association but not both.

Figure 2.18. Complex constraints



A *note*, represented by a dog-eared rectangle in UML, is a graphical symbol for holding constraints or comments attached to an element or a collection of elements. A note can also be used to link or embed other documents. It is very useful to add comments to UML models with plain text notes to provide further explanation or clarification that might not be apparent. In Figure 2.19, a note is used to provide further details about the source of information of the classes.

Figure 2.19. Note annotation



Structural Models: Examples

Example 1: A Car

A car consists of different structural components such as the engine, body, suspension, gearbox, etc. Each component in turn contains its own attributes and operations. For example, the engine has its capacity, and it can be started or stopped. Figure 2.20 shows a simplified structural model of a car in a class diagram.

Example 2: A Sales Order System

In this simple sales order system example, there are three methods of payment: cash, credit card or check. These three payment methods have the same attribute (*amount*), but they have their own individual behaviors and attributes. Figure 2.21 shows a structural model of this simple sales order system in a class diagram. The directional associations in the diagram indicate the direction of navigation from one class to another. For example, the *Order* class can access information from the *Payment* class, but not the other way round.

Figure 2.20. Structural model of a car

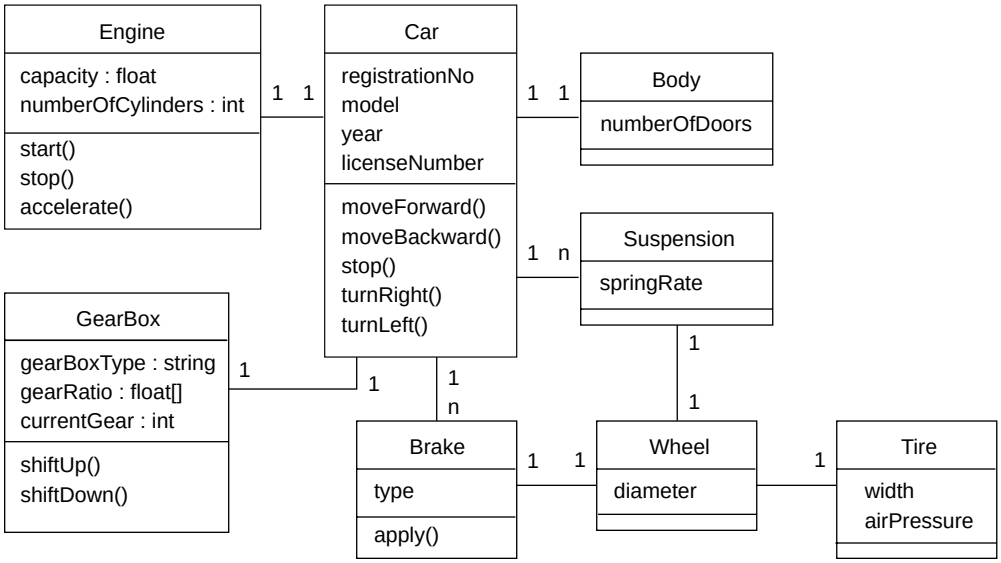
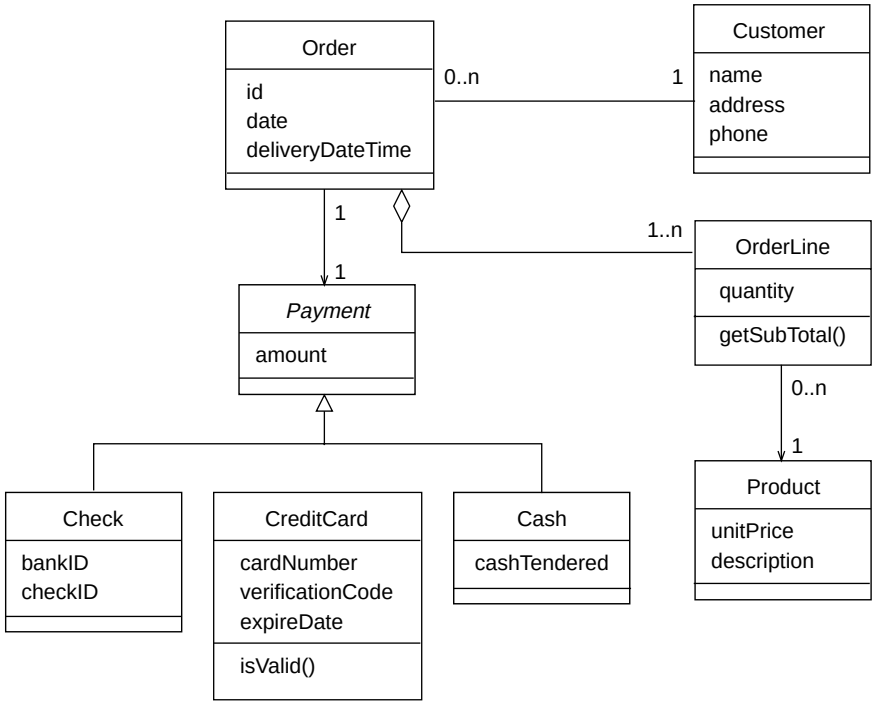


Figure 2.21. Structural model for a sales order system



Summary of UML Notation for Structural Modeling

UML provides a comprehensive range of components for structural modeling. Table 2.1 summarizes the more common ones in the UML notation. In this chapter, we have discussed how to use the class model in structural modeling from the analysis perspective. Thus, only some of the constructs in Table 2.1 are introduced.

Table 2.1. Summary of UML notation for structural modeling

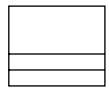
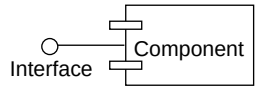
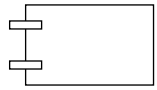
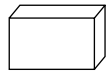
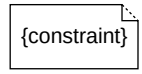
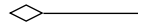
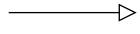
Construct	Description	Syntax
class	A set of objects that share the same attributes, operations, methods, relationships and semantics	
interface	A set of operations that characterize the behavior of an element	
component	A modular, replaceable and significant part of a system that packages implementation and exposes a set of interfaces	
node	A runtime physical object that represents a computational resource	
constraint	A semantic condition or restriction	
association	A relationship between two or more classifiers that involves connections between their instances	
aggregation	A special form of association that specifies a whole–parts relationship between the aggregate (whole) and the components (parts)	
generalization	A taxonomic relationship between a more general and a more specific element	

Table 2.1. (Cont'd)

Construct	Description	Syntax
dependency	A relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element)	---------->
realization	A relationship between a specification and its implementation	----------▷

Structural Analysis Techniques

In developing object-oriented systems, we often adopt a bottom-up approach first to develop a set of highly reusable components for assembling our system. These components should also be suitably placed in a flexible and expandable system architecture that can only be carried out through a top-down approach. To do this, a set of highly reusable components is developed first before they are assembled to form the system. In order to develop a stable system architecture that can comfortably accommodate the object components, the top-down and bottom-up approaches are often applied in an inter-play manner throughout the system development life cycle.

This section shall discuss various domain analysis techniques for object identification, after which leads you through the classical object identification process by performing a textual analysis. A set of long-established heuristics are elaborated, followed by a case study.

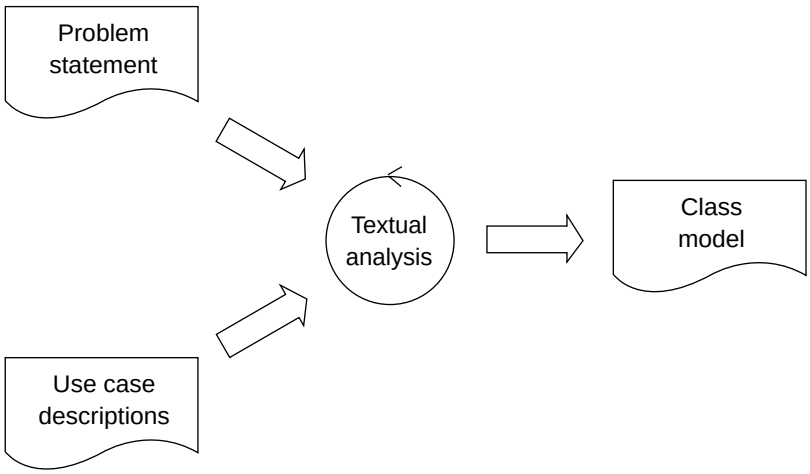
How Are Classes Obtained?

Practitioners and methodologists always claim that the object-oriented approach is far superior to the traditional structured approach. This may well be true. However, for those new to the object-oriented approach, they often find object identification a very difficult task, especially because a real-world object may be considered as either an attribute or an object depending on the context. For example, a city is a physical object in the real world. In the context of an *address*, *City* is only an attribute of the *Person* class. In an urban planning system, *City* would be a class itself.

How good a class model is can be judged by examining its usability, extendibility and maintainability. Furthermore, a good class model should be reusable in other object-oriented system components, so that the fruits of reusability can be harvested. Reusability is one of the key advantages of the object-oriented approach.

To tackle the object identification problem, both domain analysis and use case analysis (see Chapter 4 for details of the use case analysis) should be performed. Domain analysis starts with the problem statement to produce a class model (see Figure 2.22). Domain analysis focuses on identifying reusable objects that are common to most applications of the same problem domain. Hence, objects specific to the system can also be identified from use case descriptions. The results of both the domain analysis and use case analysis can be adopted to produce a robust and versatile class model. This will ensure that the class model can fulfill the users' requirements and be reused for other applications in the same domain.

Figure 2.22. Two ways to perform object identification



Keeping the Model Simple

Once you start modeling more than just a handful of classes, be sure that your abstractions provide a balanced set of responsibilities. What this means is that any one class should not be too big or too small. Each class should do one thing well. If the classes are too big, the models will be difficult to change and not very reusable. If the classes are too small, this will result in too many classes

in the model, which may be difficult to manage or understand. The "rule of seven" is often used, which postulates that people's short-term memory can only cope with about seven chunks of information at a time.

When there are more than seven classes, draw diagrams for different contexts. For example, in a retail information system, the classes can be packaged according to different areas of activities such as sales, inventory control, purchasing, etc., which in turn are represented in different class diagrams. It is often necessary to develop the same diagram iteratively and incrementally. In other words, the initial version of the diagram tends to be conceptual and should capture the "big picture" of the model. Later iterations capture additional details and are generally more implementation-oriented. Expect to revise the model many times before you are happy with it.

Heuristics in Using Structural Analysis

The following list of heuristics can help you perform structural analysis:

- Do not attempt to develop a single giant class diagram. Choose *only* those that fit into the context. For example, a class diagram may only represent one major system functionality (use case) instead of the entire system. Remember: humans can process about seven chunks of information at one time.
- Use model management constructs such as subsystems, packages and software framework to form the system architecture through the top-down approach.
- Consider both logical and physical aspects, such as grouping by role, responsibility, deployment and/or hardware platform, when grouping classes into model management constructs.
- Use data or middleware for communication among major subsystems whenever possible. Data coupling is easier to maintain than logical coupling because a change in requirements will only result in a change in data, and not the program itself. It is, however, not possible for some real-time or time critical applications since performance may become an important issue.
- Wisely apply design patterns for those architecturally significant classifiers to make the system architecture flexible and adaptable. This will be discussed in detail in Chapter 6.
- Apply domain analysis such as textual analysis, Class-Responsibility-Collaboration (CRC) or legacy and documentation reviews to identify reusable components using a bottom-up approach, so that the concepts and terminologies are understood and well accepted by the industry.

- Inter-play what have been found in the top-down approach and the bottom-up approach to ensure that the resulting artifacts (architecture, subsystem and components) can comfortably coexist.
- Use packages to organize the domain classes incrementally as development progresses. Each system functionality (use case) developed in turn will yield a set of domain classes. The set of domain classes should then be grouped into appropriate packages so that each package contains a cohesive set of classes. Organizing the classes into packages can also make it easier to manage the domain class model as it grows.
- Conduct use case analysis to yield two artifacts: a set of use case instance scenarios to help us walk through the objects that participate (are required) in the interaction, and the responsibilities (operations) that are required to be assigned to each object through the analysis of the messages sent to and from it. These resulting artifacts (a set of objects and its operations) help us identify the missing pieces in the structural model.
- Review whether a particular class is becoming too large. If so, consider reorganizing the class into two or more classes and structure the resulting classes using relationships.

Conducting Domain Modeling and Analysis

Domain analysis seeks to identify classes and objects that are common to many applications in a domain. This is partly to avoid wasting time and effort in reinventing the wheel and to promote reusability of the system components. Domain analysis involves finding out what other people have done in implementing other systems and looking at the literature in the field. Bear in mind that the object-oriented approach is superior to the traditional structured approach because of system reusability and extendibility, and not because they are more trendy or popular.

As already stated, the goal of domain analysis is to identify a set of classes that are common to all applications when dealing with problems of the same domain. Then, according to their nature, the domain classes and the application-specific classes are grouped into different packages. In so doing, the cohesion of the class model is maximized and the coupling between classes minimized, greatly enhancing the system's maintainability and extendibility. In short, the benefit of domain analysis is that domain classes can be reused for other applications when solving problems in the same domain. Furthermore, using well-understood terminologies in the domain for naming domain classes will improve the readability of the documentation.

Unfortunately, there is no simple or straightforward way to identify a set of classes for a problem domain. The domain analysis relies heavily on the designer's knowledge of the problem domain, intuition, prior experience and skills. A common way to perform a domain analysis is to prepare a statement of the problem domain first and then perform a textual analysis to identify the candidate classes. The problem statement and textual analysis provide a good starting point for domain analysis. The candidate classes are then refined iteratively to add the associations, attributes and operations to the domain class model (see the next section for details).

Domain Modeling and Analysis Process

Overview

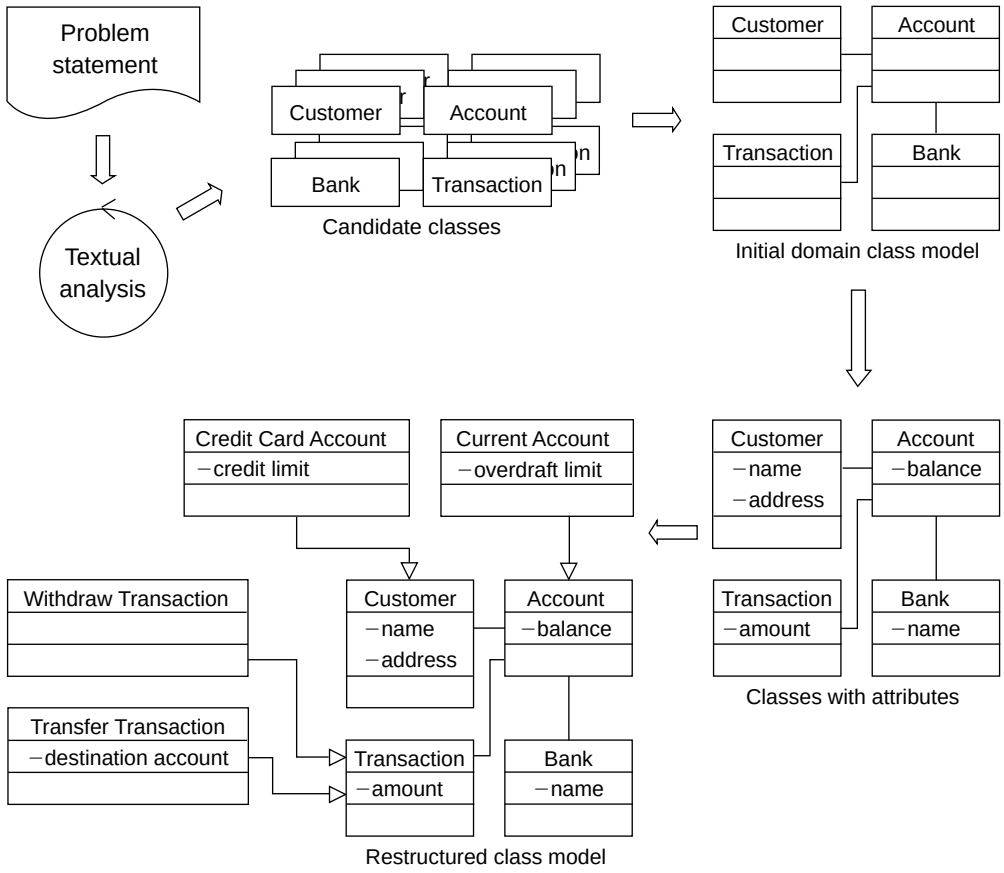
Before domain analysis is conducted, we need to understand the problem domain of the system. We need to find out the general requirements of the system of the domain by interviewing users and the domain experts of the system. After interviewing them, a problem statement can be prepared. The output of the domain analysis is a domain class model describing the classes and their relationships. The domain class model consists of class diagrams, a data dictionary describing the classes and their associations, and definitions of terminologies.

Developing Domain Class Models

The domain analysis starts with the preparation of a problem statement to provide a generic description of the problems of the domain. The problem statement is usually prepared after interviewing experts in the domain. Rumbaugh et al. recommend the following steps for developing a domain class model (see Figure 2.23):

1. Preparing the problem statement
2. Identifying the objects and classes using textual analysis
3. Developing a data dictionary
4. Identifying associations between classes
5. Identifying attributes of classes and association classes
6. Structuring classes using inheritance
7. Verifying access paths for likely queries
8. Iterating and refining the model

Figure 2.23. Domain analysis process



Preparing Problem Statement

Before any analysis work is carried out, it is important to clearly describe the problem in the context of the domain. A clear and detailed problem statement helps to reduce misunderstanding and the possibility of significant reworking at a later stage. Since the objective of domain analysis is to develop a class model that can be reused in other applications to solve problems in the same domain, it will be expedient that the problem statement describes the general requirements of the domain rather than the requirements of a specific application. The problem description should, therefore, focus on the description of the objects and their relationships in the domain rather than the specific procedures of the problem domain, since the procedures for carrying out tasks would not be the same for every organization. For example, in the problem

statement for the banking domain, it should be described that a customer can have several accounts with a bank but avoid specifying how a person opens a bank account since each bank has its own procedure in performing the same operation. As problem statements are written in natural language, they may have ambiguities and inconsistencies. Therefore, the problem statement is just one of the many inputs to the domain analysis. Throughout the analysis process, we need to use our own judgment or that of domain experts to resolve such ambiguities and inconsistencies.

Online Stock Trading Example

The following problem statement is for an automated online stock trading system for a stock brokerage firm.

A stock brokerage firm wants to provide an online stock trading service to enable its clients to make trades via the computer. With this system, a client must first be registered before he can trade online. The registration process involves the client providing his ID number, address and telephone number. A client may open one or more accounts for stock trading. The stock brokerage firm needs to be registered with a stock exchange before its clients can trade the stocks listed on the stock exchange. A stock brokerage firm can be registered with one or more stock exchanges. The stock brokerage firm may need to pay monthly charges for using the services provided by the stock exchange. Once registered, the client can buy and sell stocks. The client can check the current price, bid price, ask price and traded volume of a stock in real time. The stock price and traded volume information is provided by the stock exchange on which the stock is listed and traded. When a client issues a buy order for an account, the client must specify the stock code, the number of shares and the maximum price (bid price) that he is willing to pay for them. A client must have sufficient funds in his account to settle the transaction when it is completed. When a client issues a sell order, the client must specify the stock code, the number of shares and the minimum price (ask price) that he is willing to sell them. The client must have sufficient number of shares of the stock in his account before he can issue the sell order.

A client can check the status of execution of his (buy or sell) orders. The client can issue a buy or sell order before the end of the trading day

of the stock exchange which processed the order. All trade orders will be forwarded to the stock trading system of the stock exchange for execution. When an order is completed, the stock trading system of the stock exchange will return the transaction details of the order to the online stock trading system. The transaction details of a trade order may be a list of transactions, each transaction specifying the price and the number of shares traded. For example, the result of a buy order of 20,000 HSBC (stock code: 0005) shares at HKD 88.00 in the Hong Kong Stock Exchange may be as follows:

- 4,000 shares at HKD 87.50
- 8,000 shares at HKD 87.75
- 8,000 shares at HKD 88.00

An order will be kept on the system for execution until the order is completed or the end of a trading day. There are three possible outcomes for a trade order:

1. The trade order is completed. For a buy order, the total amount for the buy order will be deducted from the client's account and the number of shares of the stock purchased will be deposited into the account. For a sell order, the number of shares sold will be deducted from the client's account and proceeds of the sell order will be deposited into the client's account.
2. The trade order is partially completed. The number of shares actually traded (sell or buy) is less than the number of shares specified in the order. The number of shares successfully traded in the order will be used to calculate the amount of the proceeds, and the client's account is adjusted accordingly.
3. The trade order is not executed by the end of a trading day. The order will be canceled.

A stock exchange may require that the number of shares specified in an order must be in multiples of the lot size of the stock. Each stock has its own lot size. Common lot sizes are 1, 400, 500, 1,000 and 2,000 shares.

The client can deposit or withdraw cash or stock shares from his account. Upon the deposit or withdrawal of cash or stock shares, the account cash or stock balance will be updated accordingly.

Identifying Objects and Classes

To identify the objects and classes, perform textual analysis to extract all noun and noun phrases from the problem statement. The objective of this step is to identify a set of candidate objects which can be further elaborated and refined in subsequent steps. Therefore, it is not necessary (nor possible) to get it right the first time. Rather, do not be too selective in choosing classes at this stage so as to avoid the possibility of excluding some classes. For each extracted noun or noun phrase, we need to carefully evaluate whether it actually represents an object of the domain. It is necessary to stress that the object identification process is not a straightforward task. A noun or noun phrase can be an object in one domain and not so in another. We need to exercise our own judgment in the process. Amour and Miller (2001) suggest that from their past experiences, nouns or noun phrases of the following categories are more likely to represent objects:

- Tangibles (e.g. classroom, playground)
- Conceptuals (e.g. course, module)
- Events (e.g. test, examination, seminar)
- External organizations (e.g. publisher, supplier)
- Roles played (e.g. student, teacher, principal)
- Other systems (e.g. admission system, grade reporting system)

Table 2.2 shows the nouns and noun phrases extracted from the problem statement of the online stock trading example.

Table 2.2 Nouns and noun phrases extracted from the problem statement

Stock brokerage firm (concept)	Buy order (event)
Monthly charge	Stock code (simple value, attribute)
Trade (event)	Number of shares (simple value, attribute)
Trade order (event)	Maximum price (simple value, attribute)
Computer (tangible)	Transaction (event)
Client (role played)	Sell order (event)
ID (simple value, attribute)	Trading hours (simple value, attribute)
Address (simple value, attribute)	Trading day (simple value, attribute)

Table 2.2 (Cont'd)

Telephone number (simple value, attribute)	Stock trading system (other systems)
Account (concept)	Order (event)
Stock Exchange (extenal organization)	Execution result (event)
Stock (concept)	HSBC (instance of stock)
Current price (simple value, attribute)	Hong Kong Stock Exchange (instance of stock exchange)
Bid price (simple value, attribute)	Lot size (simple value, attribute)
Ask price (simple value, attribute)	Registration process (not an object)
Traded volume (simple value, attribute)	

As the purpose of this step is to identify the classes in the domain, other issues, such as inheritance and implementation, should be ignored. They will be dealt with in later steps. For each extracted noun or noun phrase, a category is assigned to it as shown in parentheses in Table 2.2. The candidate classes are then consolidated by eliminating inappropriate ones. Rumbaugh et al. (1991) suggest a set of criteria for eliminating inappropriate classes (see Table 2.3):

Table 2.3. Categories of inappropriate classes

Categories	Description
Redundant classes	Classes that mean the same thing. For example, order, trade and trade order mean the same thing. Eliminate trade and order, and retain trade order. Choose the most descriptive class.
Irrelevant classes	Classes that are not directly related to the problem. For example, monthly charge is not directly related to the system.
Vague classes	Classes that are loosely defined.
Attributes	Attributes of classes are also represented as nouns or noun phrases. Therefore, the list of nouns or noun phrases extracted by textual analysis may contain attributes of classes. For example, address and telephone number are attributes of the client.

Table 2.3. (Cont'd)

Categories	Description
Operations	The performance of actions is sometimes expressed as nouns or noun phrases. For example, the registration process is the action taken by the client to register on the system. It should be considered an operation of a class, rather than a class.
Roles	Role names help to differentiate the responsibilities of the objects in an interaction. However, they should not be considered as classes.
Implementation constructs	Implementation details of a particular solution are sometimes written in the problem statement, e.g. array, indexed sequential file, etc. Candidate classes representing the implementation details should be removed.

After following the above guidelines, a number of classes may be found to be inappropriate (see Table 2.4) in the online stock trading example.

Table 2.4. Inappropriate classes

Stock brokerage firm (irrelevant)	Stock code (attribute)
Monthly charge (irrelevant)	Number of shares (attribute)
Trade (redundant)	Maximum price (attribute)
Computer (implementation)	Trading hours (attribute)
ID (attribute)	Trading day (attribute)
Address (attribute)	Order (redundant)
Telephone number (attribute)	HSBC (instance of stock)
Current price (attribute)	Hong Kong Stock Exchange (instance of stock exchange)
Bid price (attribute)	Lot size (attribute)
Ask price (attribute)	Registration process (operation)
Traded volume (attribute)	

The revised list of candidate classes is shown in Table 2.5 after removing the inappropriate classes in Table 2.4.

Table 2.5. Revised list of candidate classes

Trade order (event)	Transaction (event)
Client (role played)	Sell order (event)
Account (concept)	Stock trading system (other systems)
Stock Exchange (external organization)	Execution result (event)
Buy order (event)	Stock (concept)

Developing Data Dictionary

After the candidate classes have been consolidated, prepare a data dictionary to record the definition of classes. For each class, write a short description to define its scope as well as details about the class such as its attributes and operations. The data dictionary also describes the associations between the classes and is continuously revised throughout the entire development life cycle of the system. Table 2.6 shows the data dictionary for the online trading system example.

Table 2.6. Data dictionary for the candidate classes

Class	Definition
Client	An individual or a company registered with the stock brokerage firm for online stock trading services. The class has attributes address, telephone number and ID. A client may have one or more accounts.
Account	A client can issue trade order on his or her accounts. An account holds details about the cash and stock balances for trading.
Stock exchange	A financial institution that provides a platform where stock trading is carried out.

Table 2.6. (Cont'd)

Class	Definition
Stock trading system	A platform for the execution of the trade orders of stock.
Trade order	A trade order specifies the price, stock code and number of shares. A trade order can be a buy order or a sell order.
Buy order	A buy order specifies the bid price, stock code and number of shares.
Sell order	A sell order specifies the ask price, stock code and number of shares.
Stock	A company listed in a stock exchange. Shares of a company can be traded only in a multiple of its lot size.
Execution result	The result of the execution of a trade order. It contains a list of transactions.
Transaction	The execution of a trade order at a particular price. It also contains the number of shares traded at that price.

Identifying Associations between Classes

An association is a relationship between objects. For example, John and Peter are instances of the class *person* and John is the father of Peter. Association can be identified by looking for verbs and verb phrases connecting two or more objects in the problem statement. In the online stock trading system example, the statement "a client may *open* one or more *accounts* for stock trading" [emphasis added] contains the verb "open" which links the client and the account. The association between the client and the *account* may be named as *has* since it is an ownership relationship. The association can also be named as *opened by* to reflect the action performed by the *client*. However, the word *has* can more accurately describe the nature of the association. Hence, the association should be named according to its nature rather than according to the verb or verb phrase linking the classes in the problem statement. Table 2.7 shows the list of verb phrases extracted from the problem statement to identify the candidate associations.

Table 2.7. Associations identified by extracting verb phrases from the problem statement

Verb phrase	Association
A client may open one or more accounts for stock trading.	has
When a client issues a buy order for an account, the client must specify the stock code, the number of shares and the maximum price that he is willing to pay for them (the bid price).	issued by, buy
When a client issues a sell order for an account, the client must specify the stock code, the number of shares and the minimum price that he is willing to sell them at (the ask price).	issued by, sell
All trade orders will be forwarded to the stock trading system of the stock exchange for execution.	executed by
When an order is completed, the stock trading system of the stock exchange will return the transaction details of the order to the online stock trading system.	returned by
The transaction details of a trade order may be a list of transactions, and each transaction specifies the price and the number of shares traded.	consists of

From the domain knowledge, we have the following associations:

- A stock is listed on a stock exchange
- A stock is traded on a stock trading system of a stock exchange
- The result of a trade order is a list of transactions
- A stock exchange has one or more stock trading systems

Based on the above information, formulate the initial domain class model for the system as illustrated in Figure 2.24.

Then refine the associations by eliminating unnecessary and inappropriate associations and by adding additional associations from the knowledge of the problem domain. Rumbaugh et al. propose the following criteria in Table 2.8 to determine whether an association should be eliminated.

Figure 2.24. Initial domain class model

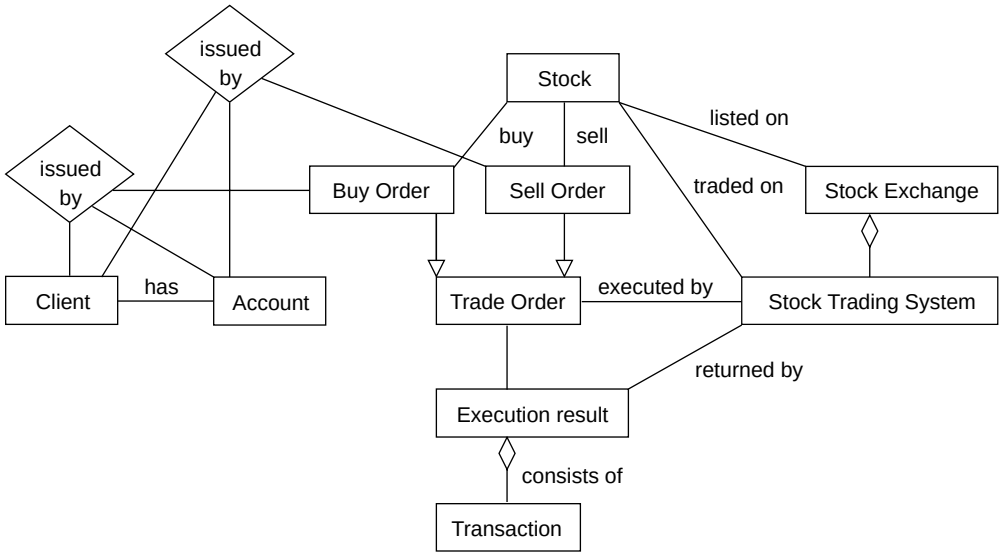


Table 2.8. Criteria to eliminate associations

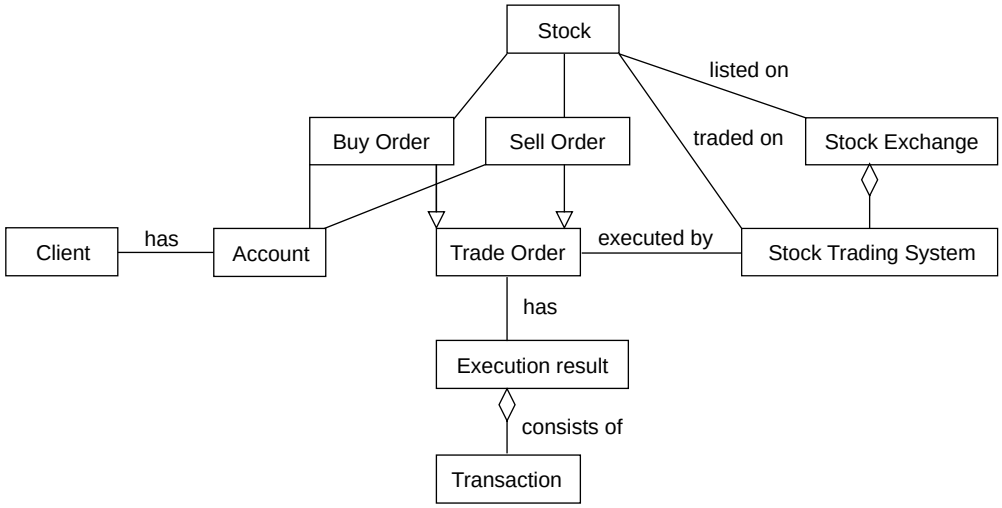
Criteria	Description
Associations between eliminated classes	If a class is eliminated from the domain class model, then all associations linking to it should be removed. In some cases, the dangling links of the classes caused by the removal of a class may be joined to form a new association.
Irrelevant or implementation associations	Associations that are not directly related to the problem domain or are only related to the solution of the problem should be eliminated.
Actions	The association should define structural relationships between domain classes, not an event. For example, “the client can check the status of execution of his (buy or sell) orders” describes an action performed by the client in an interaction between the client and the system.

Table 2.8. (Cont'd)

Criteria	Description
Ternary associations	Many associations involving three or more classes can be decomposed into binary associations. For example, "a client issues a buy order for an account" can be decomposed into two binary associations: "a client issues an order" and "the order is associated with the client's account".
Derived associations	Remove associations that can be defined in terms of other associations or a condition of the attributes of the classes. For example, "the stock trading system of the stock exchange will return the execution result" can be defined in terms of "a trade order is executed by a stock trading system" or "the trade order has an execution result".

Based on these guidelines, the revised domain class model can be refined as shown in Figure 2.25.

Figure 2.25. Revised domain class model



Identifying Attributes of Classes and Association Classes

Attributes are properties of a class, such as *name*, *address* and *telephone number* of the *Client* class. Look for nouns or noun phrases followed by possessive phrases, e.g. "address of the client." Adjectives that appear immediately before a noun and correspond to a class can also be an enumerated value of an attribute, e.g. "a canceled buy order." Attributes are less likely to be discovered from the problem statement. However, it is not necessary to identify all attributes in this step because the attributes do not affect the structure of the domain class model. Instead, this should only be done if they can be readily identified. At later stages of the development life cycle (e.g. detailed design phrase), the attributes can be more readily identified.

Structuring Classes Using Inheritance

At this point, most of the classes and associations have been identified, and it is possible to try to restructure the class diagram using inheritance. Inheritance provides an effective and convenient way to specify commonality between classes. Identify inheritance in two opposite directions: top down and bottom up.

Bottom-up Approach

For the bottom-up approach, we compare the properties of classes to look for commonality. Usually the names of the classes provide the first hint for the identification process. Look for classes with similar attributes, operations and associations with other classes. For example, the *Buy Order* and *Sell Order* classes both have the *price* and *number of shares* attributes and both of them are associated with the *Stock* class and *Account* class. Their names also suggest that they may share similar properties and behaviors.

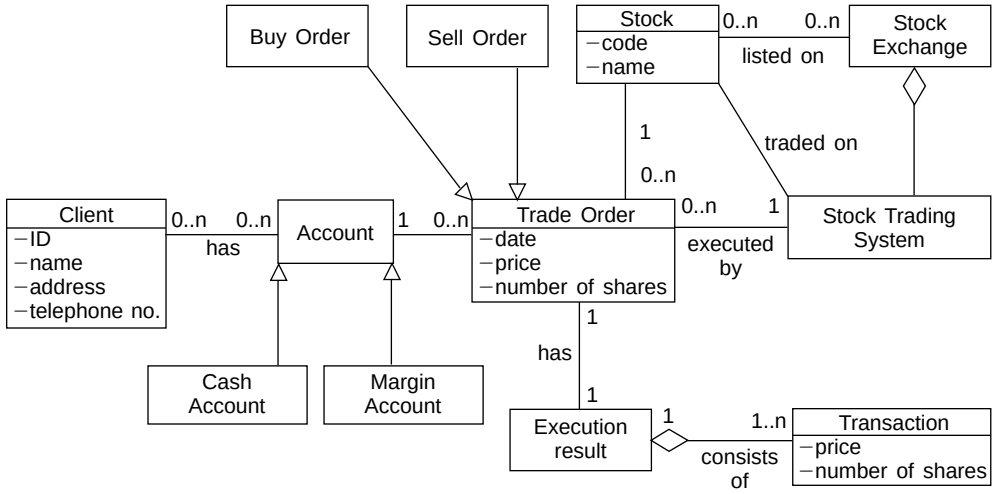
Also define a *superclass* to cover classes with a common structure. For example, the *Trade Order* class can cover the common structure of the *Buy Order* and *Sell Order* classes. Add an association between the *Trade Order* class and the *Account* class, and between the *Trade Order* class and the *Stock* class. The associations between the *Buy Order*, *Sell Order*, *Account* and *Stock* classes should be eliminated as these associations can be derived from inheritance and the associations of the superclass *Trade Order*.

Top-down Approach

For the top-down approach, check whether a class has some special cases that have additional structural or behavioral requirements. Look for noun phrases consisting of adjectives and class names. For example, the *Sell Order* and *Buy Order* classes are specializations of the *Trade Order* class. Taxonomies of

real-life objects can also suggest specializations of a class which may not be included in the problem statement. Think more broadly and use your domain knowledge in identifying specializations. For example, an *Account* can be categorized into two types: *Cash Account* and *Margin Account*. The revised domain class model is shown in Figure 2.26.

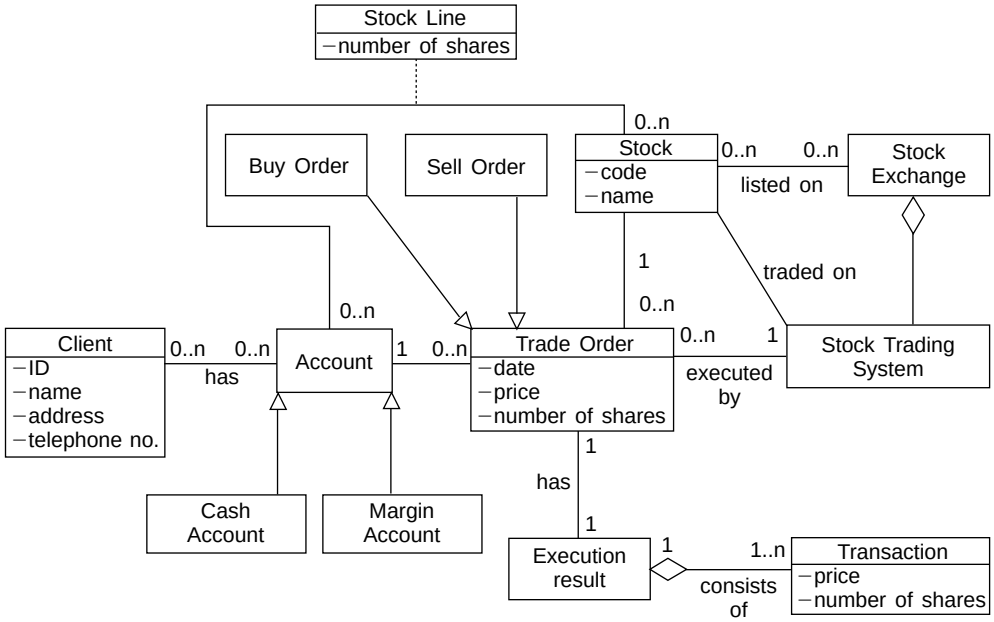
Figure 2.26. Revised domain class diagram after restructuring using inheritance and adding attributes



Verifying Access Paths for Likely Queries

One way to verify the correctness and usefulness of the domain class model is to check whether the domain class diagram can provide correct answers to queries that are common to other applications in the domain. In the online stock trading system example, a typical client query would be the current stock balance of his account. This requires an association between the *Account* class and the *Stock* class to provide the information on the number of shares held in the account. Although a path from the *Account* class to the *Stock* class exists in the domain class model in Figure 2.26, it would only provide the buy and sell orders information of the account but not the information on stock balances. To cope with this additional requirement, an association between the *Stock* class and the *Account* class as illustrated in Figure 2.27 needs to be added. The domain class model should always provide a correct answer to a typical query of the system.

Figure 2.27 Addition of an association between account class and stock class



Iterating and Refining Model

It is highly unlikely that the correct domain class model can be developed in one pass. The domain class model needs to be refined several times before it becomes robust. The development of the domain class model is not a rigid process, and it is necessary to repeatedly apply the above steps until the domain class model finally becomes stable. The following checklist can help in identifying areas of improvement of the domain class model.

- Where a class is without attributes, operations and associations, consider removing the class.
- Where a class is with many attributes and operations covering a wide area of requirements, consider splitting the class into two or more classes.
- Where a query cannot be answered by tracing the domain class model, consider adding additional associations.
- Where there are asymmetries in generalizations and association, consider adding additional associations and restructuring the classes with inheritance.
- Where attributes or operations are without a hosting class, consider adding new classes to hold these attributes and operations.

Tricks and Tips in Structural Modeling and Analysis

Set Focus and Context of Diagram

Make sure the class diagram only deals with the static aspects of the system. Do not attempt to consolidate everything into one single class diagram. Before you start to develop the diagram, set the context and the purpose it is to serve and the scope of the class diagram.

Use Appropriate Names for Classes

The classes can be identified from two sources: domain analysis and use case analysis. If the classes identified from the use case analysis are similar or identical to those derived from the domain analysis, that would be a perfect situation to be in. On the other hand, where inconsistent classes are derived from these two sources, discuss them with the end users, advising them to use standard terminologies of the industry, allowing for a dominant player in the field. If they insist on using their (non-standard) terminologies, it may be necessary to put the standard ones in the libraries and use subclasses for their non-standard terminologies specifically for this application.

Organize Diagram Elements

Not only should the classes be structured with various object-oriented semantics, but also organize their elements spatially to improve readability. For example, minimize cross lines in the diagram and place the semantically similar elements close together.

Annotate Diagram Elements

Attach notes to those elements where unclear concepts need to be clarified, and where necessary, attach external files, documents or links within the notes (i.e. a http link or a directory path). Some automated CASE tools support such annotations (e.g. Visual Paradigm for UML), so that resources can be glued into a navigable visual model.

Refine Structural Model Iteratively and Incrementally

As you progress through the development stages, the structural models can be enriched from time to time. For example, dynamic models help to identify the

responsibilities of the classes, or possibly even new classes, implementation classes and control classes. This concept will be discussed in more details in Chapter 4 (Dynamic Modeling and Analysis).

Show Only Relevant Associations

If a class is used by a number of use cases or even several applications, the class may have a number of associations that are related to different contexts. In the diagram, only show the associations related to the context that you are concerned with and hide the irrelevant associations. Do not attempt to consolidate all the associations and classes into a large class model as this cannot be easily managed by most people.

Domain Modeling and Analysis with VP-UML

In this section, the use of the key features of VP-UML to perform domain analysis will be demonstrated. The online stock trading system discussed earlier will be used in this chapter as an example. Simply follow the instructions on the following pages to create the sample domain class diagram. Follow the steps below to perform the domain model and analysis:

1. Prepare problem statement for the system being developed
2. Identify objects and classes
3. Develop data dictionary
4. Identify associations between classes
5. Identify attributes of classes and association classes
6. Structure object classes using inheritance
7. Verify access paths for likely queries
8. Iterate and refine the model

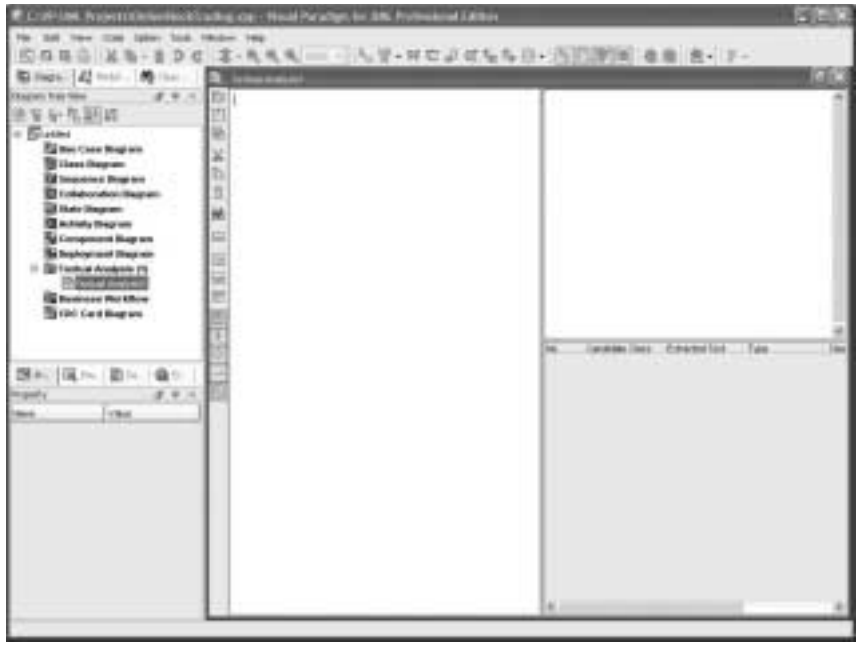
Step 1: Prepare Problem Statement

The problem statement is prepared through interviews with domain experts familiar with the application domain. Here, the application domain is an online stock trading system for stock brokerage firms. Alternatively, interview stakeholders of several stock brokerage firms to directly collect the requirements information. The problem statement should cover only the general requirements of an online stock trading system.

First, start up the VP-UML Integrated Development Environment and go through the following steps to enter the problem statement into VP-UML:

1.1. Click  on the **application toolbar** (see Figure 2.28).

Figure 2.28. Domain analysis work area



1.2. Type in the following problem statement in the **text pane**, or open it from a file (see Figure 2.29).

> For a stock brokerage firm that wants to provide an online stock trading service to enable its clients to make trades via the computer, a client must first be registered before he can trade online. The registration process involves the client providing his ID, address and telephone number. A client may open one or more accounts for stock trading.
>
> The stock brokerage firm needs to be registered with a stock exchange before its clients can trade the stocks listed on the stock exchange. A stock brokerage firm can be registered with one or

Figure 2.29. Entering problem statement

more stock exchanges. The stock brokerage firm may need to pay monthly charges for using the services provided by the stock exchange.

Once registered, the client can begin to buy and sell stocks. The client can check the current price, bid price, ask price and traded volume of a stock in real time. The stock price and traded volume information is provided by the stock exchange on which the stock is listed and traded. When a client issues a buy order for an account, the client must specify the stock code, the number of shares and the maximum price (bid price) that he is willing to pay for them. A client must have sufficient funds in his account to settle the transaction when it is completed. When a client issues a sell order, the client must specify the stock code, the number of shares and the minimum price (ask price) that he is willing to sell

them. The client must have sufficient number of shares of the stock in his account before he can issue the sell order. A client can check the status of execution of his (buy or sell) orders.

The client can issue a buy or sell order before the end of the trading hours of a trading day of the stock exchange which processed the order. All trade orders will be forwarded to the stock trading system of the stock exchange for execution. When an order is completed, the stock trading system of the stock exchange will return the transaction details of the order to the online stock trading system. The transaction details of a trade order may be a list of transactions, and each transaction specifies the price and the number of shares traded. For example, the result of a buy order of 20,000 HSBC (stock code: 0005) shares at HKD 88.00 in the Hong Kong Stock Exchange may be as follows:

- 4,000 shares at HKD 87.50
- 8,000 shares at HKD 87.75
- 8,000 shares at HKD 88.00

An order will be kept on the system for execution until the order is completed or the end of a trading day. There are three possible outcomes for a trade order:

1. The trade order is completed. For a buy order, the total amount for the buy order will be deducted from the client's account and the number of shares of the stock purchased will be deposited into the account. For a sell order, the number of shares sold will be deducted from the client's account and the proceeds of the sell order will be deposited into the account.
2. The trade order is partially completed. The number of shares actually traded (sell or buy) is less than the number of shares specified in the order. The number of shares successfully traded in the order will be used to calculate the amount of the proceeds, and the client's account is adjusted accordingly.
3. The trade order is not executed by the end of a trading day. The order is canceled.

A stock exchange may require that the number of shares specified in an order must be in multiples of the lot size of the

stock. Each stock has its own lot size. Commonly used lot sizes are 1, 400, 500, 1,000 and 2,000 shares.

The client can deposit or withdraw cash or stock shares from his account. Upon the deposit or withdrawal of cash or stock shares, the account cash or stock balance will be updated accordingly.

Step 2: Identify Objects and Classes

Once the problem statement is entered into the case tool, the next step is to identify objects and classes in the textual analysis working area.

2.1. Let us highlight the term *client* as a candidate class (see Figure 2.30) and drag it to the **Candidate Class Container** on the top right hand corner.

2.2. Notice that all occurrences of the same class in the problem statement is highlighted automatically (see Figure 2.31).

2.3. Repeat the above steps to identify the remaining classes:

- *Trade Order*
- *Account*
- *Stock Exchange*
- *Buy Order*
- *Transaction*
- *Sell Order*
- *Stock Trading System*
- *Execution Result*
- *Stock*

Step 3: Develop Data Dictionary

Let us define the candidate classes identified in Step 1. Select the **Class Description** cell next to the classes – *Client*. Enter the following description in the **Class Description** cell next to the class (see Figure 2.32). Adjust the size of the cell to view the whole description.

An individual or a company registered with the stock brokerage firm for the use of online stock trading services. The class has attributes address, telephone number and ID. A client can have one or more accounts.

Figure 2.30. Highlighting the word *client*

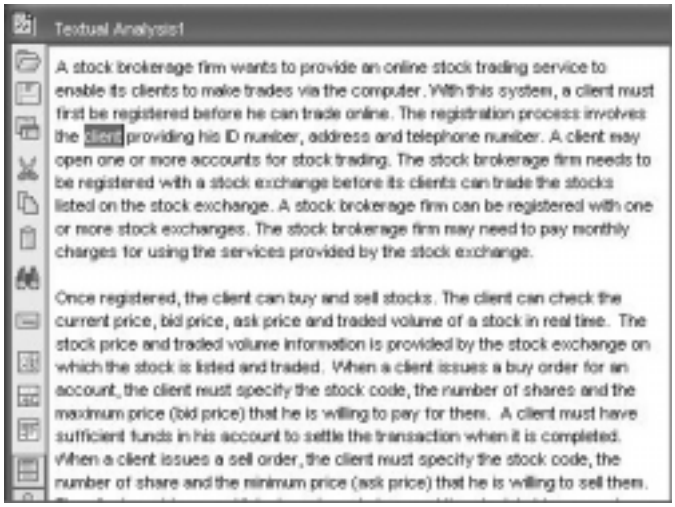


Figure 2.31. All occurrences of the word *client* are highlighted automatically

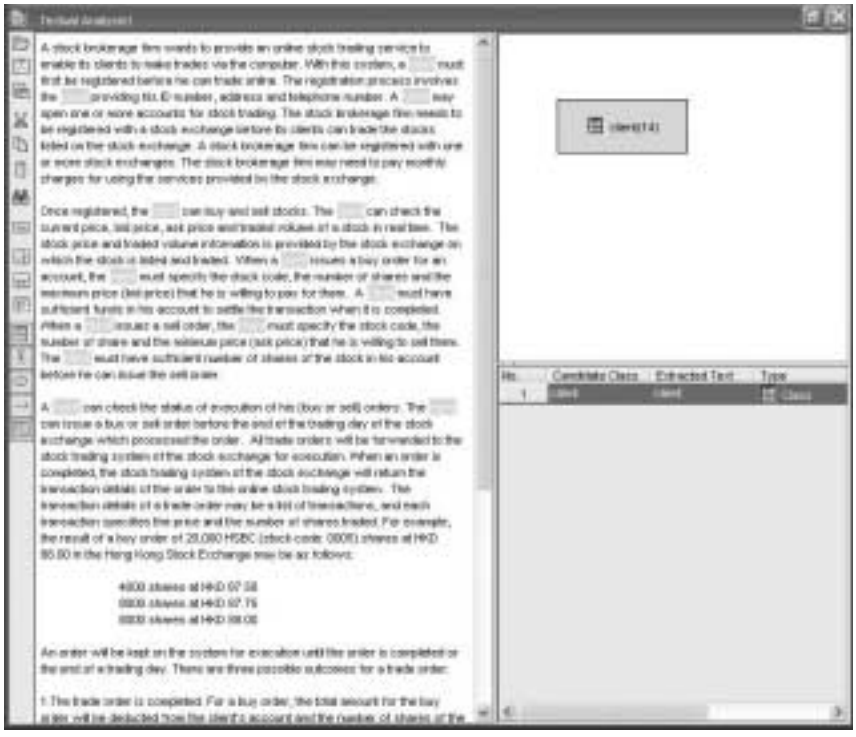
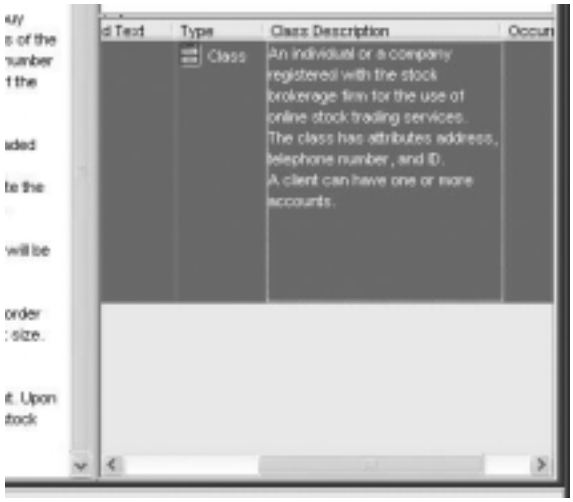


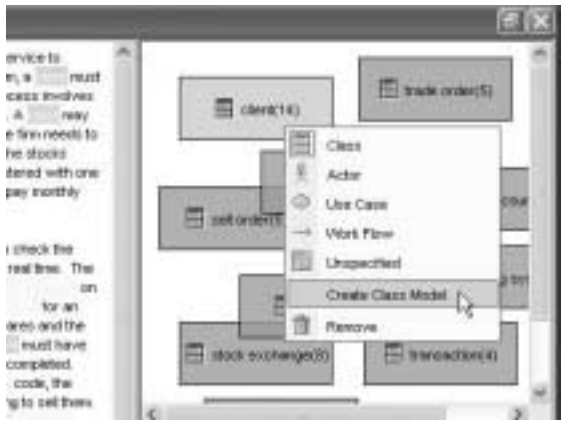
Figure 2.32. Data dictionary



Repeat the above steps to complete the dictionary for all remaining candidate classes.

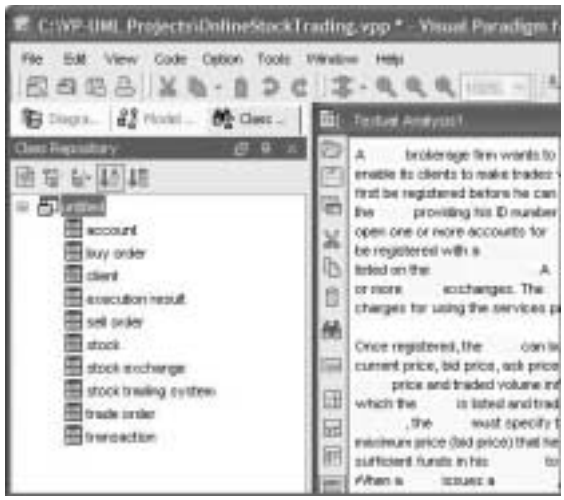
When the data dictionary has been defined, create the models from the candidate classes. To create a model, right click on the candidate class and select **Create Class Model** (see Figure 2.33). After that is done, the type of the candidate class will change to **Generated Model**, and the class model is created in the **Class Repository**.

Figure 2.33. Create model from Candidate Class



The candidate classes can be viewed by clicking the **Class Browser** tab at the bottom left corner of the screen (see Figure 2.34).

Figure 2.34. Class browser

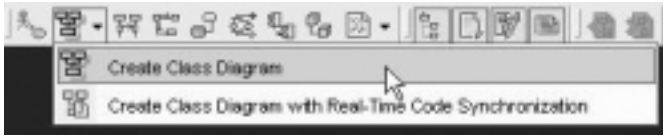


Step 4: Identify Associations between Classes

Having identified the candidate classes, the next step is to identify the associations among them. By analyzing the verb phrases in the problem statement, we find that the verb *open* connects two candidate classes in the statement "a client may open one or more accounts for stock trading." This is a "has a" relationship between *Client* and *Account*. So we can create an association between *Client* and *Account*.

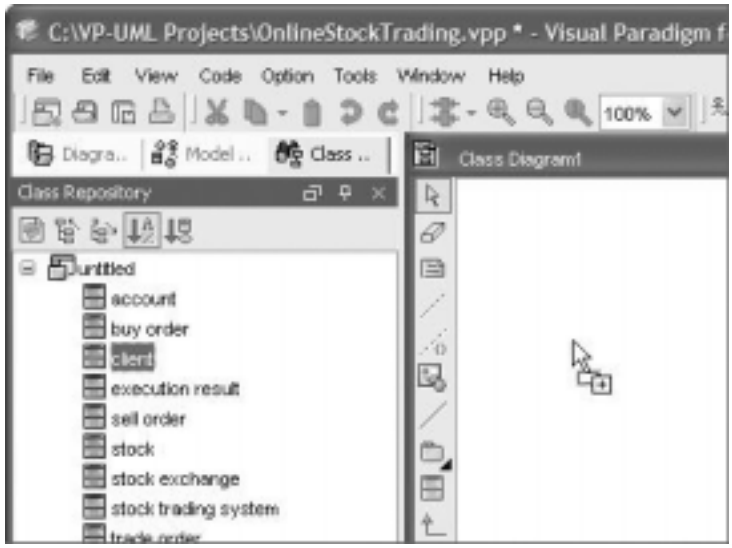
4.1. Create a class diagram by right clicking the **Class Diagram** button on the toolbar and select **Create Class Diagram** (see Figure 2.35). A new class diagram will appear in the **diagram pane**.

Figure 2.35. Create a Class Diagram



4.2. Drag the class *Client* from the **Class Browser** and drop it to the **Class Diagram** (see Figure 2.36).

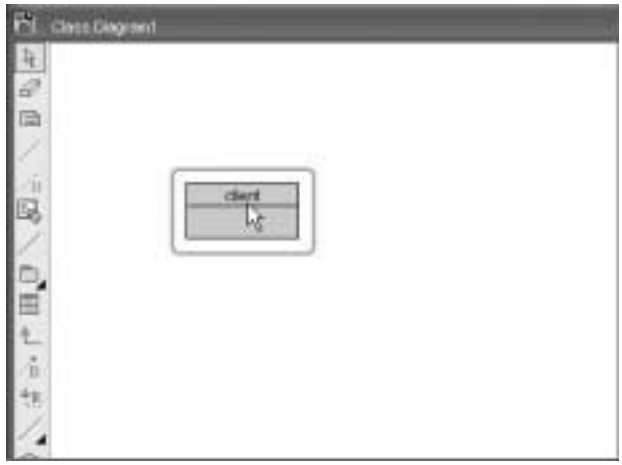
Figure 2.36. Creating class using Class Browser



4.3 The class *Client* now appears in the **Class Diagram** (see Figure 2.37).

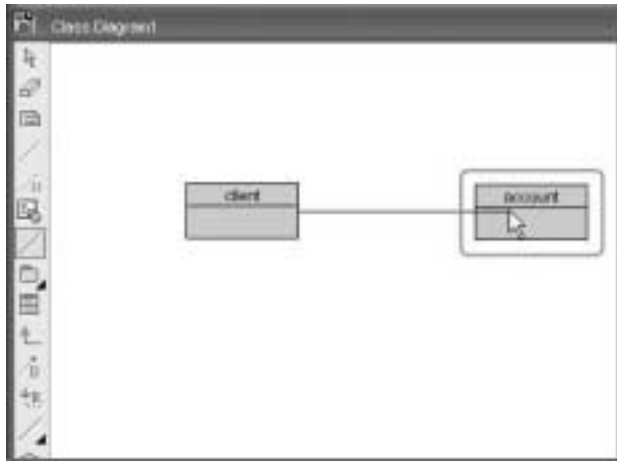
4.4 Repeat the previous steps to create the class *Account* in the **Class Diagram**.

Figure 2.37. Creating Class *Client*



4.5 To create an association between *Client* and *Account*, select the class *Client*, then click the association icon from the resource-centric interface and drag it to the class *Account*. The association between the *Client* and *Account* classes will then be created (see Figure 2.38).

Figure 2.38. Making an association between the classes *Client* and *Account*



4.6 Repeat the above steps to complete all other associations. Figure 2.39 shows the initial class model for the system.

Step 5: Identify Attributes of Classes and Association Classes

At this point, the basic structure of the domain class model is up and running. The domain class model should be refined by adding attributes to individual classes. As discussed earlier, attributes can be identified by textual analysis on nouns, noun phrases or adjectives. Look for nouns or noun phrases followed by a possessive phrase and a noun and corresponding to a class, e.g. *address of the client*. Adjectives appearing immediately before a noun and corresponding to a class can also be an enumerated value of a class's attribute, e.g. *a canceled sales order*. Follow the instructions below to add attributes to individual classes.

5.1. To create attributes in VP-UML, first select a class. Right click on the class *Client*, then select **New Attribute** (Figure 2.40).

5.2. Type in the attribute in the in-line **text editing area** and then press enter (see Figure 2.41).

5.3. Repeat the above steps to enter the attributes of the other classes. The domain class diagram with attributes is shown in Figure 2.42.

Figure 2.39. The initial domain class diagram

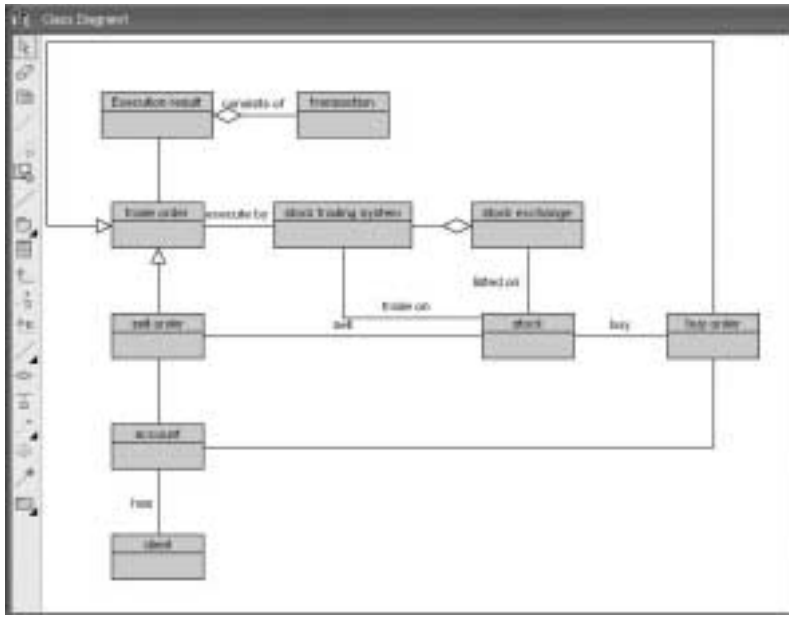


Figure 2.40. Adding an attribute

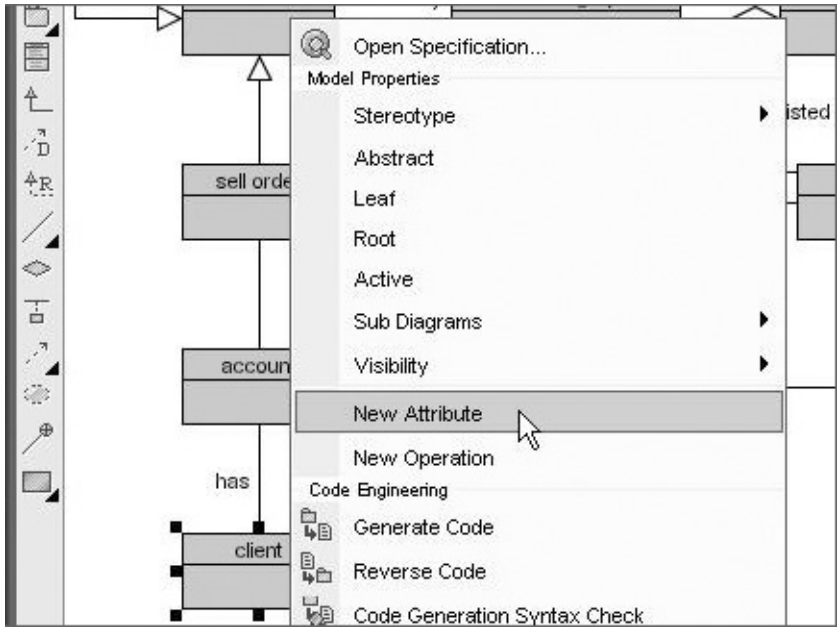


Figure 2.41. Editing an attribute name

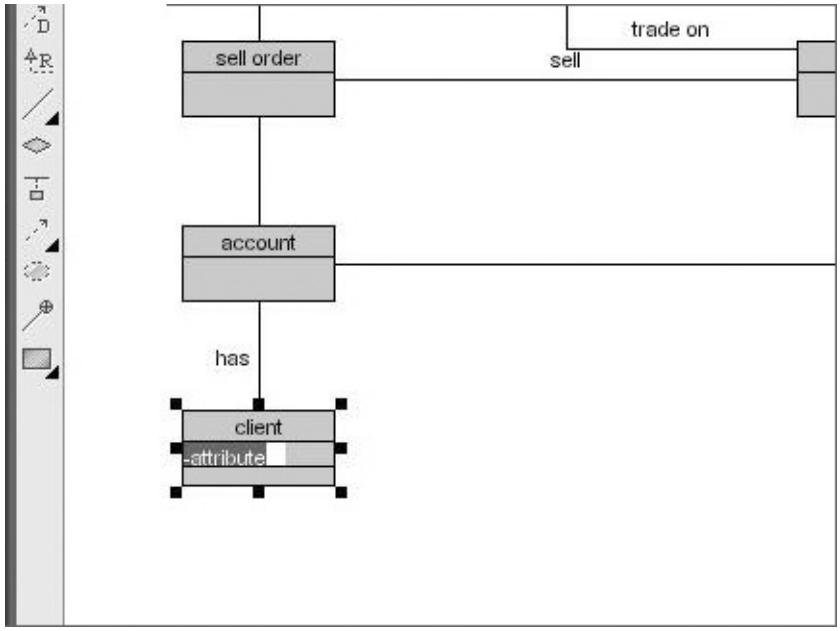
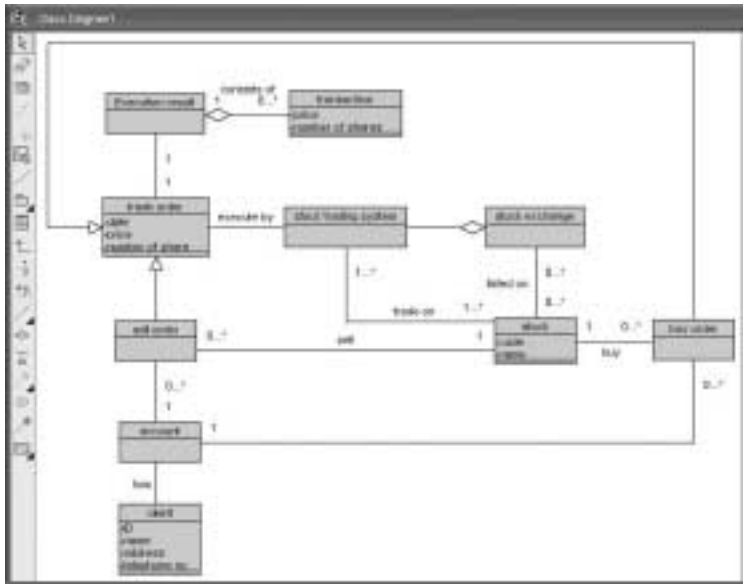


Figure 2.42. Initial domain class diagram with attributes



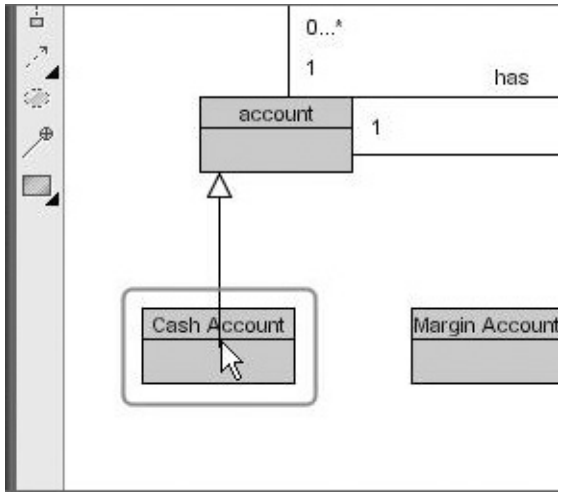
Step 6: Structure Object Classes Using Inheritance

As most classes have now been identified, start to reorganize the classes in order to further improve reusability and cohesion. We eliminate duplication of classes by singling out the common attributes and operations into superclasses. The cohesion within a class can be improved by breaking a "loosely coupled" class into two or more classes which may be related by inheritance or association.

6.1. By adopting the top-down approach, we discover that the class *Account* has two subtypes, *Cash Account* and *Margin Account*. To structure the *Cash Account* and *Account* classes using inheritance, first create the *Cash Account* and *Margin Account* subclasses.

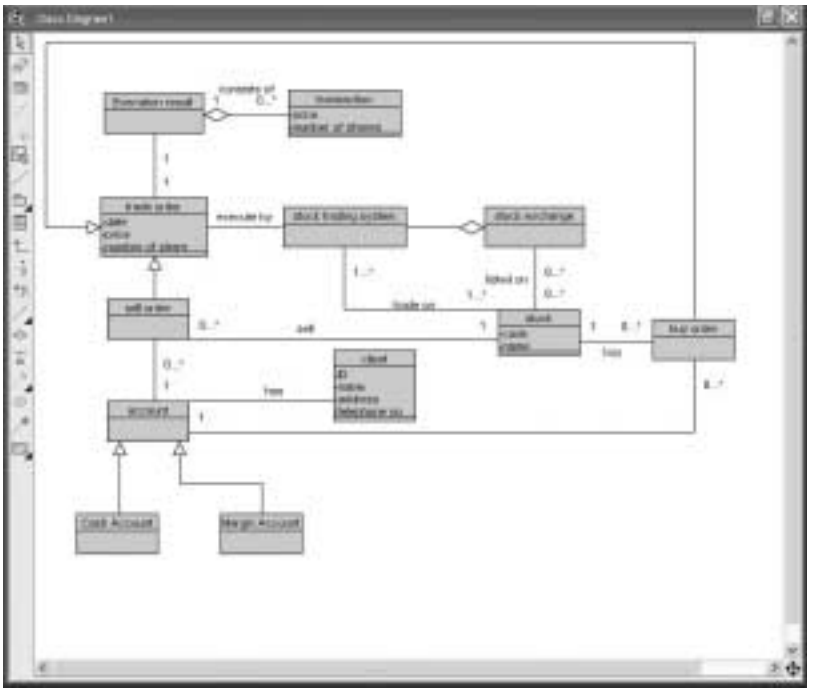
6.2. Select the *Account* class. Then click on the  icon from the resource-centric interface, and drag and place it on the *Cash Account* class. The inheritance relationship between *Account* and *Cash Account* is then specified (see Figure 2.43).

Figure 2.43. Creating inheritance relationship between *Margin Account* and *Cash Account*



6.3. Repeat the above steps to create the inheritance relationship between the *Margin Account* and *Account* classes. The restructured domain class diagram is shown in Figure 2.44.

Figure 2.44. Restructured domain class diagram



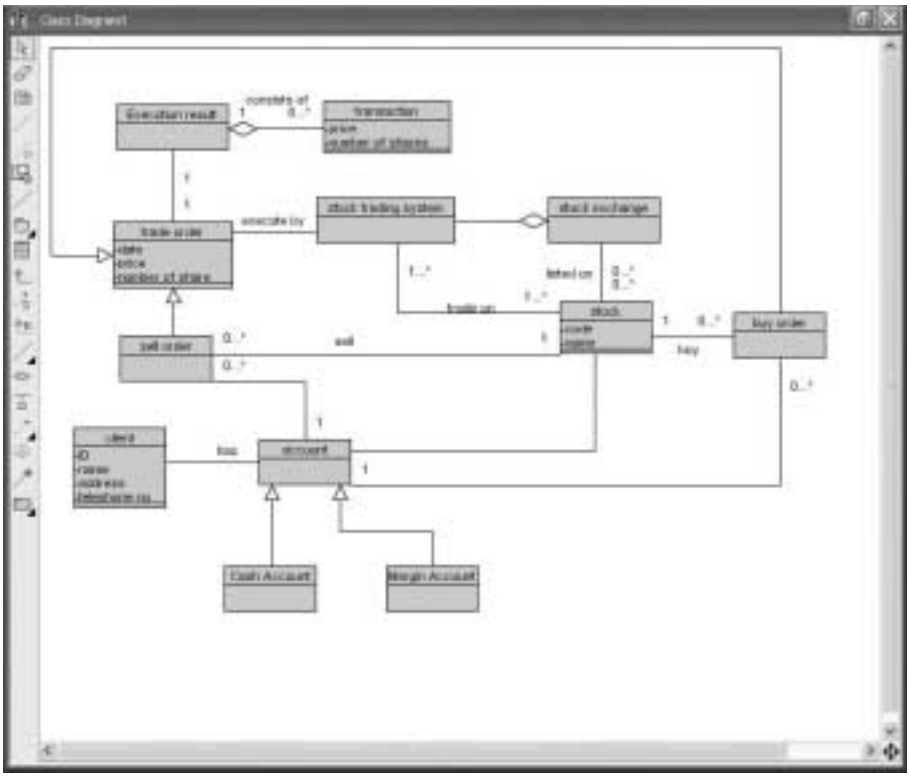
Step 7: Verify Access Paths for Likely Queries

Now verify the class diagram to see whether it can support typical queries of the application domain. Let us consider the following query: How does a client find out the stock balance of his account?

By examining the class diagram, the query cannot be answered directly as the class diagram can only show the transactions performed by the client. Of course the balance of a stock can be determined by all the transactions of the stock performed by the client. However, it is rather inconvenient and inefficient as a large number of transactions may be involved. Therefore, an association is added between *Account* and *Stock*. Follow the steps below to add the required association.

7.1. Follow the instructions given in Step 3 to create an association between *Account* and *Stock*, after which a domain class diagram like Figure 2.45 will be created.

Figure 2.45. Adding an association between *Account* and *Stock*



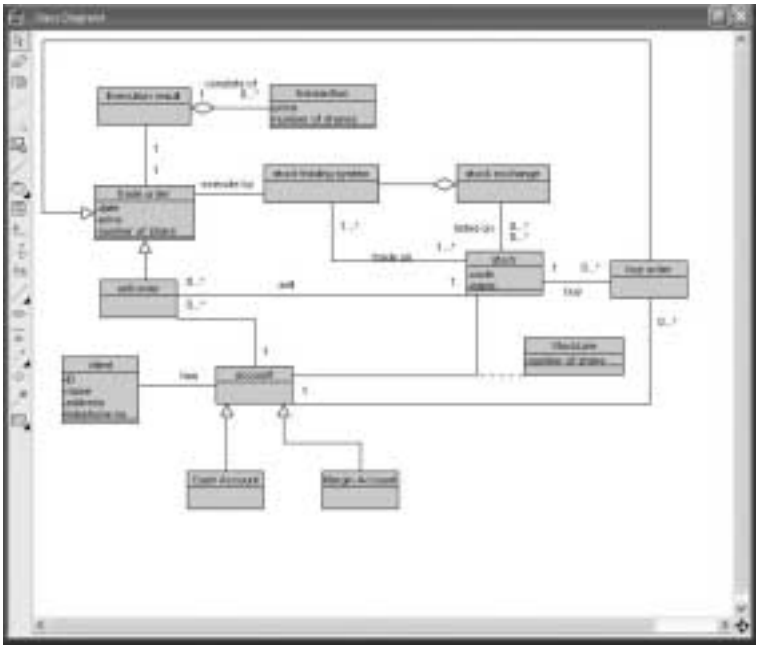
7.2. Now create an association class between *Account* and *Stock* to keep track of the balance of a stock in an account. Follow Step 6 to create the class *StockLine*. Click on the icon on the **diagram palette**, then click the *StockLine* class and drag it to the association.

7.3. Edit the name of the class in the **in-line editing area** of the class. A revised domain class diagram like Figure 2.46 will then be created.

Step 8: Iterate and Refine Model

Repeatedly apply Steps 2 to 7 to refine the domain class model until it becomes stable.

Figure 2.46. Revised domain class diagram after first iteration



Summary

A structural model provides a static view of a system, showing its key components and their relationships. In the UML notation, a structural model is represented by a class diagram.

In performing structural modeling and analysis, we start off with the problem statement to identify the domain objects and classes, which in turn can be used to compile a data dictionary for the system. By determining the associations between the classes and by identifying the attributes of the classes, the domain's class diagram can be created. The diagram can be structured more concisely for implementation by using inheritance. Finally, access paths for likely queries are verified and the model can be further refined by repeating this modeling and analysis procedure.

To illustrate the concepts described in this chapter, the modeling and analysis of an online stock trading system has been presented, detailing the steps involved by using the powerful features of the VP-UML CASE tool.

Exercise

Consider the following problem statement:

Problem Statement of an Online Book Store

The Pearl River Book Company is developing an online book store system through which its customers can buy books and sell their used books. Public users are those who are not registered customers of the system.

Public users or registered customers can search books by entering keywords, which may appear in the title, author or book description. The system displays a list of books that matches the keywords. Each entry of the book list consists of the book title, author(s), price for a new copy and price range for used copies. The user can select a book from the list to display more detailed information about it (availability, price for new copy, prices for used copies, table of contents, author, ISBN). The user can add a copy of the book (either new or old) to the shopping cart. The user can then continue to search for another book. When the user finishes searching, the user can checkout the books in the shopping cart. The system asks the user to login to his/her account by entering the user's email address and the account password. If the user has not registered yet, the user can register for a new customer account at that point. The user enters the email address, home address and password. The system verifies that the email address has not been used by an existing customer before confirming the creation of the new customer account through an email message. The system then asks the user to select the shipping option (express, priority or ordinary). Different shipping options have different prices. The user can then select the payment method (credit card or the user account of the book store). If the user selects payment by credit card, the user enters the card number, type and expiry date. The system then sends the credit card information and the amount charged to the external payment gateway. The amount is calculated by adding the prices of the selected books and the selected shipping option. If the credit card transaction is approved, the external payment gateway sends back an approval code. Otherwise, the systems will ask the user to reselect the payment method and re-enter the payment information. If the user selects payment by his/her account with a sufficient balance, the system charges the amount to the customer account. Otherwise, the system asks the user to re-reselect the payment

method. Upon completion of payment, the system arranges delivery of the ordered books. An external shipping agent is responsible for the delivery of the ordered books. If an order involves new books, the system sends a shipping request to notify the shipping agent to collect the books from the book store. New books in the same order are shipped together. If a used book has been ordered, the system sends a delivery request to notify the seller of the book and a shipping request to the shipping agent of the book store. The shipping agent collects the book from the seller and delivers the book to the buyer. Used books of the same order from the same seller are shipped together. After the book(s) has/have been delivered to the buyer, the shipping agent sends a shipping completion message to the system. Upon receipt of this message, the system updates the seller's customer account by adding the price of the used book minus the commission charge for the service.

A public user or a registered customer wanting to sell a used book can go through the above process by searching the book and displaying its information. The user can then post a used copy for sale. The system will ask the user to enter the price and the general condition of the used book. Then the system further asks the user to enter the email address and password of his/her customer account for login purposes. If the user does not have a customer account, the user can create a new customer account as described in the previous paragraph.

Incrementally and iteratively develop a domain class model for the online ticket reservation system by following the steps below:

- Identify objects and classes
- Develop a data dictionary
- Identify associations between classes
- Identify attributes of classes and association classes
- Structure object classes using inheritance
- Verify access paths for likely queries
- Iteratively refine the model