

CHAPTER 5

UML Overview

This chapter presents an overview of the constituent parts of the UML, how they relate to one another, and how they contribute to a method or process utilizing the UML. The chapter provides a framework for understanding:

- An overall view of the constituent parts of the UML.
- How the UML renders knowledge about a problem or solution.
- How the UML facilitates problem solving and the application of knowledge.

I start with the architecture of the UML and the metamodel that defines the UML. I then introduce the details of how the UML provides a set of diagrams organized around architectural views for viewing problems and solutions. I also discuss how the UML relates to problem solving and how the diagrams may be used. The complete details of each diagram type are found in the reference chapters.

Architecture

Models are complete abstractions of systems, and diagrams are graphical projections of models or subsets of models. The UML is defined in a circular manner, in which a subset of the language notation and semantics is used to specify the language itself. The UML is defined within a conceptual framework for modeling that consists of four distinct layers or levels of abstraction (Chapter 3). This framework is based on the most fundamental UML notation that concepts are depicted as symbols, and relationships among concepts are depicted as paths (lines) connecting symbols. Both of these types of elements may be named. The concepts introduced by the UML are organized around architectural views (Chapter 2) to define the various diagrams. The UML diagrams are used to understand or conceptualize a problem, solve the problem, and implement or realize the solution.

To understand the UML architecture, consider how computer programs and programming languages are related. We develop programs in many different programming languages (C, C++, Java, etc.), and each particular program is

developed using a specific programming language. All programming languages support declarative constructs for declaring data, and procedural constructs for sequential, conditional, and repetitive logic. A model is an abstraction (Chapter 2). Therefore, these programming language concepts may be defined in a model called a *metamodel*. Each particular programming language may be defined in a model that utilizes and specializes the concepts within the metamodel. Each program implemented in a programming language may be defined in a model called a *user model* that utilizes and instantiates the concepts within the model of the appropriate language. This scheme of a metamodel representing computer programming constructs, models representing computer programming languages, and user models representing computer programs exemplifies the architecture of the UML.

The first level, within the conceptual framework for modeling, consists of the most basic element on which the UML is based, as depicted in Figure 5-1. The concept of a "Thing" represents anything that may be defined. Since this is the basis for all models, it is called the *meta-metamodel* and is used to formalize the notion of a concept.

The next level consists of those elements that constitute the UML, as depicted in Figure 5-2. This level introduces the concepts within the object-oriented paradigm. The Class concept is an instance of the meta-metamodel Thing concept. *Classes* are descriptions of a set of objects with common structural features, behavioral features, relationships, and semantics. They are used to model a set of entities with common characteristics. The Object concept is an instance of the meta-metamodel Thing concept. *Objects* are instances of classes. They are used to model particular entities. The Association concept is an instance of the meta-metamodel Thing concept. *Associations* are descriptions of a set of links with common structural features, behavioral features, relationships, and semantics. They are used to model a set of relationships that relate two or more other entities where the relationships have common characteristics. The Link concept is an instance of the meta-metamodel Thing concept. *Links* are instances of associations. They are used to model instances of relationships that relate two or more other objects. Associations relate classes, and links relate objects, as depicted in Figure 5-1 using the Relate relationship. Links are instances of associations, and objects are instances of classes, as depicted in Figure 5-1 using the "Instance of" relationship. These concepts are classes, and the paths connecting them are associations. This layer is called the *metamodel layer* and is used to formalize concepts within a paradigm.

Figure 5-3 completes the metamodel by introducing the multiplicity between the classes. The *multiplicity* specifies how many objects of a class are associated with a single object of the other class in the association. Any number (zero or more) of links may relate the same object, as depicted by a "*" string near the Link class and the Relate association leading to the Object class. A link may relate one or more objects, as depicted by a "1..*" string near the Object class and the Relate association leading to the Link class. The association between the Association class and the Class class specifies that each association must involve one or more classes, and each class may participate in zero or more associations. The association between the Link class and the Association class specifies that each link must be an instance of one association, and each association may have zero or more links.

The association between the Object class and the Class class specifies that each object must be an instance of one or more classes, and each class may have zero or more objects.

The next level consists of UML models. This is the level at which modeling of problems and solutions is done. Figure 5-4 shows that a person can be a member of only one team, and a team will have zero or more members. This layer is called the *model layer* and is used to formalize the concepts for a given subject, in this case, how people are related to teams. This layer corresponds to class diagrams, sequence diagrams, or collaboration diagrams.

The next level consists of those elements that exemplify UML models. Figure 5-5 shows that Si, Andy, Phillip, and Nora are all people. Si and Andy are members of the Requirements team, and Nora and Phillip are members of an unnamed team. Each object is depicted by specifying and underlining the name of the object followed by a colon and the class of the object. Each link is depicted by specifying the name of the association on the path connecting the objects. Figure 5-6 shows an ill-formed user model since Si belongs to both teams, whereas our model of Figure 5-4 shows that a person can only belong to one team. This layer is called the *user model layer* and is used to formalize specific expressions about a given subject, in this case, that Si and Andy belong to the Requirements team and Nora and Phillip belong to the unnamed team. This layer corresponds to object diagrams or to scenario diagrams of sequence diagrams or collaboration diagrams.

This brief example demonstrates that a metamodel is capable of defining concepts and relationships among concepts. However, it is not used to describe the notation used to render the concepts. Furthermore, because the concepts are defined using natural language, they may have various ambiguous interpretations. Thus, a formal constraint language similar to the Object Constraint Language (OCL) is used to specify constraints or rules for the concepts within the metamodel, and a notation guide is used to depict how concepts are rendered. The metamodel used in this example is not the actual UML metamodel. The UML metamodel is significantly more involved and is briefly discussed in the next section.



Figure 5-1: Meta-metamodel Example

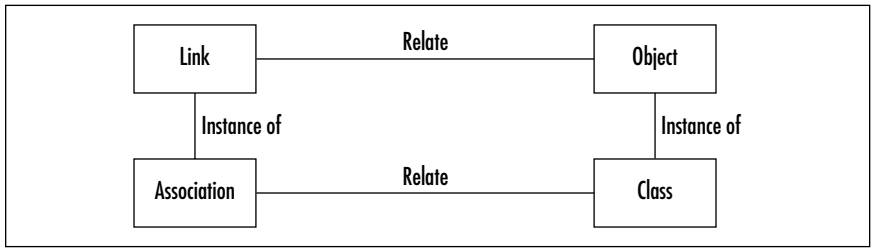


Figure 5-2: Incomplete Metamodel Example

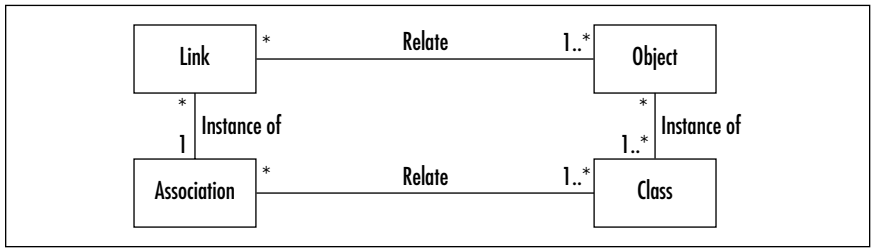


Figure 5-3: Complete Metamodel Example

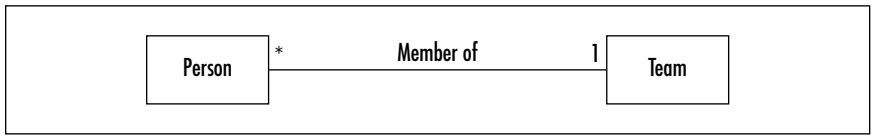


Figure 5-4: Model Example

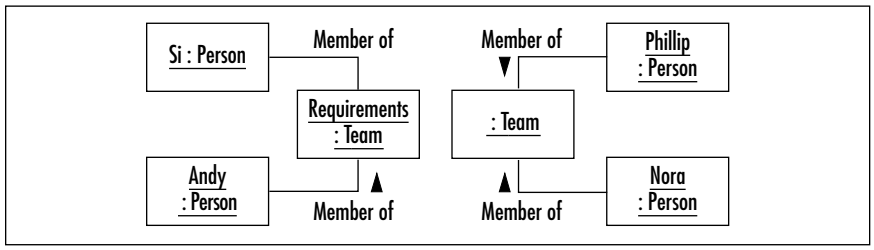


Figure 5-5: User Model Example

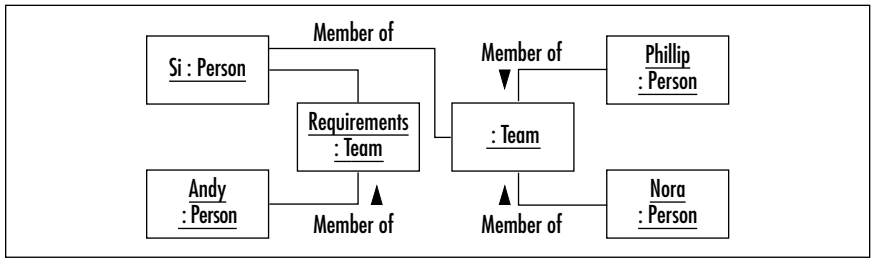


Figure 5-6: Ill-formed User Model Example

Metamodel

The UML metamodel is organized into a collection of logical packages (Figure 5-7). Packages are general-purpose grouping mechanisms used to organize semantically related model elements. The metamodel organizes metamodel elements within packages. Elements within the same package have strong cohesion with

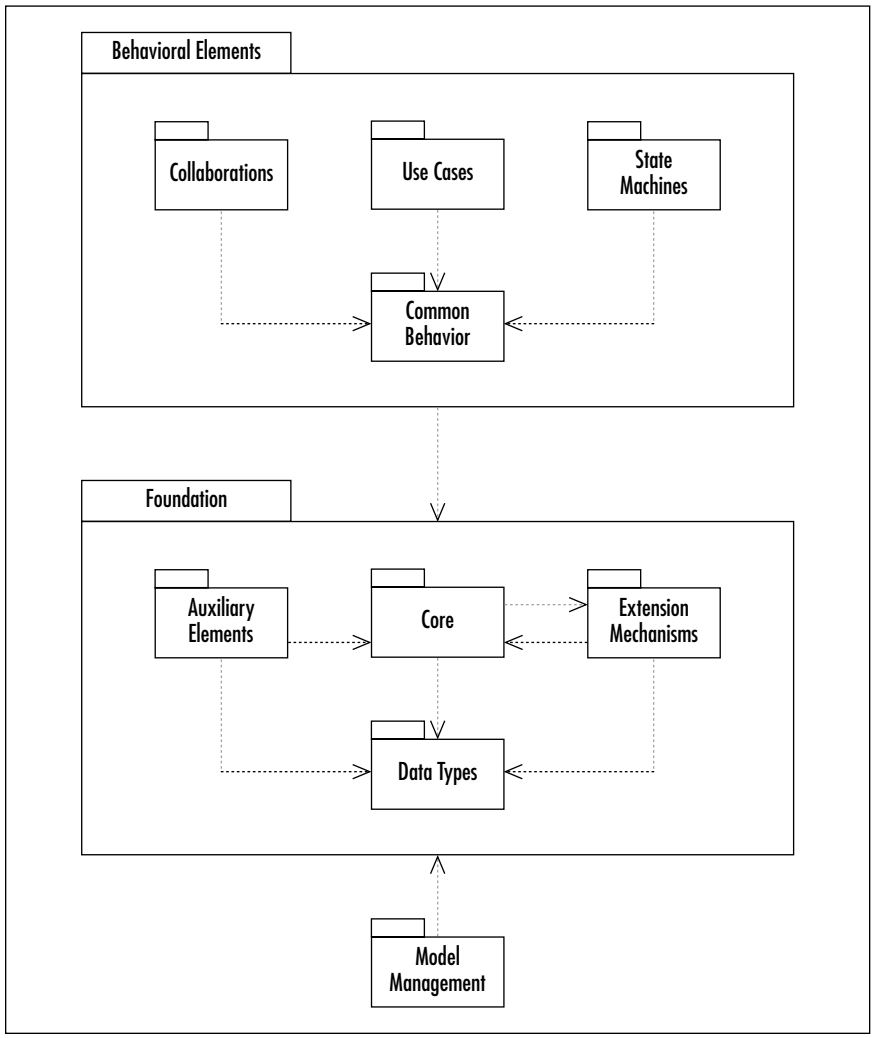


Figure 5-7: The UML Metamodel Architecture

each other and loose coupling with elements in other packages. Dashed arrows between two packages or model elements indicate that a relationship exists between the elements, where the source of the dashed arrow is dependent on the presence or existence of the target of the dashed arrow. The UML is defined within the following packages.

- The *Foundation package* defines the concepts required for structural modeling and the mechanisms for extending the language. These modeling elements let you model the structure of a system. Other modeling elements enable the language to be extended. The Foundation package is composed of the following packages:

  - The *Data Types package* defines different data types used within the UML metamodel.
  - The *Core package* defines the most fundamental concepts of the UML. It actually contains sufficient semantics to define the remainder of the UML, but other packages extend these modeling elements to meet that purpose. Core concepts include classes and associations. The concepts within this package are the subject of Chapter 7.
  - The *Auxiliary Elements package* defines other fundamental concepts of the UML that extend the Core package. These concepts include notes or comments, dependencies, templates, components, and nodes. The concepts within this package are appropriately distributed over Chapters 6, 7, 13, and 14 in correlation to how they are used within diagrams.
  - The *Extension Mechanisms package* defines the concepts required to customize and extend the UML. These concepts are stereotypes, tagged values, and constraints. The concepts within this package are the subject of Chapter 15.
- The *Behavioral Elements package* defines the concepts required to model a system's behavior. It is composed of the following packages:
  - The *Common Behavior package* defines the core concepts required for dynamic elements. It provides the foundation for use cases, collaborations, and state machines. These concepts include signals and actions. The concepts within this package are appropriately distributed over Chapters 8, 9, 10, 11, and 12 in correlation to how they are used within diagrams.
  - The *Use Cases package* defines the modeling of an entity's functions, or what it offers to outside users. These concepts include use cases and actors. The concepts within this package are the subject of Chapter 8.
  - The *Collaborations package* defines how different elements of a model interact to realize use cases. These concepts include collaborations and interactions. The concepts within this package are appropriately distributed over Chapters 9 and 10.
  - The *State Machines package* defines how control and data flow among elements of a model interacting to realize use cases. These concepts include events, states, and transitions. The concepts within this package are appropriately distributed over Chapters 11 and 12.
- The *Model Management package* defines how modeling elements are organized. It defines packages as grouping units for other model elements. The concepts within this package are the subject of Chapter 6.

Architectural Views and Diagrams

The UML metamodel elements are organized into diagrams. Different diagrams are used for different purposes depending on the angle from which you are viewing the system. The different views are called *architectural views* (Figure 5-8). Architectural views facilitate the organization of knowledge, and diagrams enable the communication of knowledge. The knowledge itself is within the model or set of

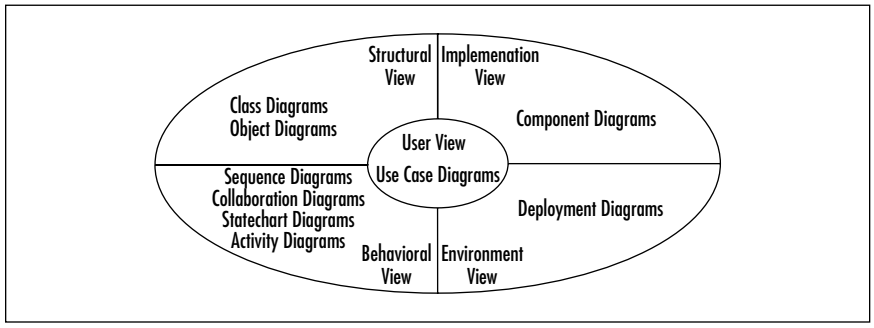


Figure 5-8: Model View and Diagrams

models that focuses on the problem and solution. The architectural views and their diagrams are summarized below.

- The *user model view* encompasses a problem and solution from the perspective of those individuals whose problem the solution addresses. This view presents the goals and objectives of the problem owners and their requirements of the solution. This view is composed of *use case diagrams*. These diagrams describe the functionality provided by a system to external interactors. These diagrams contain actors, use cases, and their relationships. Use case diagrams are the subject of Chapter 8.
- The *structural model view* encompasses the static, or structural, aspects of a problem and solution. This view is also known as the static or logical view. This view is composed of the following diagrams:
  - *Class diagrams* describe the static structure of a system, or how it is declared rather than how it behaves. These diagrams contain classes and associations.
  - *Object diagrams* describe the static structure of a system at a particular time during its life. These diagrams contain objects and links.

  Class and object diagrams are the subject of Chapter 7.
- The *behavioral model view* encompasses the dynamic, or behavioral, aspects of a problem and solution. This view is also known as the dynamic, process, concurrent, or collaborative view. This view is composed of the following diagrams:
  - *Sequence diagrams* render the specification of behavior. These diagrams describe the behavior provided by a system to interactors. These diagrams contain classes that exchange messages within an interaction arranged in time sequence. In generic form, these diagrams describe a set of message exchange sequences among a set of classes. In instance form (scenarios), these diagrams describe one actual message exchange sequence among objects of those classes. Sequence diagrams are the subject of Chapter 9.
  - *Collaboration diagrams* render how behavior is realized by components within a system. These diagrams contain classes, associations, and their

message exchanges within a collaboration to accomplish a purpose. In generic form, these diagrams describe a set of classes and associations involved in message exchange sequences. In instance form (scenarios), these diagrams describe a set of objects of those classes, links conforming to the associations, and one actual message exchange sequence that is consistent with the generic form and uses those objects and links. Collaboration diagrams are the subject of Chapter 10.

  - *Statechart diagrams* render the states and responses of a class participating in behavior, and the life cycle of an object. These diagrams describe the behavior of a class in response to external stimuli. State diagrams are the subject of Chapter 11.
  - *Activity diagrams* render the activities or actions of a class participating in behavior. These diagrams describe the behavior of a class in response to internal processing rather than external events. Activity diagrams describe the processing activities within a class and are the subject of Chapter 12.

- The *implementation model view* encompasses the structural and behavioral aspects of the solution's realization. This view is also known as the component or development view and is composed of *component diagrams*. These diagrams describe the organization of and dependencies among software implementation components. These diagrams contain components and their relationships. Component diagrams are the subject of Chapter 13.
- The *environment model view* encompasses the structural and behavioral aspects of the domain in which a solution must be realized. This view is also known as the deployment or physical view. This view is composed of *deployment diagrams*. These diagrams describe the configuration of processing resource elements and the mapping of software implementation components onto them. These diagrams contain nodes, components, and their relationships. Deployment diagrams are the subject of Chapter 14.

Mechanisms

When solving a problem, diagrams are used to communicate information about the problem and solution. When modeling and diagramming, there are a number of fundamental mechanisms that enable us to create better or more precise and communicable models. Thus, *mechanisms* are means or tools for approaching modeling and diagramming.

Fundamental to all mechanisms is the idea that information is being communicated for a purpose. When deciding what diagram to use, consider the reason for the communication. That is, what is the question or questions that the communication is addressing? Next, consider what diagram or set of diagrams may be used to most effectively communicate the response. The mechanisms discussed below may be utilized best if this notion of communication is kept in mind.

The first mechanism involves the perspective through which a diagram is rendered or interpreted. A *perspective* (Figure 5-9) defines a particular point of view from which to draw or read a diagram. The perspective must clearly identify the diagram's subject and goal. One of the following three perspectives should be selected:

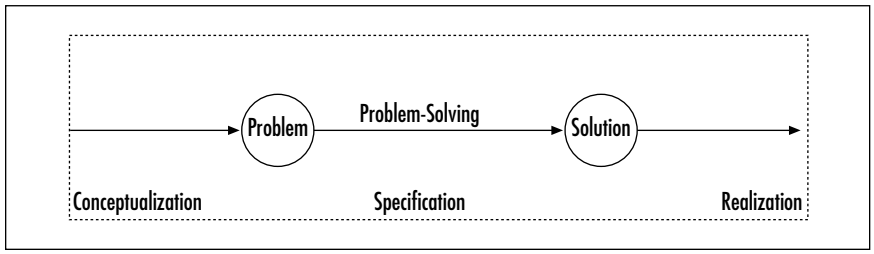


Figure 5-9: Perspectives

- The *conceptualization* perspective is concerned with the problem and is directed at representing the problem in some malleable and communicable form. This is also known as the conceptual perspective of problem solving. This perspective addresses the following questions: What is the problem? What are the requirements of the problem that its solution must satisfy?
- The *specification* perspective is concerned with the problem and solution and is directed at specifying the solution that satisfies the requirements imposed by the problem. This is also known as the logical perspective of problem solving. This perspective addresses the following questions: What is the solution? What are the specifications of the solution that satisfies the requirements imposed by the problem?
- The *realization* perspective is concerned with implementing the solution and is directed at realizing the representation of the solution in some concrete and usable form. This is also known as the physical perspective of problem solving. This perspective addresses the following question: What tools, programs, or activities will realize the solution?

Once a perspective is selected, it should be made evident on the diagram, perhaps using a note (Chapter 6) attached to the diagram. When the diagram is interpreted, the reader will have a clear viewpoint from which to read it. This makes communication using the diagram much more effective, and the reader need not be left guessing as to what perspective was used to render the diagram.

The second mechanism involves the use of various dichotomies. A *dichotomy* defines how something may be viewed from two different perspectives. The following dichotomies are clearly evident:

- The *type-instance* dichotomy allows a generic descriptor to describe many individual items. This is the relationship that exists between a class and its objects, an association and its links, and an interaction and its scenarios. This dichotomy lets you create object diagrams that are combined into class diagrams, or class diagrams that are validated by considering their instantiated object diagrams. This dichotomy also enables scenario diagrams to validate sequence and collaboration diagrams.
- The *specification-realization* dichotomy specifies that knowledge gathered using the specification perspective is elaborated or translated into the realization perspective. This dichotomy allows for the capturing of a specification before a realization: for example, an operation specification before a method

realization, a type specification before a class realization, or a sequence diagram specification of a use case before a collaboration diagram realization of the use case.

- The *static-dynamic,* or *structural-behavioral,* dichotomy specifies that knowledge gathered via the structural model view and the behavioral model view are complementary. This dichotomy enables the use of class diagrams to determine the participants of behavioral view diagrams, and the use of behavioral view diagrams to highlight the messages, events, and operations that participants from the structural model view must support. This dichotomy enables the structural model view and the behavioral model view to evolve concurrently and cooperatively from conceptualizing a problem, solving the problem, and realizing the solution.

The use of these dichotomies allows us to collect and represent knowledge using UML diagrams as we progress in problem solving. The dichotomies allow us to relate different diagrams from different perspectives through a problem-solving process.

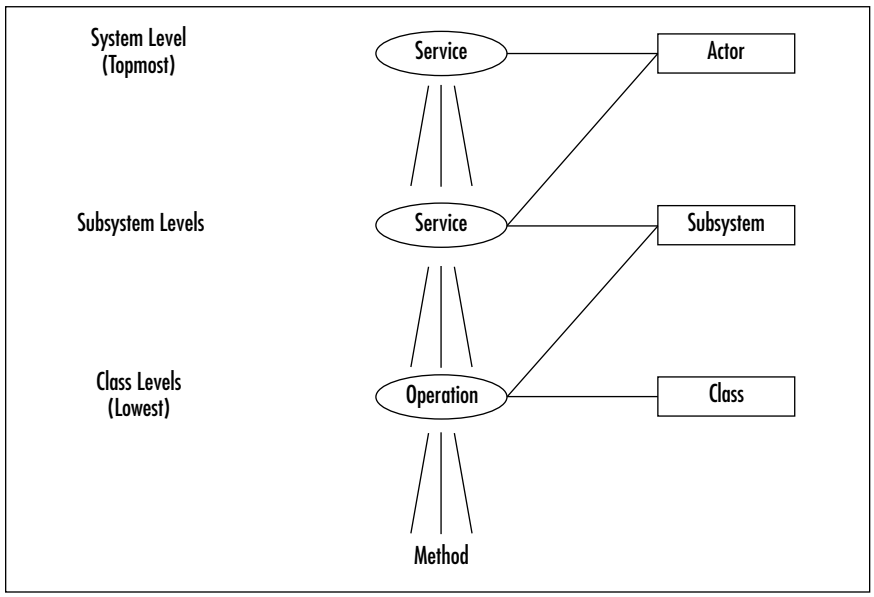


Figure 5-10: Levels of Abstraction

The third mechanism involves the use of *layers* or *levels of abstraction* through which a diagram is rendered or interpreted. A layer defines a particular level of abstraction (Figure 5-10) and establishes a level of detail at which attention and concentration are focused regarding the subject (problem or solution). The level of abstraction at which a diagram is drawn or interpreted must be clearly identified. One of the following layers should be selected:

- The *system level* specifies the complete services offered by a system to actors.

- *Subsystem levels* realize the next higher level of services by specifying the complete services offered by a system or subsystem. Any number of subsystem levels may be used.
- *Class levels* realize the next higher level of services by specifying functional fragments or operations offered by a class. Any number of class levels may be used.
- The *method level* realizes the next higher level of services.

This mechanism is crucial for consolidating all the UML diagrams into a coherent body of knowledge. This mechanism is the key for appropriately using use cases and is the subject of Chapters 6 and 8.

The final mechanism involves the means for customizing and extending the UML. These *extension mechanisms* include the following:

- *Stereotypes* are used for classifying or marking model elements and introducing new types of modeling elements.
- *Tagged values* are used for specifying properties or characteristics of model elements.
- *Constraints* are used for specifying semantics or conditions that must be maintained as true for model elements.

These extensibility mechanisms are fundamental to ensuring that the UML will evolve rather than be redefined to meet the changing needs and demands of its users. They are discussed in Chapter 15.

Problems, Solutions, and Problem Solving

The UML enables and promotes (but does not require or mandate) a process that is

- *Use case driven*, which means that use cases are utilized to
  - Capture requirements and define the behavior or services required of a system by actors, services required of a subsystem by other subsystems, and operations required of a class by other classes.
  - Analyze a problem to address the following questions: What is the problem? What are the requirements of the problem that its solution must satisfy?
  - Design a solution to address the following question: What are the specifications of the solution that satisfies the requirements imposed by the problem?
  - Validate the realization of the solution. Use cases that capture the requirements of the problem are used to validate the solution against the problem.
  - Manage and provide focus for a problem-solving effort. Use cases are work units that may be distributed over a complete project development cycle.
- *Architecture-centric*, which means that architecture is used to

  - Methodically and systematically specify, realize, and validate a solution to a problem.
  - Capture (acquire), enable the communication of (share), and enable the leveraging (utilizing) of knowledge. The knowledge that transpires within a problem-solving effort is organized around architectural views. This organization also emphasizes reuse within the effort and across multiple efforts.
  - Manage complexity and maintain integrity and focus as a solution to a problem evolves and is realized.
- *Iterative and incremental.* An iterative process is repetitive, and an incremental process is evolutionary. A problem-solving effort repeatedly applies a process to evolve a solution to a problem. An iterative and incremental process enables
  - A problem to change as required by those who own the problem. This allows the requirements imposed by a problem on its solution to evolve as a problem evolves.
  - Progressive integration of the different parts of a complex solution rather than a single point of integration.
  - The management of risks within each iteration rather than at a single or few points over the life of a problem-solving effort.
  - A focus on an architecture that evolves over time.
  - A focus on reuse since reusable elements may be identified at the end of each iteration and may be used within the following iterations.
  - A focus on self-improvement and refinement. Each iteration improves on previous iterations and provides lessons for future iterations.

Problems and Solutions

Problems and solutions occur within a context. The problem must be understood in order to be solved. The solution to a problem must be understood in order to be constructed and utilized. The solution must be organized in order to facilitate its realization and adhere to the various constraints of the context (available computer systems, development time, etc.) in which it will be realized. To solve the problem, appropriate knowledge about the problem and solution must be captured, organized around decisions regarding the problem and solution, and depicted using some language so that it can be communicated and leveraged in the problem-solving process.

When deciding what diagram to use for communication, consider the reason for the communication. That is, what is the question or questions the communication is addressing? Then consider what diagram or set of diagrams may be used to most effectively communicate the response. This decision centers on what aspect of a system or society of objects is to be emphasized in the response.

Each diagram emphasizes a particular aspect of the society of objects that constitute a system (Figure 5-11):

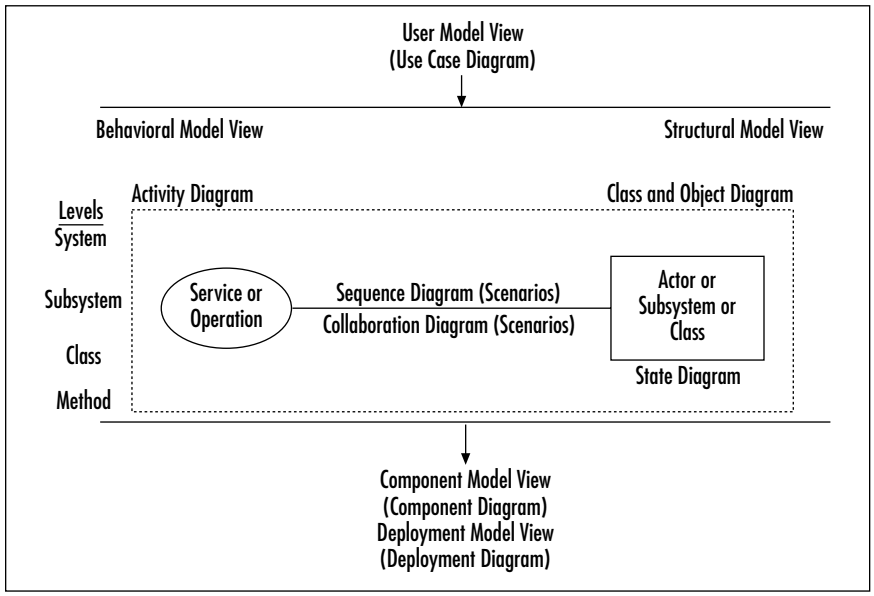


Figure 5-11: Model Views and Diagrams with Levels of Abstraction

- The user model view and use case diagrams emphasize the functionality provided by a society. Use these diagrams to emphasize services and operations that a society offers to entities outside the society.

  Use case diagrams emphasize *functionality*, the services offered by a system to its users.

- The structural model view is composed of the following diagrams:
  - Class diagrams emphasize the static structure of a society.

    Class diagrams emphasize *structure*.
  - Object diagrams emphasize the static structure of a society at a particular time during its life. Use these diagrams to exemplify and validate the static structure of a society.

    Object diagrams emphasize *examples of structure*.
- The behavioral model view is composed of the following diagrams:
  - Sequence diagrams emphasize the messages exchanged over time among the objects within a society. Use scenarios to exemplify and validate sequence diagrams.

    Sequence diagrams emphasize *classes and messages*.

    Scenario sequence diagrams emphasize *objects and messages*.
  - Collaboration diagrams emphasize the messages exchanged over time among the objects and their links within a society. Use scenarios to exemplify and validate collaboration diagrams.

Collaboration diagrams emphasize *classes, messages, and associations for propagating messages*.

Scenario collaboration diagrams emphasize *objects, messages, and links for propagating messages*.

- Statechart diagrams emphasize how an object changes or responds to external stimuli within a society.

  Statechart diagrams emphasize *changes of state due to messages*.

- Activity diagrams emphasize how an object changes or responds to internal processing within a society.

  Activity diagrams emphasize the *flow of control and information*.

- The implementation model view and component diagrams emphasize the packaging of an object within a society as a solution.

  Component diagrams emphasize the *packaging of a solution*.

- The behavioral model view and deployment diagrams emphasize the deployment of objects within a society as a solution within an environment.

  Deployment diagrams emphasize the *deployment of a solution in an environment*.

A method must specify the order in which these diagrams are created; however, the following generic order is common:

1. Use case diagrams are used to drive the problem-solving process.
2. Structural diagrams and behavioral diagrams are concurrently used to elaborate a problem or solution. Focus should shift between these two types of diagrams until a solution is derived from a problem.
   - Structural (class and object) diagrams are used to initiate this concurrent process if the structural features of the problem and solution are more easily comprehensible or understood.
   - Sequence diagrams (with scenarios) followed by collaboration diagrams (with scenarios) are used to initiate this concurrent process if the dynamic features of the problem and solution are more easily comprehensible or understood.
   - State diagrams and activity diagrams are used to support the other diagrams.
3. Component diagrams are used to package the solution to a problem.
4. Deployment diagrams are used to determine how a solution is deployed in a specific environment.

Problem Solving

Problems and solutions are societies of objects. Methods specify how problem-solving efforts may be executed. They specify how knowledge is captured and communicated regarding a problem and solution, and the approach used to solve the problem and derive a solution.

Knowledge regarding problems and solutions is communicated using diagrams. While each diagram emphasizes a particular aspect of the society of objects that

constitute a system (problem or solution), a method specifies which diagrams to use and the perspective and the level of abstraction used to render and interpret these diagrams. A method also specifies how this knowledge will be utilized to realize a solution to the problem, and must address the following questions:

- What activities are done?
- When are activities done?
- Who does the activities?
- Why are activities done?
- How are activities done?
- Where (and in what situations) ought the activities be done?

Development Cycles and Phases

	Inception	Elaboration	Construction	Transition
Perpective	Conceptualization	All	All	Realization
Level of Abstraction	System	System	All	System

Figure 5-12: Development Phases with Perspectives and Levels of Abstraction

Each phase of a development cycle (Chapter 2) focuses on a particular perspective and one or more levels of abstraction (Figure 5-12):

- The inception (initiating) development phase is primarily concerned with the conceptualization perspective and focuses on the system level of abstraction. This phase is used for understanding or forming a notion of the problem and the rationale for solving it (scoping and business case).
- The elaboration (planning) development phase is concerned with all perspectives and focuses on the system level of abstraction. This phase is used for
  - Understanding or forming a notion of the problem to determine the requirements the problem imposes on its solution.
  - Establishing and verifying the foundation for the overall solution.
  - Distributing the requirements among the iteration cycles of the construction phase.
- The construction (executing) development phase is concerned with all perspectives and focuses on all levels of abstraction. This phase is used for
  - Understanding or evolving the requirements the problem imposes on its solution so that the problem can be elaborated and the solution can be specified.
  - Elaborating the solution specification.
  - Updating the foundation for the overall solution and the foundation required to support the specific solution for the iteration cycle requirements.
  - Producing the solution for the iteration cycle requirements.

  - Verifying the solution for the iteration cycle requirements.
  - Providing or integrating the solution or a subset thereof.
- The transition (closing) development phase is primarily concerned with the realization perspective and focuses on the system level of abstraction. This phase is used for providing and integrating or delivering the solution.

Iteration Cycles and Phases

	Req.	Analysis	Design	Implement.	Validation	Deployment
Perpective	Concept.	Specific.	Realization	Realization	Realization	Realization
Level of Abstraction	Subsystem, Class	Subsystem, Class	Subsystem, Class, Method	All	All	All

Figure 5-13: Iteration Phases with Perspectives and Levels of Abstraction

Each phase of an iteration cycle (Chapter 2) focuses on a particular perspective and one or more levels of abstraction (Figure 5-13):

- The management and support (controlling) phases are concerned with all perspectives and focus on all levels of abstraction as required by the development components.
- Within the elaboration (planning) development phase, the following iteration phases apply:
  - The requirements phase and analysis phase are primarily concerned with the conceptualization perspective and focus on the system level of abstraction. These phases are for understanding or forming a notion of the problem to determine the requirements of the solution and distribute the requirements among iteration cycles of the construction phase.
  - The design phase is primarily concerned with the specification perspective and focuses on the system level of abstraction. This phase is for establishing and verifying the foundation for the overall solution.
- Within the construction (executing) development phase, the following iteration phases apply:
  - The requirements phase is primarily concerned with the conceptualization perspective and focuses on the subsystem and class levels of abstraction. This phase is for understanding or elaborating the requirements the problem imposes on its solution so that the problem can be elaborated and the solution can be specified.
  - The analysis phase is primarily concerned with the specification perspective and focuses on the subsystem and class levels of abstraction. This phase is for elaborating the solution specification.
  - The design phase is primarily concerned with the realization perspective and focuses on the subsystem, class, and method levels of abstraction. This phase is for updating the foundation for the overall solution and the

foundation required to support the specific solution for the iteration cycle requirements.

- The implementation phase is primarily concerned with the realization perspective and focuses on all levels of abstraction. This phase is for producing the solution for the iteration cycle requirements.
- The validation (and integration) phase is primarily concerned with the realization perspective and focuses on all levels of abstraction. This phase is for verifying the solution for the iteration cycle requirements.
- The deployment phase is primarily concerned with the realization perspective and focuses on all levels of abstraction. This phase is for providing and integrating or delivering the solution or a subset thereof.

,ch05.12795 Page 128 Thursday, June 14, 2001 4:18 PM