Introducing Modeling and the Software Development Process

Objectives

Upon completion of this module, you should be able to:

- Describe the Object-Oriented Software Development (OOSD) process
- Describe how modeling supports the OOSD process
- Describe the benefits of modeling software
- Explain the purpose, activities, and artifacts of the following OOSD workflows (disciplines): Requirements Gathering, Requirements Analysis, Architecture, Design, Implementation, Testing, and Deployment

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Booch, Grady, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison Wesley Longman, Inc., 1999.
- Jacobson, Ivar, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley. Reading. 1999
- Jacobson, Ivar. *Object-Oriented Software Engineering*. Harlow: Addison Wesley Longman, Inc., 1993
- Larman, Craig. *Applying UML and Patterns (3rd ed)*. Upper Saddle River: Prentice Hall, 2005.
- Jacobson, Ivar, Grady Booch, James Rumbaugh. *The Unified Modeling Language Reference Manual (2nd ed)*. Addison-Wesley, 2004

Exploring the OOSD Process

The software industry has experienced exponential growth and change in the half-century from its birth. There has been a great deal of change in the technologies of programming: languages, operating systems, networking, communication protocols, component-based Application Programming Interfaces (APIs), and application server software. The software development process has changed along with the technologies. When using Object-Oriented (OO) technologies for development, then you should use OOSD processes because the OO technologies influence the software development processes.

There have also been many changes about how software projects are organized and managed. Who determines what the software is required to do? How is the software solution created? How is the development life cycle managed? The answers to these questions guide the OOSD process.

Note – There is no "one standard OOSD process." The concepts and activities presented in the majority of this course demonstrates a generic (vanilla) approach. The goal of this course is to provide you with techniques that you can use in any process (methodology). Later in the course, you will be introduced to some of the specific OO development processes that are available.

Describing Software Methodology

In Webster's New Collegiate Dictionary, *methodology* is defined as "*a body of methods, rules, and postulates employed by a discipline*." In software development, a methodology refers to the highest-level of organization of the development process itself.

Note – Technically, the term methodology means "the science which studies methods" and the term method means "the composition of a language and a process." Therefore, it is more appropriate to say that the Unified Process (UP) is a method, not a methodology. However, it is common practice to refer to UP (and other OOSD methods) as a methodology. This is the terminology used in this course.

Over the history of the software industry, there have been hundreds of methodologies developed. OO methodologies incorporate object-oriented concepts throughout the OOSD process. Many modern OO methodologies compose the development process into large-scale *phases*, such as Inception, Elaboration, Construction, and Transition. These phases are composed of *workflows (disciplines)* and these workflows are composed of specific *activities*. Activities involve workers and artifacts. A *worker* is a person that performs the activity. An *artifact* is a tangible piece of information that is produced by an activity. Artifacts such as diagrams, documents, and the software code itself are produced with tools. The Unified Modeling Language (UML) is one of our most powerful tools for modeling software.

Note – The course uses the term *workflow*. However, the Object Management Group (OMG) is recommending a new term *discipline* to replace workflow.

Figure 2-1 illustrates the hierarchical relationships between methodology phases, workflows, activities, and the tools that create artifacts.

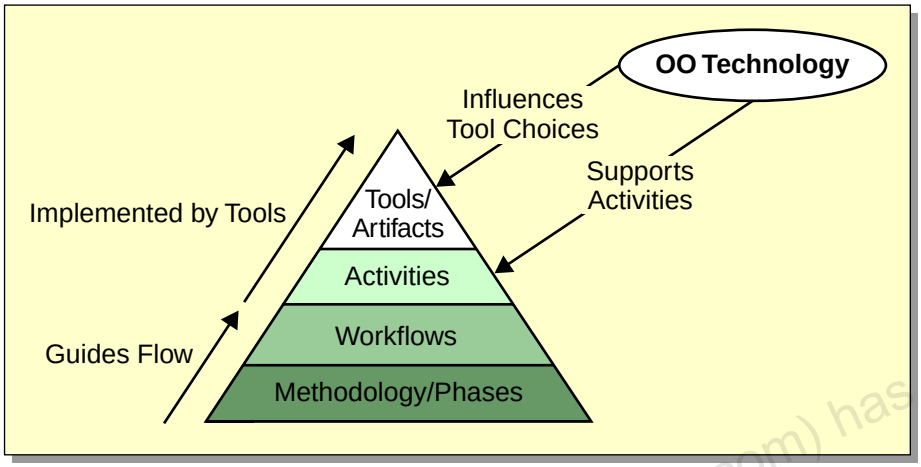


Figure 2-1 OOSD Hierarchy Pyramid[1]

To support the activities, the development team uses various tools to analyze, model, and construct the software solution. These tools include word processors, the UML, UML modeling tools, advanced text editors, and integrated development environments.

An artifact produced in one activity might be an input into another activity. For example, a Use Case diagram (which describes the intended behavior of the system) is used during the Design workflow to determine the software components needed to satisfy the functional requirements defined by the use cases.

Artifacts are meant to be long-lived, but several activities often produce short-lived modeling artifacts that are discarded because the information gained is either documented elsewhere more formerly or is found to be of no value.

Artifacts can be documents, diagrams, and even a functioning system. The goal of software development is to produce a functioning system (the final artifact) that satisfies the requirements of the business owner, users, and other client-side stakeholders.

1. This OOSD hierarchy pyramid was influenced by Jacobson's pyramid of a "rational enterprise philosophy." The hierarchy of Jacobson's pyramid is slightly different than the one shown in Figure 2-1. It has the following structure: Architecture at the bottom, then Method, then Process, and then Tools at the top. Read *Object-Oriented Software Engineering* (section 1.2) for more information.

Listing the Workflows of the OOSD Process

Workflows consist of activities performed by workers on the development team. There are many possible workflows, but this course focuses on the following seven:

- *Requirements Gathering*

  Determine the requirements of the system by meeting the business owner and users of the proposed system.

- *Requirements Analysis* (or just Analysis)

  Analyze, refine, and model the requirements of the system.

- *Architecture*

  Identify risk in the project and mitigate the risk by modeling the high-level structure of the system.

- *Design*

  Create a Solution model of the system that satisfies the system requirements.

- *Implementation*

  Build the software components defined in the Solution model.

- *Testing*

  Test the implementation against the expectations defined in the requirements.

- *Deployment*

  Deploy the implementation into the production environment.

There is a single common goal that binds all of these workflows together: *To deploy a tested software system that satisfies all of the requirements defined by the business owner and users.*

Note – Some people include *maintenance* in the list of workflows. This course considers maintenance as a new revision of the software and each revision embodies the whole development process.

Note – Some companies use different terminology for these workflows (disciplines). Implementation is sometimes called build, and deployment is sometimes called implementation.

Describing the Software Team Job Roles

A stakeholder is any person or group that has an interest in the project. The set of stakeholders includes users and managers on the client-side and the complete project team on the development side.

As previously mentioned, workers perform activities which generate artifacts that document, model, and implement the software solution. There are many job roles on a software project; however, some workers might fulfill multiple job roles. Figure 2-2 shows a common set of OOSD job roles.

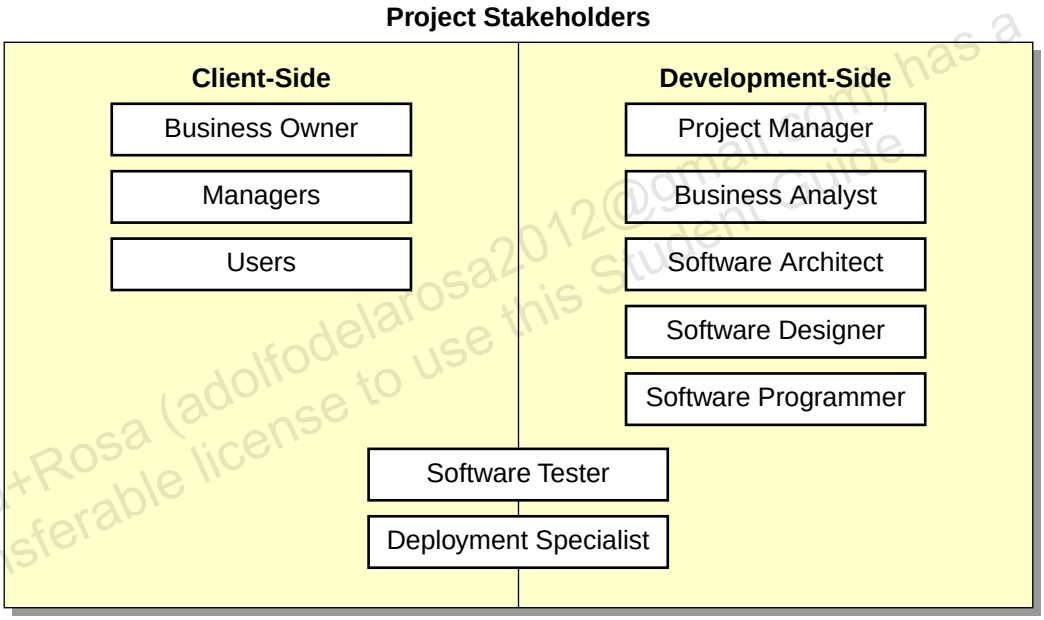


Figure 2-2 Software Development Job Roles

- *Business Owner*

  The lead stakeholder on the client-side of the project. This person is responsible for making final decisions about the behavior of the system.

- *Users*

  Any person that will be using the proposed system. For internal systems, these are usually employees of the client company. For Web-based systems, this could be any Internet user.

- *Managers*

  The people that are the managers of internal users.

- *Project Manager*

Manages the software development project. This is usually *not* a technical manager. They manage the budget, resources, and schedule of the project. They are often the account manager for the client (if consulting).

- *Business Analyst*

  Gathers requirements from the client-side stakeholders and analyzes the functional requirements by modeling the *enduring business themes* of the system. This role is responsible for the Business Domain diagrams, which include Use Case diagrams and Business Domain (Analysis) class diagrams.

  Synonyms: Enterprise Modeler, Domain Modeler, and System Analyst

- *Software Architect*

  Defines the architecture of the system, leads the development of the architectural baseline during the Inception and Elaboration phases, analyses the NFRs, identifies project risk, and creates a risk mitigation plan.

- *Software Designer*

  Creates the Solution model of the system based on the Business Domain diagrams within the framework of the architecture.

  Synonyms: Software Engineer

- *Software Programmer*

  Implements the software solution. In many small development teams, the programmer and designer roles are filled by the same person.

  Synonyms: Software Developer

- *Software Tester*

  Tests the implementation to verify that the system meets the requirements (both FRs and NFRs). This role might include the development team for unit and integration testing, as well as the client-side Quality Assurance (QA) personnel for acceptance testing.

  Synonyms: QA Specialist, Test Engineer

- *Deployment Specialist*

  Deploys the implementation onto the production platform. This role might include several other job roles: System Administrator, Network Administrator, Script Writer, and so on. This role is performed by the development team during construction, but when the system goes to production this role should be filled by the client organization.

Exploring the Requirements Gathering Workflow

The requirements of a system are divided into two fundamental categories: functional requirements (FRs) and non-functional requirements (NFRs). The FRs describe the behavior of the system relative to the perspective of the actors that use the system. High-level FRs are visualized as *use cases*.

Note – This course will first develop use cases (high-level FRs) and then drill down to determine the lower-level behavior for each use case, which implicitly includes the lower-level FRs.

The NFRs describe the quality of service of the system. These requirements include such characteristics as performance (measured in response time), throughput (measured by how many simultaneous users can be accommodated), and so on.

Workflow Purpose and Job Roles

The purpose of the Requirements Gathering workflow is to determine the requirements of the system by meeting the business owner and users of the proposed system. Table 2-1 shows the description of this workflow.

Table 2-1 Requirements Gathering Workflow

Workflow	Purpose	Description
Requirements Gathering	Determine *what* the system must do	Determine: • With whom the system interacts (actor) • What behaviors (called use cases) that the system must support • Detailed behavior of each use case, which includes the low-level FRs • Non-functional requirements

The Requirements Gathering workflow activities are usually performed by business analyst and the architect job roles.

Workflow Activities and Artifacts

Figure 2-3 illustrates the activities the Requirements Gathering workflow might include.

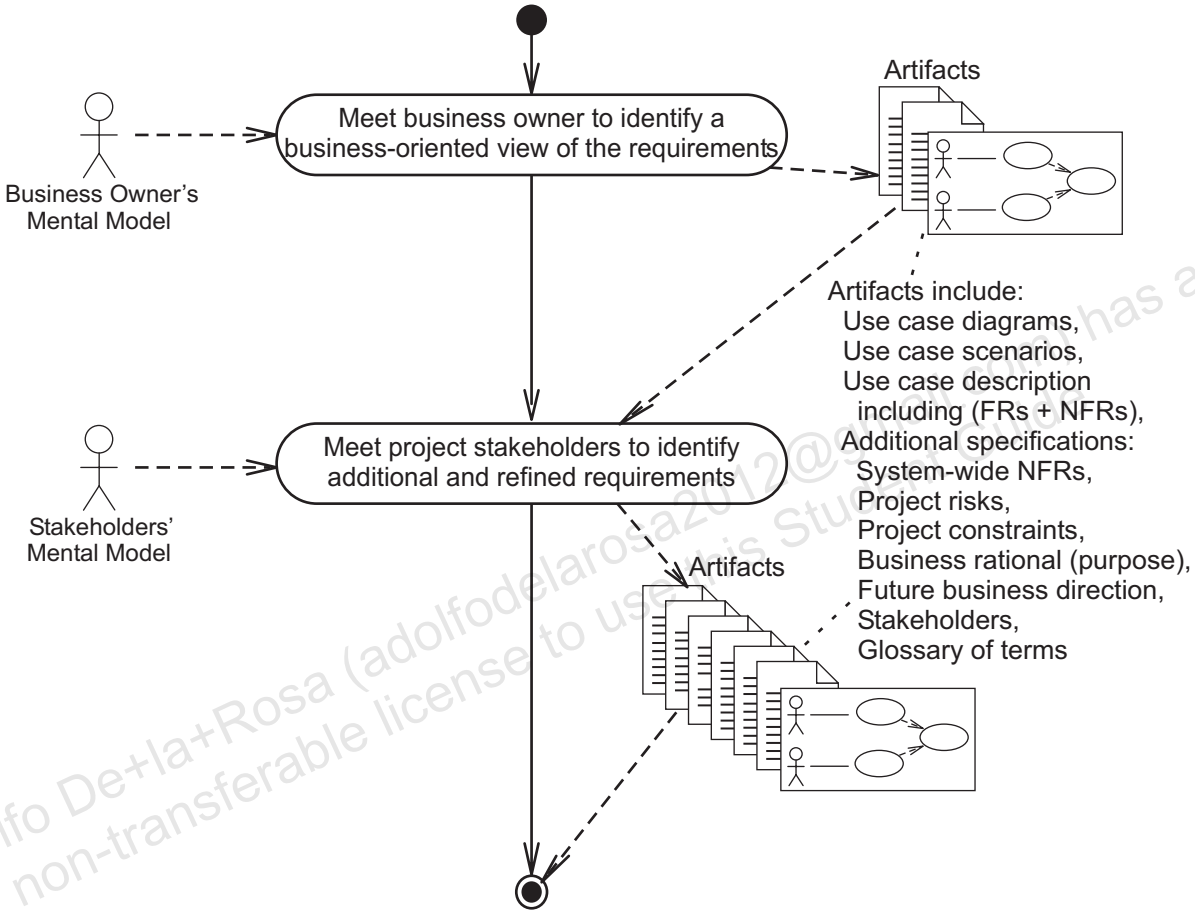


Figure 2-3 Activities and Artifacts of the Requirements Gathering Workflow

The Requirements Gathering activities include:

- Meeting with the business owner to gather the business-significant use cases and some business-specific low-level details

  To determine the size and scope of a software project, the architect and business analyst meet with the business owner to determine the high-level FRs of the system in the form of use cases. The output of this activity are artifacts that describe a business-oriented vision of the system. These artifacts include:

  - Use case diagrams
  - Use case scenarios

- Use case description (which include FRs and NFRs)

Note – The detailed flow of events in the use case description may be partially completed at this stage. However, it will be completed in the Analysis workflow.

  - Additional specification document(s), which include:
    - Additional system-wide NFRs
    - Project risks
    - Project constraints
    - Business rational (purpose)
    - Future directions of the business
    - Client-side stakeholders
    - Glossary of terms
- Meet with the other project stakeholders to gather additional information and elaborate and verify previous information. The artifacts are the same as above but each stakeholder input will add or refine the previous artifacts.

Note – There are many variations to this workflow and the created artifacts. Some of these variations will be discussed later in the course. However, two significant variations are briefly mentioned below.

Significant variations to the Requirements Gathering Workflow

Two significant variations to the Requirements Gathering workflow are as follows:

- In an iterative and incremental development process, you will add information and refinements in each iteration.
- In a non-incremental process (for example, Waterfall), you will often create a Vision document (or similar document) containing the requirements gathered from the business owner, and a System Requirement Specification document (or similar document) containing all the requirements gathered from all stakeholders.

Exploring the Requirements Analysis Workflow

There are two views of the Requirements model created in this workflow:

- The completed *Use Case Forms*, containing full details of the actor's interaction with the system and what the system does in response.
- The *Domain model* (a Class diagram of the *key abstractions* of the problem space).

These artifacts and the previously created artifacts together make up the *problem space*. The language used in these artifacts should be understandable to the client-side stakeholders because they use this language to discuss their business (also known as a *domain*). For example, in a Hotel Reservation System the problem space uses terms like customer, rooms, hotel properties, payment methods, charge items, and so on. These terms are often the key abstractions of the problem space and they are modeled with a Domain model.

Workflow Purpose and Job Roles

The purpose of Requirements Analysis workflow is to analyze, refine, and model the requirements of the system. Table 2-2 shows the description of this workflow.

Table 2-2 Requirements Analysis Workflow

Workflow	Purpose	Description
Requirements Gathering	Determine *what* the system must do	
Requirements Analysis	Model the existing business processes	Determine: • The detailed behavior of each use case • Supplementary use cases • The key abstractions that exist in the current increment of the problem domain • A business domain class diagram

The Requirements Analysis workflow activities are usually performed by business analyst and the architect job roles.

Workflow Activities and Artifacts

Figure 2-4 illustrates the activities the Requirements Analysis workflow might include.

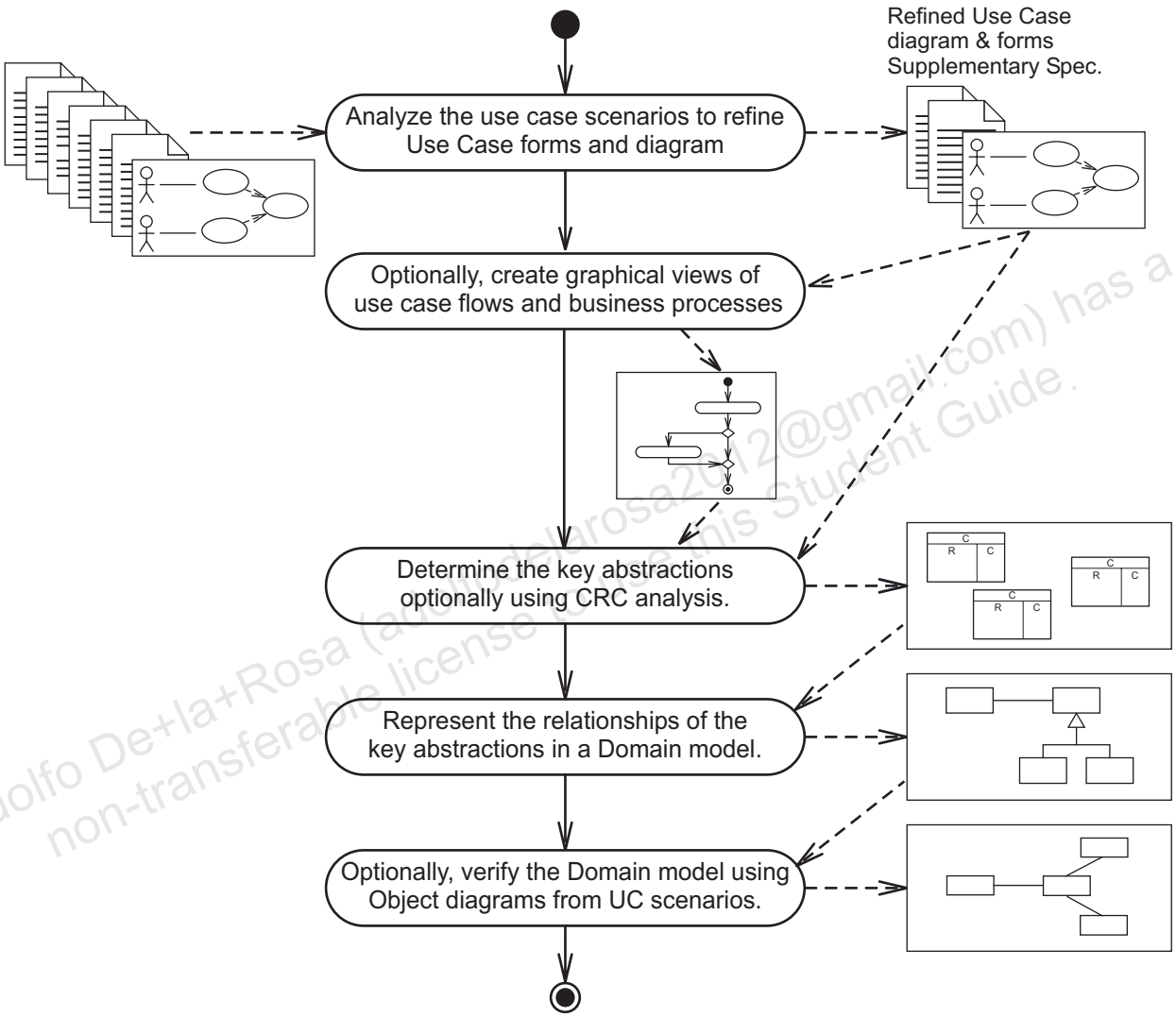


Figure 2-4 Activities and Artifacts of the Requirements Analysis Workflow

The Requirements Analysis activities are:

- Analyze the use case scenarios to discover more detail.

  Use scenarios to refine the Use Case forms. Add or refine the main flow and alternative flows to this form. Also, analyzing use case scenarios often identifies common patterns in use cases. These

common patterns can be made explicit in the Use Case form. For example, a common pattern of many use cases might be the need to log on to the system.

- Refine the Use Case diagram from the analysis.

  For example, the log-on behavior can be captured in a separate use case node and have other use cases refer to the log-on use case.

- Optionally, create an Activity diagram that provides a visual view of the flow of events to augment and verify the textual Use Case forms.

  A graphical view of the flow of events can be used to augment and verify the stakeholders understanding of the problem. Use a UML Activity diagram to model the activities of a use case at a fine level of granularity. These Activity diagrams can be shown to UML-savvy stakeholders to verify the analyst's understanding of the necessary behavior of the system. In their simplest form, Activity diagrams have some similarity to flow charts. Therefore, any stakeholders who understand flow charts should understand these diagrams.

  Activity diagrams have many other uses, but at this point the only other use worth discussing is to use them in modeling the business process, which may include multiple use cases and activities that are conducted external to the system. There are several different non-UML formal Business Processing Modeling Notations (BPMN), which may be more suited for this purpose.

- Determine the key abstractions. This might be done by using CRC analysis, however there are other techniques to determine the key abstractions.

  CRC analysis is one technique for identifying the key abstractions of the problem domain. CRC stands for Class Responsibility Collaboration.

- Represent the relationships of the key abstractions in a Domain model.

  After the key abstractions have been identified, a UML Class diagram is created or appended to represent the key abstractions discovered and their interrelationships.

- Optionally, verify the Domain model using Object diagrams from use case scenarios.

  The entities and data in a use case scenario can be modeled in Object diagrams. Verify the associations and multiplicities on the Domain model by checking the object links in the Object diagrams.

Exploring the Architecture Workflow

The Architecture workflow is rather complex and is beyond the scope of this course. However, you will see how the Architecture model (developed in this workflow) affects the Design workflow. From the perspective of the designer, the Architecture model provides a template of the high-level system structure into which the designer plugs in designed components.

Workflow Purpose and Job Roles

The purpose of the Architecture workflow is to identify risk in the project and to mitigate the risk by modeling the high-level structure of the system. Table 2-3 shows the description of this workflow.

Table 2-3 Architecture Workflow

Workflow	Purpose	Description
Requirements Gathering	Determine *what* the system must do	
Requirements Analysis	Model the existing business processes	
Architecture	Model the high-level system structure to satisfy the NFRs	• Develop the highest-level structure of the software solution • Identify the technologies that will support the Architecture model • Elaborate the Architecture model with Architectural patterns to satisfy NFRs

The Architecture workflow activities are performed by the architect.

Workflow Activities and Artifacts

Figure 2-5 illustrates the Architecture workflow usually includes six activities.

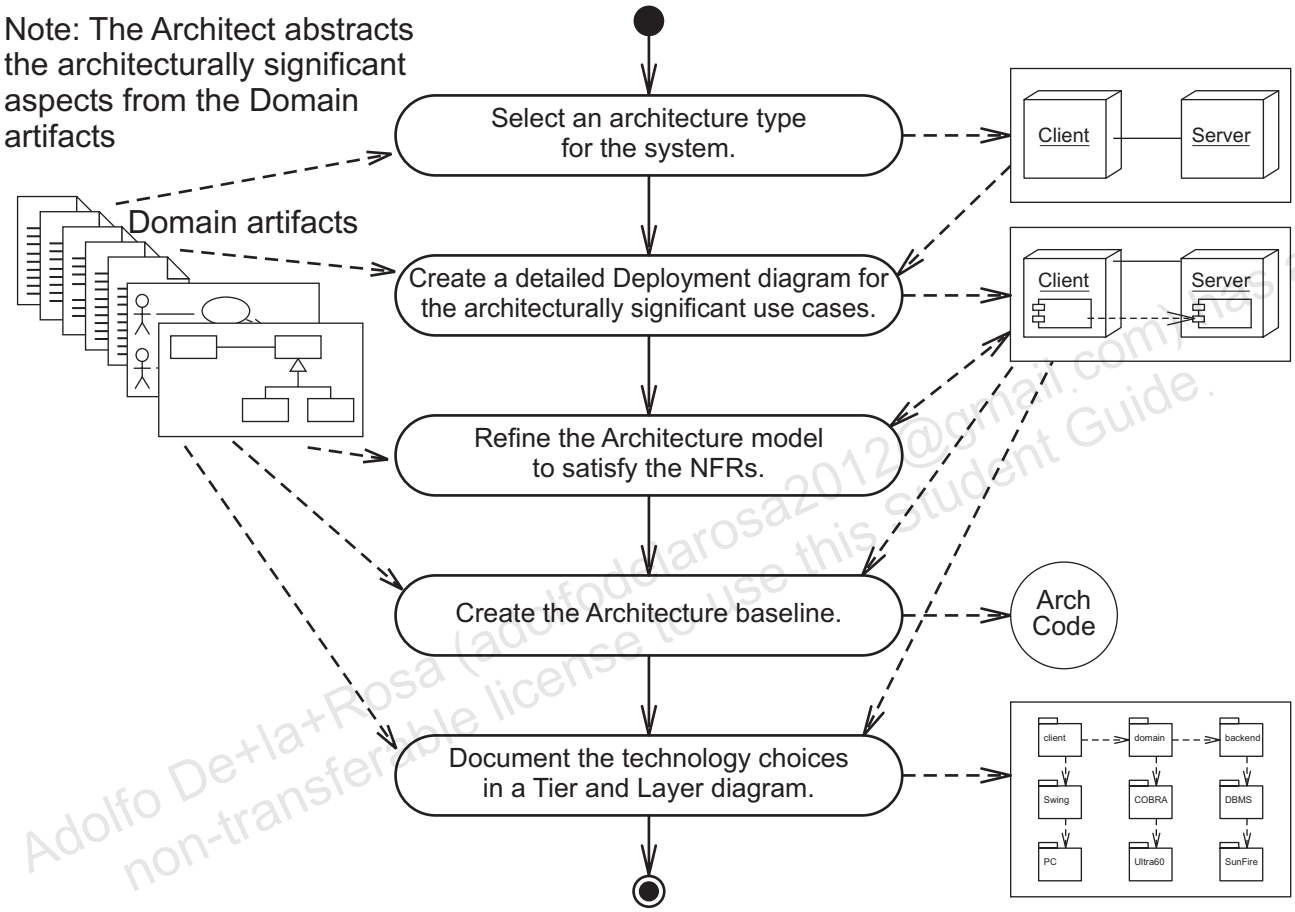


Figure 2-5 Activities and Artifacts of the Architecture Workflow

The Architecture activities are:

- Select an architecture type for the system.

  An architect selects the architecture type that best satisfies the high-level constraints and NFRs. An architecture type refers to a small set of abstract architectures, such as standalone, client/server, App-centric n-tier, Web-centric n-tier, and Enterprise n-tier.

  Based on the selected architecture type, a high-level Deployment diagram is created to show the distribution of the top-level system components on each hardware node.

- Create a detailed Deployment diagram for the architecturally significant use cases.

The architect creates a detailed Deployment diagram that shows the main components necessary to support the architecturally significant use cases. This diagram is shows the low-level components and their dependencies with the high-level Deployment diagram.

- Refine the Architecture model to satisfy the NFRs.

  The architect uses Architectural patterns to transform the high-level architecture type into a robust hardware topology that supports the NFRs. This discussion is beyond the scope of this course.

- Create and test the Architecture baseline.

  The architect implements the architecturally significant use cases in an *evolutionary prototype*. When all architecturally significant use cases have been developed, the evolutionary prototype is called the *Architecture baseline*. The Architecture baseline represents the version of the system solution that manages all risks. The Architecture baseline is the final product of the Elaboration phase and becomes the starting point of the Construction phase.

  The Architecture baseline is tested to verify that the selected systemic qualities have been satisfied. The Architecture baseline is refined by applying additional patterns to satisfy the systemic qualities.

- Document the technology choices in a Tiers and Layers diagram.

  For each tier, and each layer, the architect identifies the appropriate technologies to be used.

Exploring the Design Workflow

The ultimate goal of the Design workflow is to develop a Solution model that the development team can use to construct the code for the proposed system.

Workflow Purpose and Job Roles

The purpose of the Design workflow is to create a Solution model of the system that satisfies the behavior of the system as defined by the requirements. Table 2-4 shows the description of this workflow.

Table 2-4 Design Workflow

Workflow	Purpose	Description
Requirements Gathering	Determine what the system must do	
Requirements Analysis	Model the existing business processes	
Architecture	Model the high-level system structure to satisfy the NFRs	
Design	Model *how* the system will support the use cases	• Create a Design model for a use case using Interaction diagrams • Identify and model objects with non-trivial states using a State Machine diagram. • Apply design patterns to the Design model • Create a Solution model by merging the Design and Architecture models • Refine the Domain model

The Design model for the use cases is based on the classes discovered during analysis, which are the business objects. By using Interaction diagrams and State Machine diagrams, you can explore the detailed object interactions and state transitions. Doing this will enable you to discover new support classes as well as missing attributes and methods for the domain classes.

The Design model is then merged with the Architecture model. This combined model is called the Solution model, in this course. The Solution model also includes the refined Domain model.

The Solution model is also refined by a variety of Design patterns that are applicable to the design problem and context. The components in the Solution model can be implemented in code.

The Design workflow activities are performed by the software designer job role.

Workflow Activities and Artifacts

Figure 2-6 illustrates the Design workflow usually includes six activities.

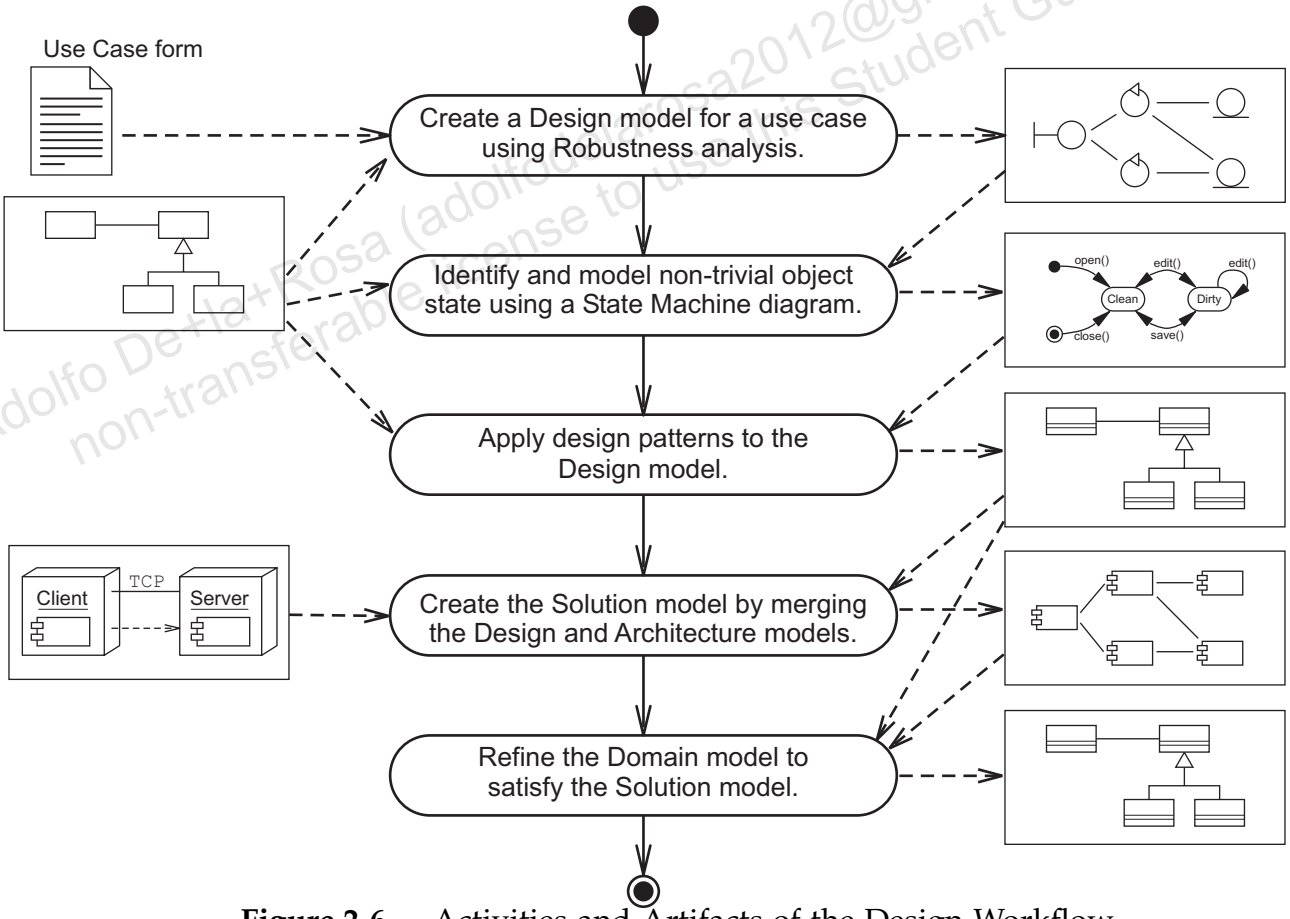


Figure 2-6 Activities and Artifacts of the Design Workflow

The Design activities are briefly described here:

- Create a Design model for a use case using Interaction (or Robustness analysis) diagrams.

  Using Interaction diagrams, you add new classes to enable to Domain objects to work with the computer system. By following the flows in a use case (and its scenarios), you can discover a collection of collaborating software components that satisfy the functional requirements of the use case. These usually include Service classes and Boundary classes. During this process you might discover missing methods and attributes in your Domain classes.

- Identify and model objects with non-trivial states using a State Machine diagram.

  A small, but significant number of classes have non-trivial state dependencies, where their behavior to any event depends on their current state. For these classes, you can model the states, trigger events, conditions, and actions that can occur. State Machine diagrams are often used in combination with Interaction diagrams to check that every scenario that could occur has been considered. For example, an automatic gearbox in a car should react differently when you select reverse, depending on whether you are stationary or traveling forward at a speed of 70 mph.

- Apply design patterns to the Design model.

  There are now hundreds of design patterns that have been documented. This activity enables the designer to refine the Design model with applicable patterns to make the software more flexible and robust. These patterns can be added at any time during design.

- Create the Solution model by merging the Design and Architecture models.

  In this activity, the designer inserts the components from the Design model into the Architectural model. This structure provides about 80 percent of the components that must be coded during the Implementation workflow.

- Refine the Domain model to satisfy the Solution model.

  The Domain model from the Analysis workflow must be refined to provide enough details (attributes and methods) for the Implementation workflow.

Exploring the Implementation, Testing, and Deployment Workflows

These three workflows are outside the scope of this course. However, they will be covered in the appendix so that you can see how these other workflows complete the software development process.

Workflow Purpose and Job Roles

The purpose of the Implementation workflow is to build the software components defined in the Solution model. The Implementation workflow is performed by the software programmer job role. The purpose of the Testing workflow is to test the implementation against the expectations defined in the requirements. The Testing workflow is performed by the software tester job role. The purpose of the Deployment workflow is to deploy the implementation into the production environment. The Deployment workflow is performed by the deployment specialist job role. Table 2-5 shows the description of this workflow.

Table 2-5 Implementation, Testing, and Deployment Workflows

Workflow	Purpose	Description
Requirements Gathering	Determine what the system must do	
Requirements Analysis	Model the existing business processes	
Architecture	Model the high-level system structure to satisfy the NFRs	
Design	Model how the system will support the use cases	
Implementation, Testing, and Deployment	Implement, test, and deploy the system	• Implement the software • Perform testing • Deploy the software to the production environment

Workflow Activities and Artifacts

Figure 2-7 illustrates the following activities that the Implementation, Testing, and Deployment workflows include.

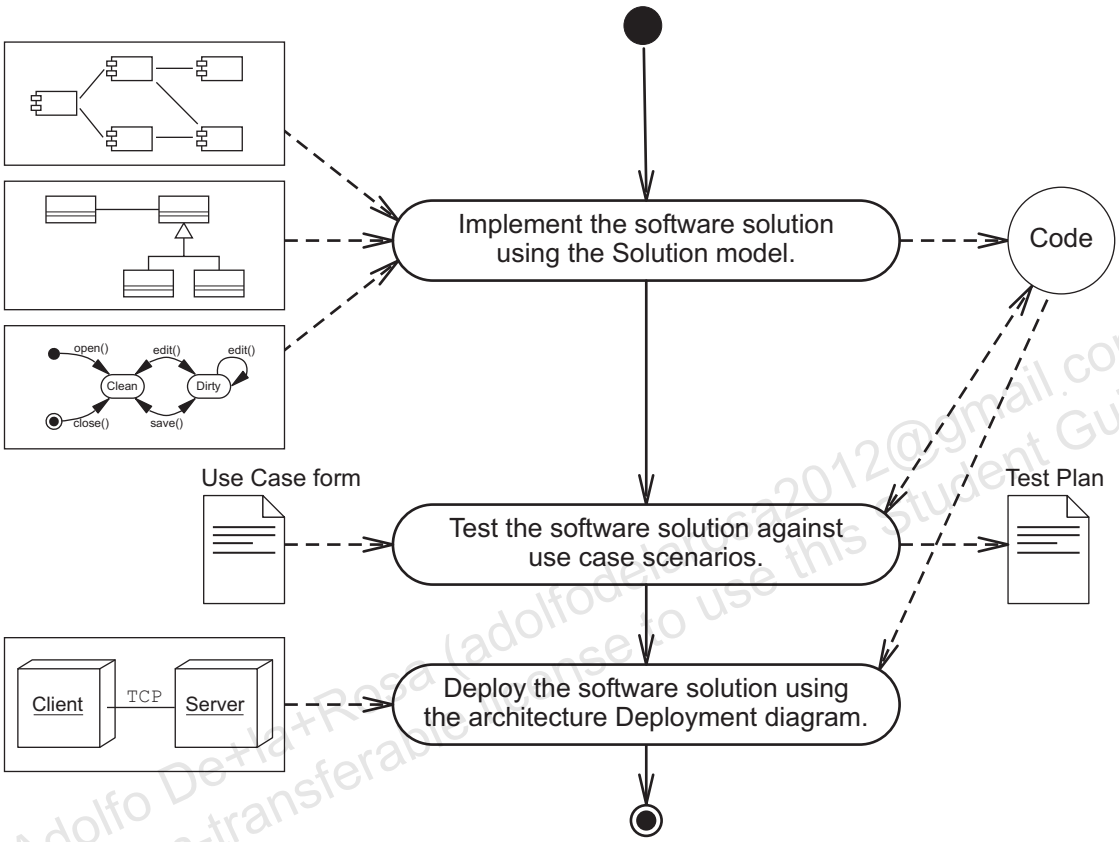


Figure 2-7 The Activities and Artifacts of the Implementation, Testing, and Deployment Workflows

The Implementation activity is: Implement the software solution using the Solution model. This activity maps the class structure defined in the refined Domain model into a physical, Java technology class structure.

The Testing activities are numerous and varied. This course focuses on acceptance testing. This activity is: Test the software solution against the use case scenarios. In this activity, the tester generates the Functional test plan from the use case scenarios. The tests are performed to verify that the functional behavior of the system matches the use case requirements.

The Deployment activity is: Deploy the software solution using the architecture Deployment diagram. In this activity, the deployment specialist uses the Deployment diagram to set up the computer, network, and component structure of the production environment.

Examining the Benefits of Modeling Software

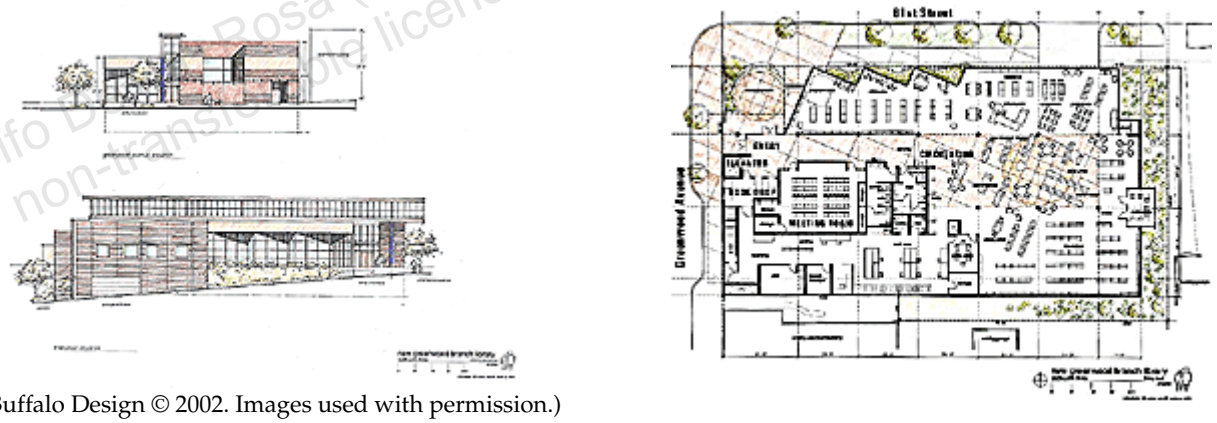
The inception of every software project starts as an idea in someone's mind. To construct a realization of that idea, the development team must create a series of conceptual models that transform the idea into a production system.

What Is a Model?

"A model is a simplification of reality." (Booch UML User Guide page 6)

"A description of static and/or dynamic characteristics of a subject area, portrayed through a number of views (usually diagrammatic or textual)." (Larman, page 617)

You can think of a model in terms of traditional architecture. Before constructing a building, an architect draws several different views of the building itself and the location of the building. Figure 2-8 shows two diagrams from the same architectural plans for a library in Seattle. These different diagrams show different views.



(Buffalo Design © 2002. Images used with permission.)

Figure 2-8 Example Models From Traditional Architecture

From architecture you can learn that:

- A model is an abstract conceptualization of some entity (such as a building) or a system (such as software).

  A model is abstract by its nature. Models can represent many different things, both physical such as buildings and computer networks, and conceptual such as software.

- Different views show the model from different perspectives.

  An architect draws many diagrams showing different views or perspectives of the building. Views include: floor plans, plumbing, electrical wiring, external construction, street plans, and so on. A software model can also be represented by different views, and you can create diagrams for all of these views.

Why Model Software?

> "We build models so that we can better understand the system we are developing." (Booch UML User Guide page 6)

Specifically, modeling enables you to:

- Visualize new or existing systems

  Models help us understand, conceptualize, and visualize any kind of system. You can model systems that already exist. For example, the policies and procedures of a business form an existing business process. A Use Case diagram is used to capture the business "system."

  You can model systems that have not been built yet, such as the Solution model of the software.

- Communicate decisions to the project stakeholders

  Some models can be easily understood by the business owner, domain experts, and other client-side stakeholders. Other models are more relevant to communication among development team members.

- Document the decisions made in each OOSD workflow

  Depending on the size of the development team, not all developers are involved in each workflow. For example, sometimes only the Project Manager and the System Architect will gather requirements. This information (in the form on a Requirements model) must be communicated to the design team.

- Specify the structure (static) and behavior (dynamic) elements of a system

  During the Design and Implementation workflows, the structural and dynamic aspects of the software system must be specified from the Requirements model. The Solution model represents the

complete conceptualization of the software system. The UML provides various diagrams that support creating these types of views of the Solution model.

- Use a template for constructing the software solution

  The Solution model is used as a template for implementing the code modules that make up the software system.

OOSD as Model Transformations

Figure 2-9 shows a few of the models that a development team might create. The project starts as the mental models of the client-side stakeholders. During the Requirements Gathering and Analysis workflows the mental models are combined and transformed into the *Requirements model*. The non-functional requirements of the Requirements model are transformed in the *Architecture model*, which defines the high-level structure of the software solution. The functional requirements of the Requirements model are transformed into a *Design model*, which defines the abstract components of the software solution. The Design model is merged with the Architecture model to produce the *Solution model*, which defines the detailed structure of the software solution. The Solution model is used to guide the construction of the code for the software solution.

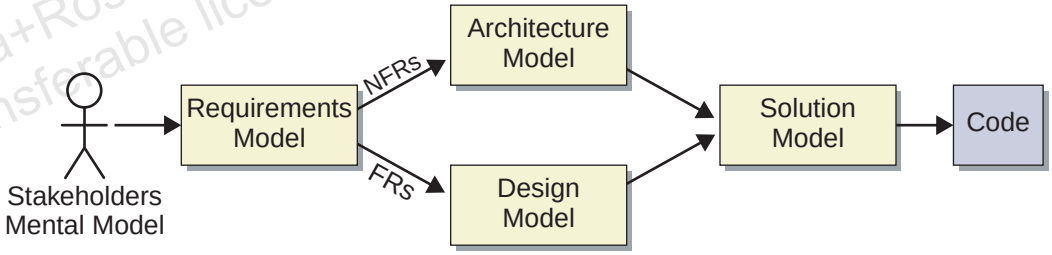


Figure 2-9 Software Development as a Series of Model Transformations

Some of these models are important to capture and record in an artifact. Some artifacts are documents. Some artifacts are diagrams of visual models of the system. Finally, some artifacts are the program modules that make up the implementation.

In this section, you will be briefly introduced to the Unified Modeling Language which this course uses to create diagram artifacts.

Note – Not all methodologies put a high value on creating artifacts of models. eXtreme Programming (XP) is such a methodology. In XP, the requirements are captured by user *stories*, and the *design* is captured by the current state of the code. However, mental models at various stages of abstraction will always exist in the minds of developers. With the techniques taught in this course you should be able to create and represent your mental models in the UML.

Defining the UML

> "The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system." (UML v1.4 page xix)

Using the UML, a model is composed of:

- Elements (things and relationships)
- Diagrams (built from elements)
- Views (diagrams showing different perspectives of a model)

UML Elements

UML diagrams are built from modeling primitives or elements. There are two broad categories of elements: things (also called nodes) and relationships (also called links). Figure 2-10 shows many of the fundamental UML elements.

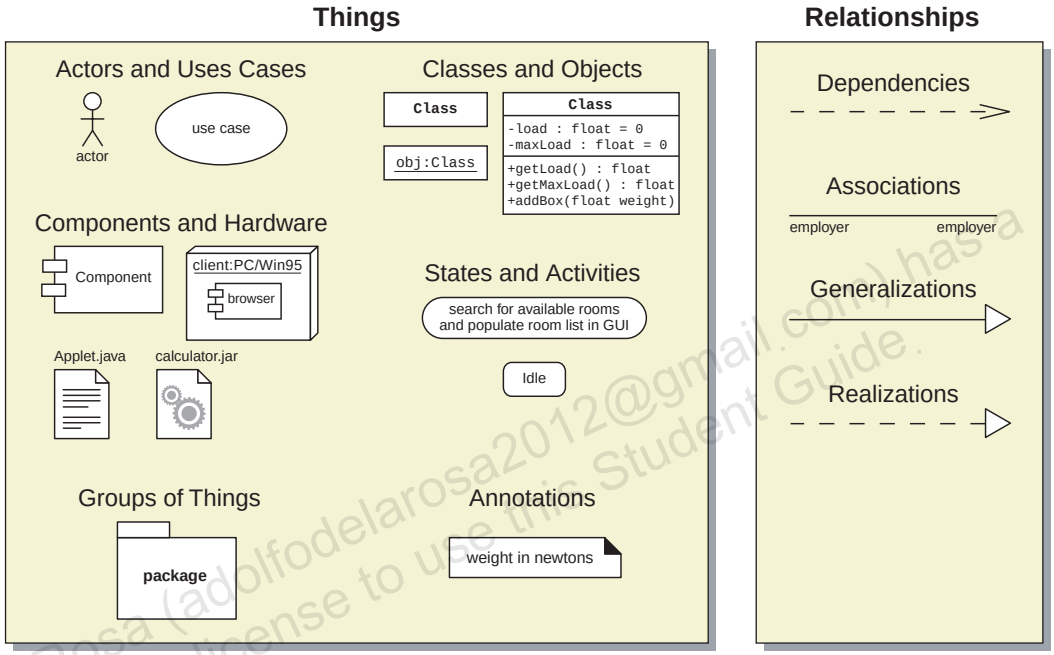


Figure 2-10 Elements of UML Diagrams

Throughout the course, you will be introduced to each UML diagram and all of the elements that make up that diagram.

UML Diagrams

UML diagrams enables you to create visualizations of your mental models of a software system. These diagrams are used to construct many of the artifacts in the workflows described in this course. Figure 2-11 shows there are currently 14 fundamental types of diagrams in UML 2.2.

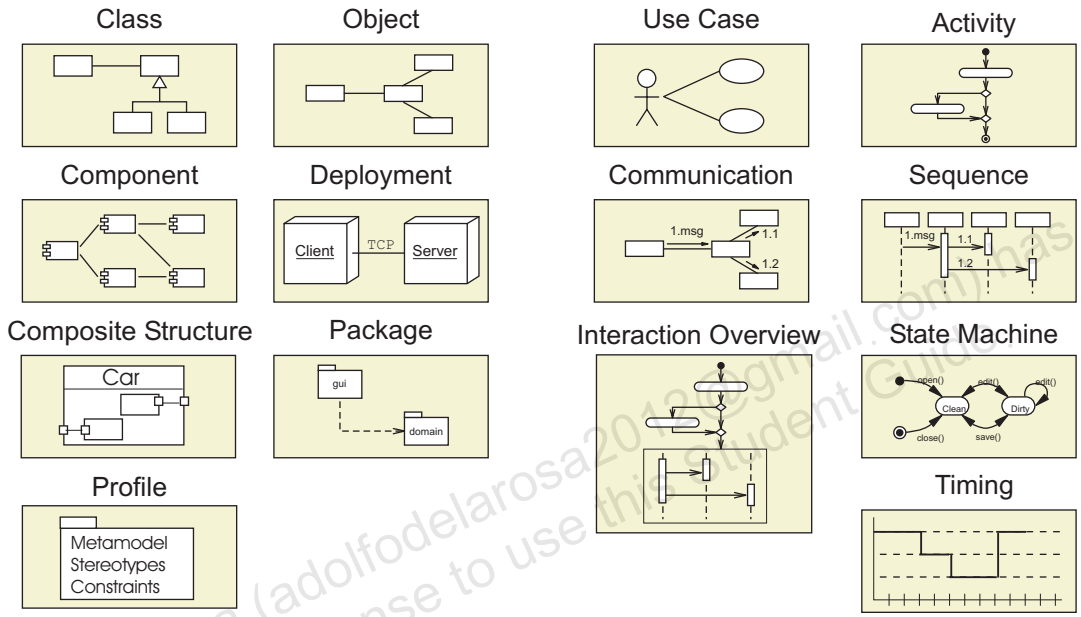


Figure 2-11 Set of UML Diagrams

Following is a brief description of each diagram:

- A *Use Case diagram* represents the set of high-level behaviors that the system must perform for a given actor.
- A *Class diagram* represents a collection of software classes and their interrelationships.
- An *Object diagram* represents a runtime snapshot of software objects and their interrelationships.
- A *Communication diagram (formerly Collaboration diagram)* represents a collection of objects that work together to support some system behavior.
- A *Sequence diagram* represents a time-oriented perspective of an object communication.
- An *Activity diagram* represents a flow of activities that might be performed by either a system or an actor.

- A *State Machine diagram* represents the set of states that an object might experience and the triggers that transition the object from one state to another.
- A *Component diagram* represents a collection of physical software components and their interrelationships.
- A *Deployment diagram* represents a collection of components and shows how these are distributed across one or more hardware nodes.
- A *Package diagram* represents a collection of other modeling elements and diagrams.
- An *Interaction Overview diagram* represents a form of activity diagram where nodes can represent interaction diagram fragments. These fragments are usually sequence diagram fragments, but can also be communication, timing, or interaction overview diagram fragments.
- A *Timing diagram* represents changes in state (state lifeline view) or value (value lifeline view). It can also show time and duration constraints and interactions between timed events.
- A *Composite Structure diagram* represents the internal structure of a classifier, usually in form of parts, and can include the interaction ports and interfaces (provided or required).
- A *Profile diagram* might define additional diagram types or extend existing diagrams with additional notations.

UML Diagram Categories

The UML diagrams can be categorized into two main categories: structural (show the static structure of the objects in a system) and behavioral (show the dynamic behavior of objects in a system). Figure 2-12 groups the UML diagrams into these two main categories. It further shows that four of the behavioral diagrams can be subcategorized as interaction diagrams.

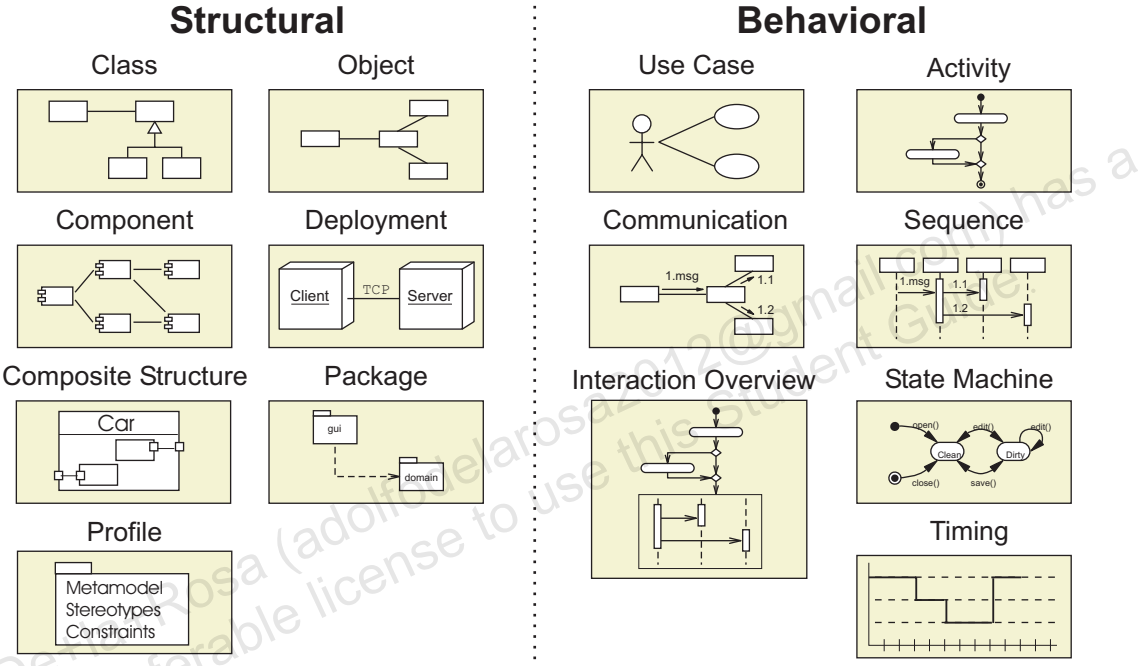


Figure 2-12 Views Created From UML Diagrams

Note – A few of these UML 2.2 diagrams will not be formally covered in this course.

Common UML Elements and Connectors

UML has a a few elements and connectors that are common across UML diagrams. These elements and connectors include:

- Package
  - A package is used to group together any UML elements and diagrams.
  - Packages are elements, therefore they can be nested.

- A package is a logical view. Therefore, an element might exist in more than one package to form different views.
- Java technology packages are a subset of UML packages.

A package in UML is used to group together elements (which includes packages) or diagrams to form a cohesive logical view. These packages might be named to provide a namespace for these groups. Packages might be nested to provide a hierarchical logical or conceptual view. Most packages do not relate to any physical grouping. Because they are logical, an element can exist within more than one package to form different views of that element. For example, a use case might be viewed related to other use cases, and it might also be shown packaged with the classes it interacts with.

Figure 2-13 shows an example of using packages for views. The Door Access Use Cases View shows the use cases in the system. The Lock Door View shows the Lock Door use case, along with a subset of classes that are used by that use case and one or more interaction diagrams that show the scenarios of the use case.

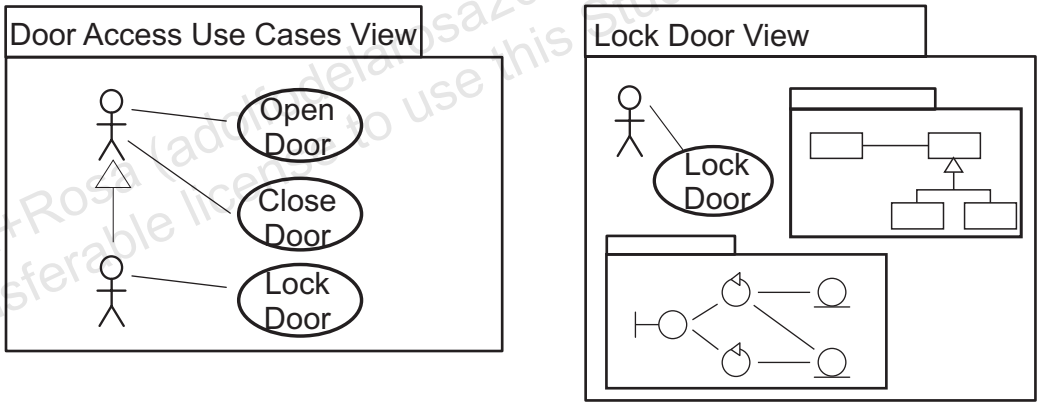


Figure 2-13 Example of Using Packages

- Note

  A note allows textual notes to be added to any aspect of a diagram. This allows additional information in the form of text to be attached (with a dashed line) to any UML element. For example, notes can annotate classes, methods, components, actors, associations, and so on. A note in UML is in the form of a dog-eared rectangle with its upper-right corner bent over. Figure 2-14 on page 2-32 shows using

UML notes to show the internal behavior of the details of the methods. In UML, colon equals (:=) is an assignment; however, you could have written Java technology syntax instead.

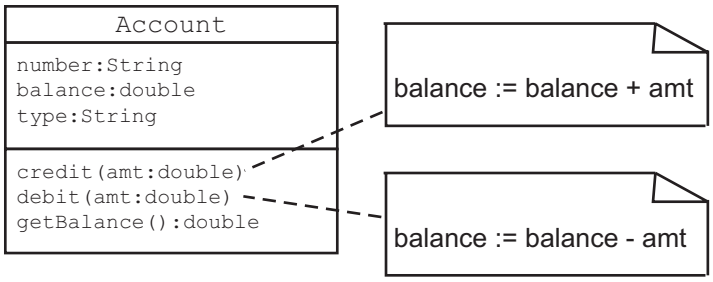


Figure 2-14 Example of Using a UML Note

- Dependency
  - The dependency notation shows that one UML element depends on another UML element.
  - The type of dependency can be attached to the line with a stereotype.

Caution – This notation of a dashed line with an open arrowhead has an alternative meaning in Activity diagrams.

- Stereotypes

  Stereotypes are used to specify a more specific type of element or connector type. The notation for stereotypes is to bracket them in a Guillemont quotation marks symbols « » used in French and some other languages. However, for the typographically challenged, double angle brackets are a suitable substitute.

Figure 2-15 shows an example of an `Account` class in its analysis form with minimal details, and then in its design (refined) form with design details added. The figure also shows an example of a Web Browser depending on a Web Server and the dependency stereotype showing the protocol as http or https.

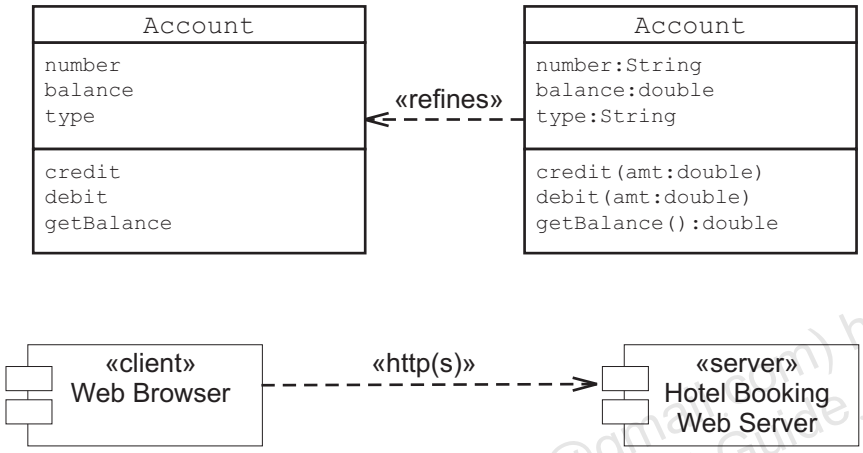


Figure 2-15 Example of UML Dependencies and Stereotypes

Table 2-6 shows some common UML stereotypes that are used in the course. For example an «`interface`» stereotype can be used on a UML class to specify that it only has OO interface (method signature definitions), but no implementation.

Table 2-7 shows some common stereotypes that were use in this course that are not part of the UML.

UML examples make extensive use of stereotypes. Table 2-6 lists some of standard UML stereotypes used in this course.

Table 2-6 Standard UML Stereotypes

Stereotype	Definition
«`actor`»	*An actor defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates.* (UML v1.4 page 3-97)
«`create`»	*Create is a stereotyped usage dependency denoting that the client classifier creates instances of the supplier classifier.* (UML v1.4 page 2-52)

Table 2-6 Standard UML Stereotypes (Continued)

Stereotype	Definition
«extend»	*An extend relationship from use case A to use case B indicates that an instance of use case B may be augmented (subject to specific conditions specified in the extension) by the behavior specified by A.* (UML v1.4 page 3-98)
«import»	*Import is a stereotyped permission dependency between two namespaces, denoting that the public contents of the target package are added to the namespace of the source package.* (UML v1.4 page 2-48)
«include»	*An include relationship from use case E to use case F indicates that an instance of the use case E will also contain the behavior as specified by F.* (UML v1.4 page 3-98)
«interface»	*An interface is a specifier for the externally-visible operations of a class, component, or other classifier (including subsystems) without specification of internal structure.* (UML v1.4 page 3-50)
«refine»	*Specifies refinement relationship between model elements at different semantic levels, such as analysis and design.* (UML v1.4 page 2-18)

Table 2-7 lists some of the stereotypes that were created for this course.

Table 2-7 Non-Standard UML Stereotypes

Stereotype	Definition
«accessors»	This stereotype delineates the set of methods that retrieve the attributes of a class.
«operations»	This stereotype delineates the set of operations for an Analysis-level class.
«constructors»	This stereotype delineates the set of Java technology constructors of a class.
«mutators»	This stereotype delineates the set of methods that change or set the attributes of a class.
«UI Frame»	A class that denotes a graphical user interface (GUI) frame or window.

Table 2-7 Non-Standard UML Stereotypes (Continued)

Stereotype	Definition
«tcp/ip»	This stereotype denotes the use of Transport Control Protocol/Internet Protocol (tcp/ip) to communicate between hardware nodes.
«http(s)»	This stereotype denotes the use of Hypertext Transfer Protocol (http) or Secure Hypertext Transfer Protocol (https) to communicate between components.
«methods»	This stereotype delineates the set of business logic methods of a class.
«Controller»	This stereotype denotes a component that acts as a user interface (UI) controller.
«View»	This stereotype denotes a component that acts as a visual UI component.
«Service»	This stereotype denotes a component that manage interactions between collections of objects. This is basically the same as the «control» stereotype defined in the "UML Profile for Software Development Processes."
«Entity»	This stereotype denotes a component that acts as a persistent Domain entity. This is basically the same as the «entity» stereotype defined in the "UML Profile for Software Development Processes."
«JDBC»	This stereotype denotes the use of the Java Database Connectivity™ technology protocol to communicate between a component and a Relational Database Management System (RDBMS).
«Java package»	A package that holds a set of Java technology class components.

What UML Is and Is Not

There are many misconceptions about UML. Table 2-8 lists some of the more significant misconceptions.

Table 2-8 Misconceptions About the UML

UML is not:	**But it:**
Used to create an executable model. Models are not executable. Only working code is executable.	Can be used to generate code skeletons. UML tools can generate the software skeletons from a model. There are attempts to produce executable UML in the future.
A programming language. It is a visual, modeling language.	Maps to most OO languages. UML has features that enable a software designer to represent programming language constructs.
A development process (methodology). UML is simply a set of diagrams that can be used during the development process to model and document. It does not define a process that tells you when and how to build these diagrams.	Can be used as a tool within the activities of a development process (methodology). UML is a fundamental tool to several popular OO development processes, including the Unified Process. UML can even be used if you do not use a formal development process.

UML Tools

UML itself is a tool. You can create UML diagrams on paper, on a white boards, on flip-chart-sized plastic sheets (which are statically charged, will adhere to most surfaces, and can be written on using whiteboard markers). For example, you can create models by drawing on napkins over lunch; have team meetings to collaborate on a Class diagram on a conference room whiteboard; or paper the walls and windows of a room with numerous plastic sheets, draw on these sheets, and then move these sheets to another room. This enables a great deal of flexibility in the creation of models and team collaboration. But when a model must be captured as a long-lived artifact, then the diagram should be drawn in a tool that will enable printing and archiving.

UML tools can be divided into roughly two categories: drawing and modeling tools. Tools such as StarOffice, Illustrator, and Visio can draw UML diagrams, but using such tools puts the burden on the developer to verify that the diagram is drawn correctly.

UML modeling tools:

- Provide computer-aided drawing of UML diagrams

  For example, UML modeling tools may prohibit you from placing an actor node in an Object diagram. This verification is accomplished by restricting the drawing operations of the UML tool by the syntactic constraints of the UML specification. This restriction may be relaxed because UML 2.2 does not strictly enforce the boundaries between diagrams. Therefore, it should be possible to include a state machine inside an internal structure.

- Support (or enforce) semantic verification of diagrams

  For example, UML modeling tools will maintain a semantic connection between a class called *Xyz* in one Class diagram and the same *Xyz* class in a Component diagram.

  This referential integrity is accomplished by maintaining a single, consistent model underlying the diagrams drawn in the modeling tool.

- Provide support for a specific methodology

  For example, Rational Software Modeler (RSM) and Rational Software Architect (RSA) include UML tools that supports the Rational Unified Process (RUP). This is a benefit if your team has chosen the methodology used by the tool that the team purchased. Otherwise, the tool might be a liability.

Note – RSA and RSM supersede the well-know product Rational Rose.

- Generate code skeletons from the UML diagrams

  Most UML tools can generate code skeletons for a variety of OO languages, such as C++ and the Java programming language. This process of generating code from models is called *forward engineering*. Some tools can also generate UML diagrams from existing source code. This is called *reverse engineering*. Some tools can keep the diagrams and source code synchronized.

- Organize all of the diagrams for a project

Keeping track of the multitude of diagrams for a large software project can be a major task with ordinary drawing tools. UML modeling tools provide built-in support for maintaining all UML diagrams for a project. Some tools also provide built-in support for version control.

- Automatic generation of modeling elements for design patterns, Java™ Platform, Enterprise Edition (Java™ EE platform) components, and so on

  Some UML tools provide advanced mechanisms that fill in templates of design patterns and other frameworks, such as Java EE platform components.

Summary

In this module, you were introduced to OOSD at a very high level. Here are a few important concepts:

- The OOSD process starts with gathering the system requirements and ends with deploying a working system.

  A working, tested, and deployed (production) system is the ultimate goal of the OOSD process. Everything that you do should be to support the creation of a working system.

- Workflows (disciplines) define the activities that transform the artifacts of the project from the use cases to the implementation code (the final artifact).

  The workflows and their activities lead you from the Requirements model to the production system.

- The UML supports the creation of visual artifacts that represent views of your models.

  Use UML as your primary tool to create visual representations of the models you build throughout the OOSD process.