Unified Modelling Language 14

Abstract
This chapter presents the unified modelling language (UML), which is a visual modelling language for software systems, and it is used to present several views of the system architecture. It was developed at rational corporation as a notation for modelling object-oriented systems. We present various UML diagrams such as use case diagrams, sequence diagrams and activity diagrams.

Keywords
Use case diagrams · Classes and objects · Sequence diagrams · Activity diagrams · State diagrams · Collaboration diagrams · Object constraint language · Rational unified process

14.1 Introduction

The unified modelling language (UML) is a visual modelling language for software systems. It was developed by Jim Rumbaugh, Grady Booch and Ivar Jacobson [1] at rational corporation (now part of IBM), as a notation for modelling object-oriented systems. It provides a visual means of specifying, constructing and documenting object-oriented systems, and it facilitates the understanding of the architecture of the system, and in managing the complexity of a large system.

The language was strongly influenced by three existing methods: the object modelling technique (OMT) developed by Rumbaught, the *Booch Method* developed by Booch and object-oriented software engineering (OOSE) developed by Jacobson. UML unifies and improves upon these methods, and it has become a popular formal approach to modelling software systems.

G. O'Regan, *Concise Guide to Software Engineering*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-57750-0_14

Models provide a better understanding of the system to be developed, and a UML model allows the system to be visualized prior to its implementation, and it simplifies the underlying reality. Large complex systems are difficult to understand in their entirety, and the use of a UML model is an aid to abstracting and simplifying complexity. The choice of the model is fundamental, and a good model will provide a good insight into the system. Models need to be explored and tested to ensure their adequacy as a representation of the system. Models simplify the reality, but it is important to ensure that the simplification does not exclude any important details. The chosen model affects the view of the system, and different roles require different viewpoints of the proposed system.

An architect will design a house prior to its construction, and the blueprints will contain details of the plan of each room, as well as plans for electricity and plumbing. That is, the plans for a house include floor plans, electrical plans and plumping plans. These plans provide different viewpoints of the house to be constructed and are used to provide estimates of the time and materials required to construct it.

A database developer will often focus on entity-relationship models, whereas a systems analyst may focus on algorithmic models. An object-oriented developer will focus on classes and on the interactions of classes. Often, there is a need to view the system at different levels of detail, and no single model in itself is sufficient for this. This leads to the development of a small number of interrelated models.

UML provides a formal model to the system, and it allows the same information to be presented in several ways, and at different levels of detail. The requirements of the system are expressed in terms of use cases; the design view captures the problem space and solution space; the process view models the systems processes; the implementation view addresses the implementation of the system, and the deployment view models the physical deployment of the system.

There are several UML diagrams providing different viewpoints of the system, and these provide the blueprint of the software.

14.2 Overview of UML

UML is an expressive graphical modelling language for visualizing, specifying, constructing and documenting a software system. It provides several views of the software's architecture, and it has a clearly defined syntax and semantics. Each stakeholder (e.g. project manager, developers and testers) has a different perspective and looks at the system in different ways at different times during the project. UML is a way to model the software system before implementing it in a programming language.

A UML specification consists of precise, complete and unambiguous models. The models may be employed to generate code in a programming language such as Java or C++. The reverse is also possible, and so it is possible to work with either the graphical notation of UML or the textual notation of a programming language. UML expresses things that are best expressed graphically, whereas a programming language expresses things that are best expressed textually, and tools are employed to keep both views consistent. UML may be employed to document the software system, and it has been employed in several domains including the banking sector, defence and telecommunications.

The use of UML requires an understanding of its basic building blocks, the rules for combining the building blocks and the common mechanisms that apply throughout the language. There are three kinds of building blocks employed:

- Things
- Relationships
- Diagrams

Things are the object-oriented building blocks of the UML. They include *structural things, behavioural things, grouping things* and *annotational things* (Table 14.1). Structural things are the nouns of the UML models, behavioural things are the dynamic parts and represent behaviour and their interactions over time, grouping things are the organization parts of UML, and annotation things are the explanatory parts. Things, relationships and diagrams are all described graphically and are discussed in detail in [1].

Table 14.1 Classification of UML things

Thing	Kind	Description
Structural	Class	A class is a description of a set of objects that share the same attributes and operations
	Interface	An interface is a collection of operations that specify a service of a class or component. It specifies externally visible behaviour of the element
	Collaboration	A collaboration defines an interaction between software objects
	Use case	A use case is a set of actions that define the interaction between an actor and the system to achieve a particular goal
	Active class	An active class is used to describe concurrent behaviour of a system
	Component	A component is used to represent any part of a system for which UML diagrams are made
	Node	A node is used to represent a physical part of the system (e.g. server and network)
Behavioural	Interaction	These comprise interactions (message exchange between components) expressed as sequence diagrams or collaboration diagrams
	State machine	A state machine is used to describe different states of system components
Grouping	Packages	These are the organization parts of UML models. A package organizes elements into groups and is a way to organize a UML model
Annotation		These are the explanatory parts (notes) of UML

There are four kinds of relationship in UML:

- Dependency
- Association
- Generalization
- Extensibility

Dependency is used to represent a relationship between two elements of a system, in which a change to one thing affects the other thing (dependent thing). *Association* describes how elements in the UML diagram are associated and describes a set of connections among elements in a system. *Aggregation* is an association that represents a structural relationship between a whole and its parts. A *generalization* is a parent–child relationship in which the objects of the specialized element (child) are substituted for objects of the generalized element (the parent). *Extensibility* refers to a mechanism to extend the power of the language to represent extra behaviour of the system. Next, we describe the key UML diagrams.

14.3 UML Diagrams

The UML diagrams provide a graphical visualization of the system from different viewpoints, and we present several key UML diagrams in Table 14.2.

Table 14.2 UML diagrams

Diagram	Description
Class	A class is a key building block of any objected-oriented system. The class diagram shows the classes, their attributes and operations, and the relationships between them
Object	This shows a set of objects and their relationships. An object diagram is an instance of a class diagram
Use case	These show the actors in the system and the different functions that they require from the system
Sequence	These diagrams show how objects interact with each other, and the order in which the interactions occur
Collaboration	This is an interaction diagram that emphasizes the structural organization of objects that send and receive messages
State chart	These describe the behaviour of objects that act differently according to the state that they are in
Activity	This diagram is used to illustrate the flow of control in a system (it is similar to a flow chart)
Component	This diagram shows the structural relationship of components of a software system and their relationships/interfaces
Deployment	This diagram is used for visualizing the deployment view of a system and shows the hardware of the system and the software on the hardware

The concept of class and objects are taken from object-oriented design, and classes are the most important building block of any object-oriented system. A class is a set of objects that share the same attributes, operations, relationships and semantics [1]. Classes may represent software things and hardware things. For example, walls, doors and windows are all classes, whereas individual doors and windows are objects. A class represents a set of objects rather than an individual object.

Automated bank teller machines (ATMs) include two key classes: Customers and Accounts. The class definition includes both the data structure for Customers and Accounts, and the operations on Customers and Accounts. These include operations to add or remove a Customer, operations to debit or credit an Account, or to transfer from one Account to another. There are several instances of Customers and Accounts, and these are the actual Customers of the bank and their Accounts.

Every class has a name (e.g. Customer and Account) to distinguish it from other classes. There will generally be several objects associated with the class. The class diagram describes the name of the class, its attributes and its operations. An attribute represents some property of the class that is shared by all objects; for example, the attributes of the class "Customer" are name and address. Attributes are listed below the class name, and the operations are listed below the attributes. The operations may be applied to any object in the class. The responsibilities of a class may also be included in the definition (Table 14.3).

Class diagrams typically include various relationships between classes. In practice, very few classes are stand alone, and most collaborate with others in various ways. The relationship between classes needs to be considered, and these provide different ways of combining classes to form new classes. The relationships include dependencies (a change to one thing affects the dependent thing), generalizations (these link generalized classes to their specializations in a subclass/superclass relationship) and associations (these represent structural relationships among objects).

A dependency is a relationship that states that a change in the specification of one thing affects the dependent thing. It is indicated by a dashed line (—— >). Generalizations allow a child class to be created from one or more parent classes (single inheritance or multiple inheritance). A class that has no parents is termed a base class (e.g. consider the base class Shape with three children: Rectangle, Circle and Polygon, and where Rectangle has one child namely Square). Generalization is indicated by a solid directed line that points to the parent (—— ►). Association is a

Table 14.3 Simple class diagram

Customer	Account
Name: String Address: String	Balance:Real Type:String

Add()
Remove()
	Debit()
	Credit()
	CheckBal()
	Transfer()

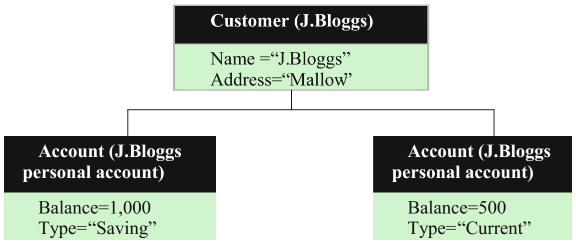


Fig. 14.1 Simple object diagram

structural relationship that specifies that objects of one thing are connected to objects of another thing. It is indicated by a solid line connecting the same or different classes.

The object diagram (Fig. 14.1) shows a set of objects and their relationships at a point of time. It is related to the class diagram in that the object is an instance of the class. The ATM example above had two classes (Customers and Accounts), and the objects of these classes are the actual Customers and their corresponding Accounts. Each Customer may have several Accounts, and the names and addresses of the Customers are detailed as well as the corresponding balance in the Customer's Accounts. There is one instance of the Customer class and two instances of the Account class in this example.

An object has a state that has a given value at each time instance. Operations on the object will typically (with the exception of query operations) change its state. An object diagram contains objects and links to other objects and gives a snapshot of the system at a particular moment of time.

A use case diagram models the dynamic aspects of the system, and it shows a set of use cases and actors and their relationships. It describes scenarios (or sequences of actions) in the system from the user's viewpoint (actor) and shows how the actor interacts with the system. An actor represents the set of roles that a user can play, and the actor may be human or an automated system. Actors are connected to use cases by association, and they may communicate by sending and receiving messages.

A use case diagram shows a set of use cases, with each use case representing a functional requirement. Use cases are employed to model the visible services that the system provides within the context of its environment, and for specifying the requirements of the system as a black box. Each use case carries out some work that is of value to the actor, and the behaviour of the use case is described by the flow of events in text. The description includes the main flow of events for the use case and the exceptional flow of events. These flows may also be represented graphically. There may also be alternate flows and the main flow of the use case. Each sequence is termed a scenario, and a scenario is one instance of a use case.

Use cases provide a way for the end-users and developers to share a common understanding of the system. They may be applied to all or part of the system (subsystem), and the use cases are the basis for development and testing. A use case is represented graphically by an ellipse. The benefits of use cases include:

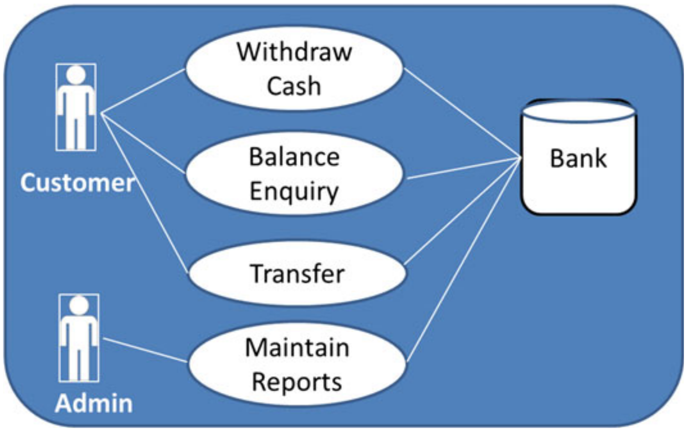


Fig. 14.2 Use case diagram of ATM machine

- Enables the stakeholders (e.g. domain experts, developers, testers and end-users) to share a common understanding of the functional requirements.
- Models the requirements (specifies what the system should do).
- Models the context of a system (identifies actors and their roles)
- May be used for development and testing.

Figure 14.2 presents a simple example of the definition of the use cases for an ATM application. The typical user operations at an ATM machine include the balance enquiry operation, cash withdrawal and the transfer of funds from one Account to another. The actors for the system include "Customer" and "admin," and these actors have different needs and expectations of the system.

The behaviour from the user's viewpoint is described, and the use cases include "withdraw cash," "balance enquiry," "transfer" and "maintain/reports." The use case view includes the actors who are performing the sequence of actions.

The next UML diagram considered is the sequence diagram which models the dynamic aspects of the system and shows the interaction between objects/classes in the system for each use case. The interactions model the flow of control that characterizes the behaviour of the system, and the objects that play a role in the interaction are identified. A sequence diagram emphasizes the time ordering of messages, and the interactions may include messages that are dispatched from object to object, with the messages ordered in sequence by time.

The example in Fig. 14.3 considers the sequences of interactions between objects for the "Balance Enquiry" use case. This sequence diagram is specific to the case of a valid balance enquiry, and a sequence diagram is also needed to handle the exception cases.

The behaviour of the "balance enquiry" operation is evident from the diagram. The Customer inserts the card into the ATM machine, and the PIN number is requested by the ATM. The Customer then enters the number, and the ATM

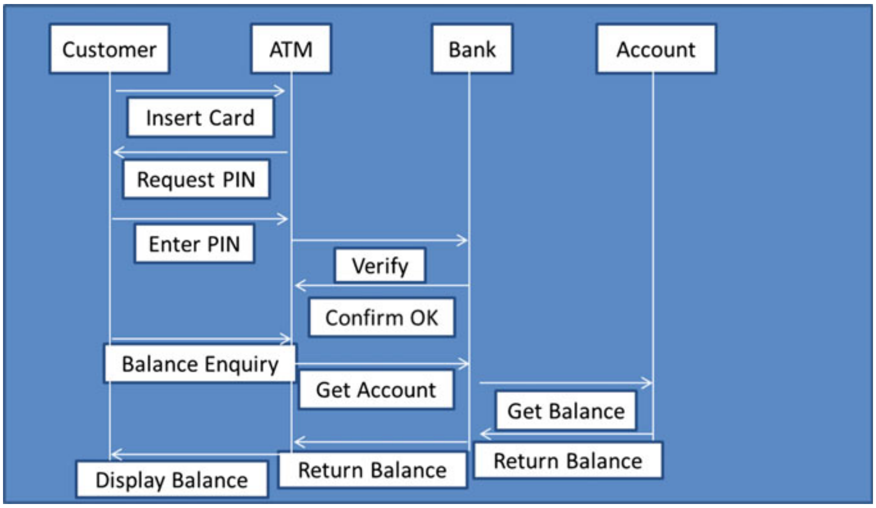


Fig. 14.3 UML sequence diagram for balance enquiry

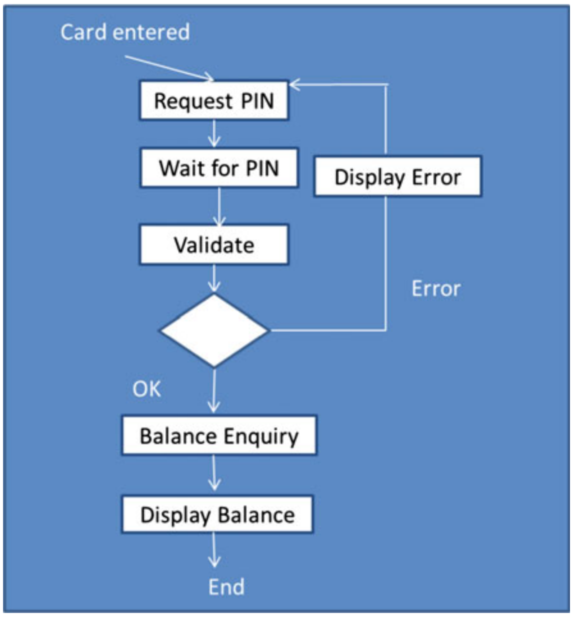
machine contacts the bank for verification of the number. The bank confirms the validity of the number, and the Customer then selects the balance enquiry operation. The ATM contacts the bank to request the balance of the particular Account, and the bank sends the details to the ATM machine. The balance is displayed on the screen of the ATM machine. The Customer then withdraws the card. The actual sequence of interactions is evident from the sequence diagram.

The example above has four objects (Customer, ATM, Bank and Account), and these are laid out from left to right at the top of the sequence diagram. Collaboration diagrams are interaction diagrams that consist of objects and their relationships. However, while sequence diagrams emphasize the time ordering of messages, a collaboration diagram emphasizes the structural organization of the objects that send and receive messages. Sequence diagrams and collaboration diagrams may be converted to the other without loss of information. Collaboration diagrams are described in more detail in [1].

The activity diagram is considered in Fig. 14.4, and this diagram is essentially a flow chart showing the flow of control from one activity to another. It is used to model the dynamic aspects of a system, and this involves modelling the sequential and possibly concurrent steps in a computational process. It is different from a sequence diagram in that it shows the flow from activity to activity, whereas a sequence diagram shows the flow from object to object.

State diagrams (also known as state machine diagrams or state charts) show the dynamic behaviour of a class and how an object behaves differently depending on the state that it is in. There is an initial state and a final state, and the operation generally results in a change of state, with the operations resulting in different states being entered and exited. A state diagram is an enhanced version of a finite state machine (as discussed in Chap. 12) Fig. 14.5.

Fig. 14.4 UML activity diagram



There are several other UML diagrams including component and deployment diagrams. The reader is referred to [1].

Advantages of UML UML offers a rich notation to model software systems and to understand the proposed system from different viewpoints. Its main advantages are shown in Table 14.4.

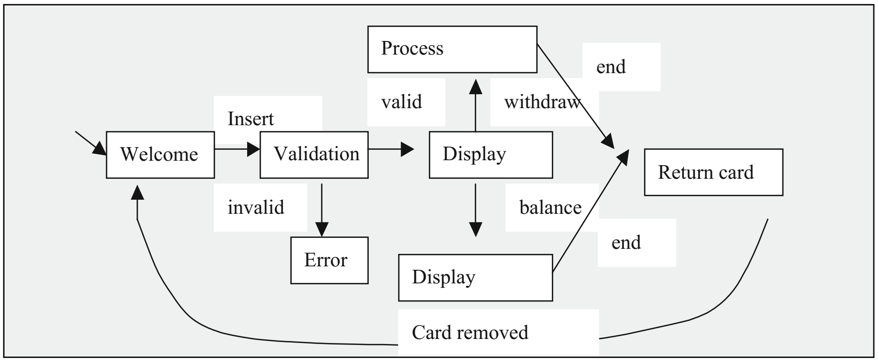


Fig. 14.5 UML state diagram

Table 14.4 Advantages of UML

Advantages of UML
Visual modelling language with a rich expressive notation
Mechanism to manage complexity of a large system Enables the proposed system to be studied before implementation
Visualization of architecture design of the system
It provides different views of the system
Visualization of system from different viewpoints
Use cases allow the description of typical user behaviour Better understanding of implications of user behaviour
Use cases provide a mechanism to communicate the proposed behaviour of the software system Use cases are the basis of development and testing

14.4 Object Constraint Language

The object constraint language (OCL) is a declarative language that provides a precise way of describing rules (or expressing constraints) on the UML models. OCL was originally developed as a business modelling language by Jos Warmer at IBM, and it was developed further by the Object Management Group (OMG), as part of a formal specification language extension to UML. It was initially used as part of UML, but it is now used independently of UML.

OCL is a pure expression language, i.e., there are no side effects as in imperative programming languages, and the OCL expressions can be used in various places in the UML model including:

- Specify the initial value of an attribute.
- Specify the body of an operation.
- Specify a condition.

There are several types of OCL constraints including are shown in Table 14.5.

There are various tools available to support OCL, and these include OCL compilers (or checkers) that provide syntax and consistency checking of the OCL constraints, and the USE specification environment is based on UML/OCL.

Table 14.5 OCL constraints

OCL constraint	Description
Invariant	A condition that must always be true. An invariant may be placed on an attribute in a class, and this has the effect of restricting the value of the attribute. All instances of the class are required to satisfy the invariant. An invariant is a predicate and is introduced after the keyword **inv**
Precondition	A condition that must be true before the operation is executed. A precondition is a predicate and is introduced after the keyword **pre**
Postcondition	A condition that must be true when the operation has just completed execution. A post-condition is a predicate and is introduced after the keyword **post**
Guard	A condition that must be true before the state transition occurs

14.5 Tools for UML

There are many tools that support UML (mainly developed by IBM/Rational), and a small selection is listed in Table 14.6.

14.6 Rational Unified Process

Software projects need a well-structured software development process to achieve their objectives, and the *Rational Unified Development Software Process* (RUP) [2] is a way to mitigate risk in software development projects. RUP and UML are often used together, and RUP is

- Use case driven
- Architecture centric
- Iterative and incremental

Table 14.6 UML tools

Tool	Description
Requisite pro	Requirements and use case management tool. It provides requirements management and traceability
Rational software modeller (RSM)	Visual modelling and design tool that is used by systems architects/systems analysts to communicate processes, flows and designs
Rational software architect (RSA)	RSA is a tool that enables good architectures to be created
ClearCase/ClearQuest	These are configuration management/change control tools that are used to manage change in the project

It includes iterations, phases, workflows, risk mitigation, quality control, project management and configuration control. Software projects may be complex, and there are risks that requirements may be missed in the process, or that the interpretation of a requirement may differ between the Customer and developer. RUP gathers requirements as use cases, which describe the functional requirements from the point of view of the users of the system.

The use case model describes what the system will do at a high level, and there is a focus on the users in defining the scope the project. Use cases drive the development process, and the developers create a series of design and implementation models that realize the use cases. The developers review each successive model for conformance to the use case model. The testers verify that the implementation model correctly implements the use cases.

The software architecture concept embodies the most significant static and dynamic aspects of the system. The architecture grows out of the use cases and factors such as the platform that the software is to run on, deployment considerations, legacy systems and non-functional requirements.

A commercial software product is a large undertaking, and the work is decomposed into smaller slices or mini-projects, where each mini-project is a manageable chunk. Each mini-project is an iteration that results in an increment to the product (Fig. 14.6).

Iterations refer to the steps in the workflow, and an increment leads to the growth of the product. If the developers need to repeat the iteration, then the organization loses only the misdirected effort of a single iteration, rather than the entire product. Therefore, the unified process is a way to reduce risk in software engineering. The early iterations implement the areas of greatest risk to the project.

RUP consists of four phases, and these are inception, elaboration, construction and transition (Fig. 14.7). Each phase consists of one or more iterations, where each iteration consists of several workflows. The workflows may be requirements,

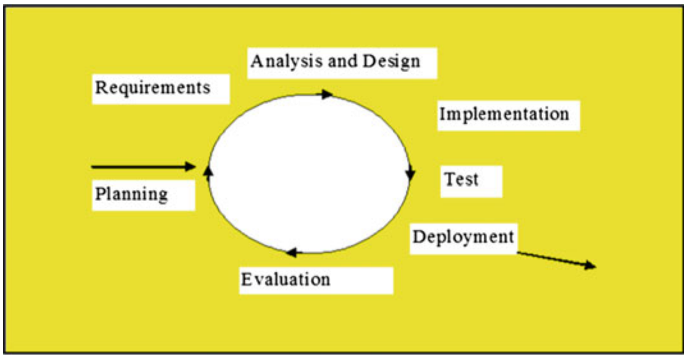


Fig. 14.6 Iteration in rational unified process

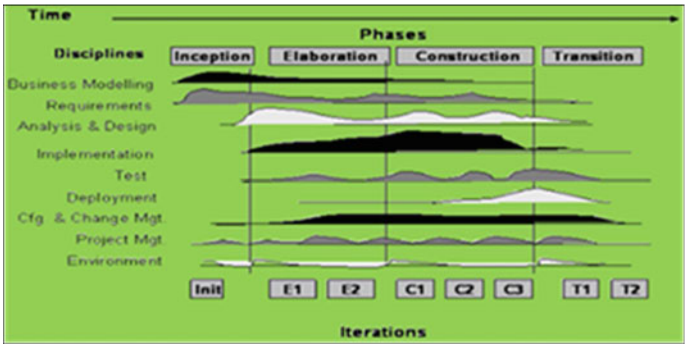


Fig. 14.7 Phases and workflows in rational unified process

analysis, design, implementation and test. Each phase terminates in a milestone with one or more project deliverables.

The inception identifies and prioritizes the most important project risks, and it is concerned with initial project planning, cost estimation and early work on the architecture and functional requirements for the product. The elaboration phase specifies most of the use cases in detail. The construction phase is concerned with building the product and implements all agreed use cases. The transition phase covers the period during which the product moves into the Customer site and includes activities such as training Customer personnel, providing helpline assistance and correcting defects found after delivery.

The waterfall lifecycle has the disadvantage that the risk is greater towards the end of the project, where it is costly to undo mistakes from earlier phases. The iterative process develops an increment (i.e. a subset of the system functionality with the waterfall steps applied in the iteration), then another, and so on, and avoids developing the whole system in one step as in the waterfall methodology. That is, the RUP approach is a way to mitigate risk in software development projects.

14.7 Review Questions

1. What is UML? Explain its main features.
2. Explain the difference between an object and a class.
3. Describe the various UML diagrams.
4. What are the advantages and disadvantages of UML?
5. What is the Rational Unified Process?
6. Describe the workflows in a typical iteration of RUP.
7. Describe the phases in the Rational Unified Process.

8. Describe OCL and explain how it is used with UML.
9. Investigate and describe tools to support UML.

14.8 Summary

The unified modelling language is a visual modelling language for software systems, and it facilitates the understanding of the architecture, and management of the complexity of large systems. It was developed by Rumbaugh, Booch and Jacobson as a notation for modelling object-oriented systems and it provides a visual means of specifying, constructing and documenting such systems. It facilitates the understanding of the architecture of the system and in managing its complexity.

UML allows the same information to be presented in several different ways and at different levels of detail. The requirements of the system are expressed in use cases, and other views include the design view that captures the problem space and solution space, the process view which models the systems processes, the implementation view and the deployment view.

The UML diagrams provide different viewpoints of the system and provide the blueprint of the software. These include class and object diagrams, use case diagrams, sequence diagrams, collaboration diagrams, activity diagrams, state charts, collaboration diagrams and deployment diagrams.

The OCL is an expression language, and the OCL expressions may be used in various places in a UML model to specify the initial value of an attribute, the body of an operation or a condition.

RUP consists of four phases, and these are inception, elaboration, construction and transition. Each phase consists of one or more iterations, and the iteration consists of several workflows. The workflows may be requirements, analysis, design, implementation and test. Each phase terminates in a milestone with one or more project deliverables. The RUP approach is a way to mitigate risk in software development project.

References

1. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Software Modelling Language User Guide* (Addison-Wesley, New York, 1999)
2. J. Rumbaugh et al., *The Unified Software Development Process* (Addison Wesley, New York, 1999)