CSC 2210
Object Oriented Analysis & Design

Dr. Akinul Islam Jony
Associate Professor
Department of Computer Science, FST
American International University - Bangladesh (AIUB)

akinul@aiub.edu

Introducing the UML

>> What is UML?
>> Building Blocks of the UML
>> Structural Things
>> Behavioral things
>> Grouping things
>> Annotational things
>> Relationships in the UML
>> Diagrams in the UML
>> Architecture
>> Software Development Life Cycle

What is UML?

>> **Unified Modeling Language (UML)** is a standardized general-purpose modeling language in the field of software engineering.

>> **UML** is a standard language for writing **software blueprints**.

>> The standard is managed, and was created
- by OMG (Object Management Group)
- based on work from Booch, Rumbaugh, Jacobson

>> UML Version 2.5.1 (December 2017)
(for details about UML version: https://www.omg.org/spec/UML/)

What is UML?

>> The **UML** is a **language** for

- Visualizing
- Specifying
- Constructing
- Documenting

the artifacts of a software-intensive system.

The UML Is a Language

>> A **language** provides a **vocabulary** and the **rules** for combining words in that vocabulary for the purpose of **communication**.

>> A **modeling language** is a language whose **vocabulary** and **rules** focus on the **conceptual and physical representation** of a **system**.

>> A **modeling language** such as the **UML** is thus a **standard language** for **software blueprints.**

>> The **vocabulary** and **rules** of a language such as the **UML** tell you how to create and read well-formed models, but they don't tell you what models you should create and when you should create them. That's the role of the software development process.

The UML is a language for Visualizing

>> Typically, projects and organizations develop their own language, and it is difficult to understand what's going on if you are an outsider or new to the group (**first problem**).

>> There are some things about a software system you can't understand unless you build models (**second problem**).

>> Sometimes, developer who cut the code never wrote down the models that are in his or her head, that information would be lost forever (**third problem**).

>> Hence, **writing/creating models in the UML addresses these issues.**

>> Besides, some things are best modeled textually; others are best modeled graphically. The **UML is such a graphical language**.

>> The **UML is more than just a bunch of graphical symbols**. Each symbol in the UML notation is a **well-defined semantics**. In this manner, one developer can write a model in the UML, and another developer, or even another tool, can interpret that model unambiguously.

The UML is a language for Specifying

>> Specifying means building models that are **precise**, **unambiguous**, and **complete**.

>> In particular, the **UML addresses** the **specification** of all the important **analysis**, **design**, and implementation **decisions** that must be made in developing and deploying a software-intensive system.

The UML is a language for Constructing

>> **UML** is not a visual programming language, but its models can be **directly connected** to a variety of **programming languages**

>> It is possible to **map** from a model in the UML to a programming language such as Java, C++, or VB, or even to tables in a RDBMS

>> This mapping permits **forward engineering**—the generation of code from a UML model into a programming language.

>> You can also **reconstruct** a model from an implementation back into the UML (**reverse engineering**)

The UML is a language for Documenting

>> The UML addresses the **documentation** of a system's **architecture** and all of its **details.**

>> The UML also provides a language for expressing requirements and for tests.

>> Finally, the UML provides a language for modeling the activities of project planning and release management.

Primary Goals in the Design of UML

>> Provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.

>> Provide extensibility and specialization mechanisms to extend the core concepts.

>> Be independent of particular programming languages and development processes.

>> Provide a formal basis for understanding the modeling language.

>> Encourage the growth of the OO tools market.

>> Support higher-level development concepts such as collaborations, frameworks, patterns, and components.

>> Integrate best practices.

Where Can the UML Be Used?

>> The UML is intended primarily for software-intensive systems. It has been used effectively for domains such as

- Enterprise information systems
-Banking and financial services
-Telecommunications
- Transportation
- Defense/aerospace
- Retail
- Medical electronics
- Scientific
- Distributed Web-based services

>> The UML is not limited to modeling software. In fact, it is expressive enough to model non-software systems, such as workflow in the legal system, the structure and behavior of a patient healthcare system, software engineering in aircraft combat systems, and the design of hardware.

A Conceptual Model of the UML

>> To understand the UML, you need to form a conceptual model of the language, and this requires learning **three** major **elements**:

- the UML's basic **building blocks**,
- the **rules** that dictate how those building blocks may be put together,
- some common **mechanisms** that apply throughout the UML.

>> Once you have grasped these ideas, you will be able to read UML models and create some basic ones.

>> As you gain more experience in applying the UML, you can build on this conceptual model, using more advanced features of the language.

Building Blocks of the UML

>> The vocabulary of the **UML** encompasses **three** kinds of building blocks:

- Things
- Relationships
- Diagrams

>> **Things** are the abstractions that are first-class citizens in a model; **relationships** tie these things together; **diagrams** group interesting collections of things.

Things in the UML

>> There are **four** kinds of **things** in the UML:

- Structural things
  - nouns/static of UML models (irrespective of time).
- Behavioral things
  - verbs/dynamic parts of UML models.
- Grouping things
  - organizational parts of UML models.
- Annotational things
  - explanatory parts of UML models.

>> These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models.

Structural Things

>> **Structural** things are the **nouns** of UML models.

>> These are the mostly static parts of a model, representing elements that are either conceptual or physical.

>> Collectively, the structural things are called ***classifiers***.

Structural Things: **Class**

>> **A class** is a description of set of objects that share the same attributes, operations, relationships, and semantics.

>> Graphically, a **class** is rendered as a rectangle, usually including its name, attributes, and operations, as in Figure 2-1

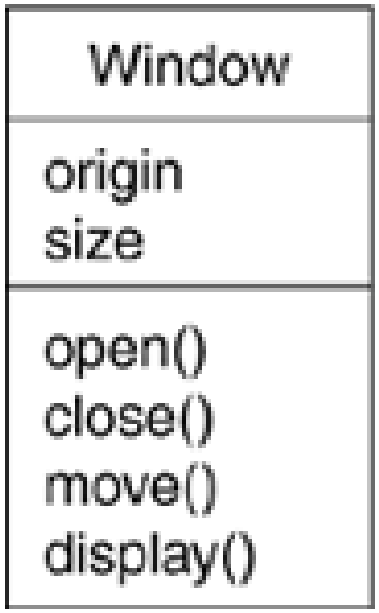


Figure 2-1. Classes

Structural Things: **Interface**

>> An ***interface*** is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element.

>> An interface **defines** a set of **operation specifications** (that is, their **signatures**) but never a set of operation implementations.

>> The declaration of an interface looks like a class with the keyword **«interface»** above the name; attributes are not relevant, except sometimes to show constants.

Structural Things: **Interface**

>> An interface rarely stands alone, however. An interface **provided by a class** to the outside world is shown as a small **circle** attached to the class box by a line.

>> An interface **required by a class from some other class** is shown as a small **semicircle** attached to the class box by a line, as in Figure 2-2.

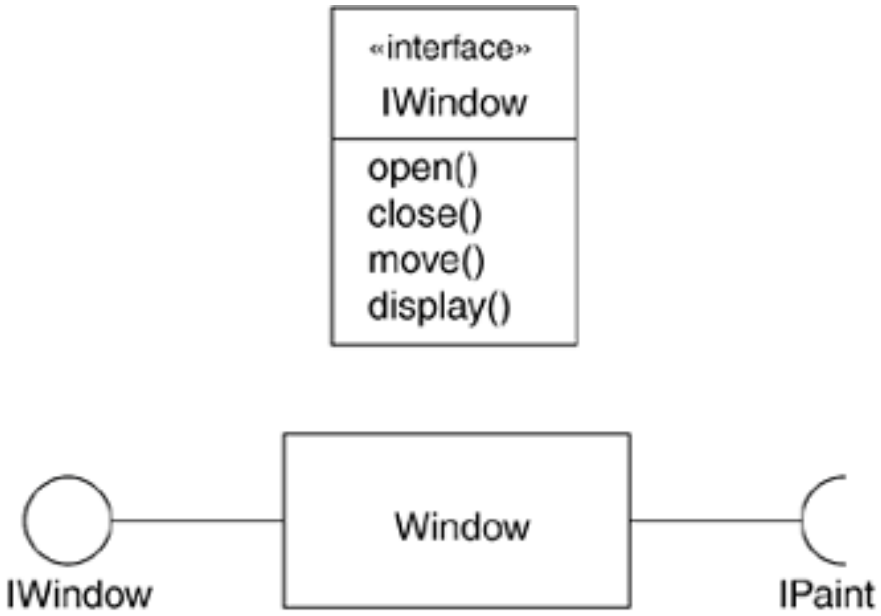


Figure 2-2. Interfaces

Structural Things: **Collaboration**

>> A **collaboration** defines an **interaction** and is a **society of roles** and **other elements** that work together to provide some **cooperative behavior** that's bigger than the sum of all the elements.

>> Collaborations have structural, as well as behavioral, dimensions.

>> A given class or object might participate in several collaborations. These collaborations therefore represent the implementation of patterns that make up a system.

>> Graphically, a collaboration is rendered as an *ellipse* with dashed lines, sometimes including only its name, as in Figure 2-3.

Figure 2-3. Collaboration

Structural Things: **Use Case**

>> A **use case** is a description of sequences of actions that a system performs that yield observable results of value to a particular **actor**.

>> A use case is used to structure the behavioral things in a model.

>> A use case is realized by a collaboration.

>> Graphically, a use case is rendered as an *ellipse* with solid lines, usually including only its name, as in Figure 2-4.

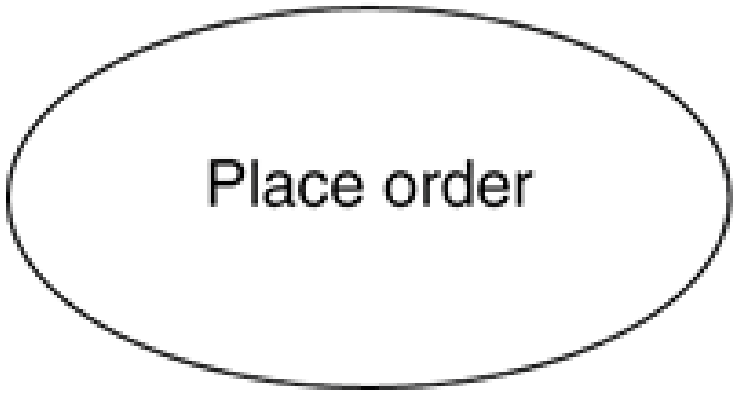


Figure 2-4. Use Cases

Structural Things: **Active Class**

>> An **active class** is a class whose objects own one or more processes or threads and therefore can initiate control activity.

>> An active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements.

>> Graphically, an active class is rendered as a class with double lines on the left and right; it usually includes its name, attributes, and operations, as in Figure 2-5.

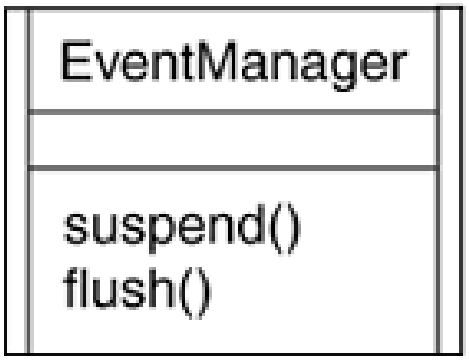


Figure 2-5. Active Classes

Structural Things: **Component**

>> A **component** is a **modular part** of the system design that hides its implementation behind a set of external interfaces.

>> **A component** is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.

>> Graphically, a component is rendered like **a class with a special icon in the upper right corner**, as in Figure 2-6.

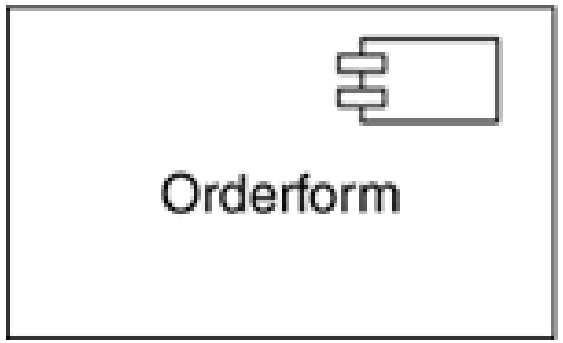


Figure 2-6. Components

Structural Things: **Artifact**

>> An **artifact** is a physical and replaceable part of a system that contains physical information ("bits"), such as source code files, executables, and scripts.

>> An artifact typically represents the physical packaging of source or run-time information.

>> Graphically, an artifact is rendered as a rectangle with the keyword «artifact» above the name, as in Figure 2-7.

Figure 2-7. Artifacts

Structural Things: **Node**

>> A **node** is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability.

>> A set of components may reside on a node and may also migrate from node to node.

>> Graphically, a node is rendered as a **cube**, usually including only its name, as in Figure 2-8.

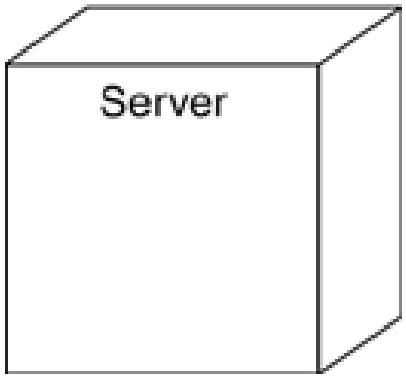


Figure 2-8. Nodes

Behavioral Things

>> **Behavioral** things are the dynamic parts of UML models.

>> These are the **verbs** of a model, representing behavior over time and space.

>> In all, there are **three** primary kinds of behavioral things:

- interactions,
- state machines, and
- activities

Behavioral Things: **Interaction**

>> An **interaction** is a behavior that comprises a set of messages exchanged among a set of objects or roles within a particular context to accomplish a specific purpose.

>> The behavior of a society of objects or of an individual operation may be specified with an interaction.

>> An interaction involves a number of other elements, including messages, actions, and connectors (the connection between objects).

>> Graphically, a message is rendered as a **directed line**, almost always including the name of its operation, as in Figure 2-9.

Figure 2-9. Messages

Behavioral Things: **State Machine**

>> A **state machine** is a behavior that specifies the **sequences of states** an object or an interaction goes through during its lifetime in response to events, together with its responses to those events.

>> The behavior of an individual class or a collaboration of classes may be specified with a state machine.

>> Graphically, a state is rendered as a **rounded rectangle**, usually including its name and its substates, if any, as in Figure 2-10.

Figure 2-10. States

Behavioral Things: **State Machine**

>> A state machine involves a number of other elements, including **states**, **transitions** (the flow from state to state), **events** (things that trigger a transition), and **activities** (the response to a transition).

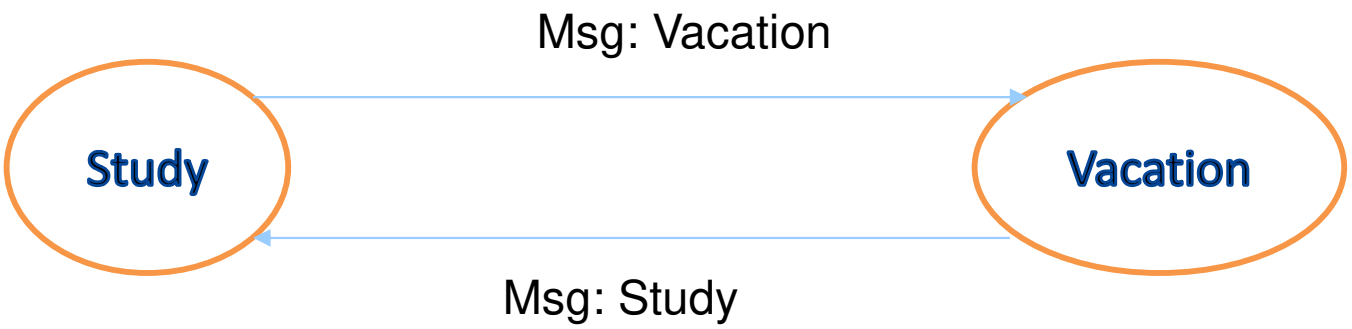


Figure. Example of States

Behavioral Things: **Activity**

>>An **activity** is a behavior that specifies the **sequence of steps** a computational process performs.

>> In an interaction, the focus is on the set of objects that interact. In a state machine, the focus is on the life cycle of one object at a time. In an activity, the focus is on the flows among steps without regard to which object performs each step.

>> A step of an activity is called an *action*.

>> Graphically, an action is rendered as a **rounded rectangle** with a name indicating its purpose, as in Figure 2-11.

>> **States** and **actions** are **distinguished** by their different **contexts**.

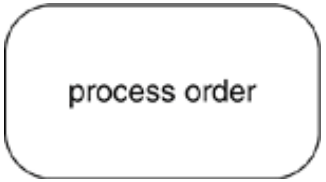


Figure 2-11. Actions

Grouping Things

>> **Grouping** things are the **organizational parts** of UML models.

>> These are the boxes into which a model can be decomposed.

>> There is one primary kind of grouping thing, namely, **packages**.

>> There are also variations, such as **frameworks**, **models**, and **subsystems** (kinds of packages).

Grouping Things: **Package**

>> **A package** is a general-purpose mechanism for **organizing** elements into groups.

>> Structural things, behavioral things, and even other grouping things can be placed in a package.

>> Unlike components (which exist at run time), a package is purely conceptual (meaning that it exists only at development time).

>> Graphically, a package is rendered as a **tabbed folder**, usually including only its name and, sometimes, its contents, as in Figure 2-12.

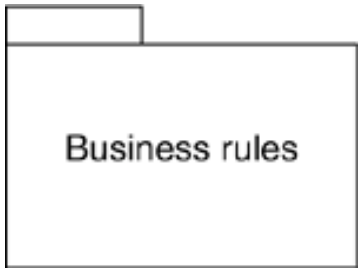


Figure 2-12. Packages

Annotational Things

>> **Annotational** things are the **explanatory parts** of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model.

>> There is one primary kind of annotational thing, called a **note**.

>> A **note** is simply a symbol for rendering constraints and comments attached to an element or a collection of elements.

>> Graphically, a note is rendered as a **rectangle with a dog-eared corner**, together with a textual or graphical comment, as in Figure 2-13.

Figure 2-13. Notes

Relationships in the UML

>> There are **four** kinds of **relationships** in the UML:

- Dependency
- Association
- Generalization
- Realization

>> These relationships are the basic relational building blocks of the UML.

Relationships in the UML: **Dependency**

>> A **dependency** is a **semantic relationship** between two things in which a change to one (the **independent** thing) may affect the semantics of the other thing (the **dependent** thing).

>> Graphically, a dependency is rendered as a **dashed line**, possibly directed, and occasionally including a label, as in Figure 2-14.

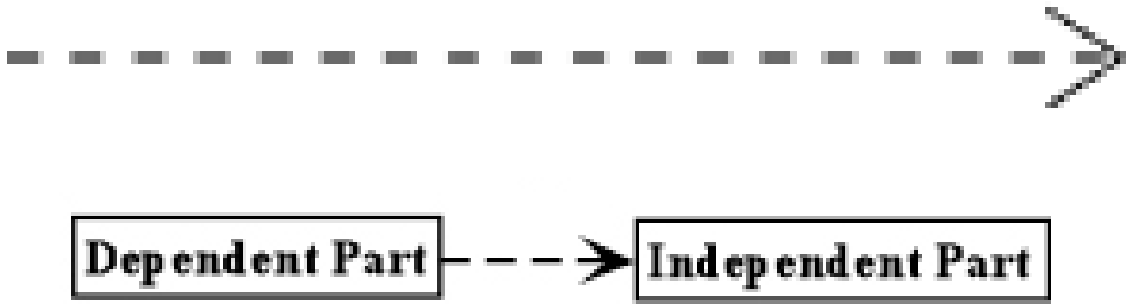


Figure 2-14. Dependencies

Relationships in the UML: **Association**

>> An **association** is a **structural relationship** among classes that describes a set of links, a link being a connection among objects that are instances of the classes.

>>**Aggregation** is a special kind of association, representing a structural relationship between a whole and its parts.

>> Graphically, an association is rendered as a **solid line**, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names, as in Figure 2-15.

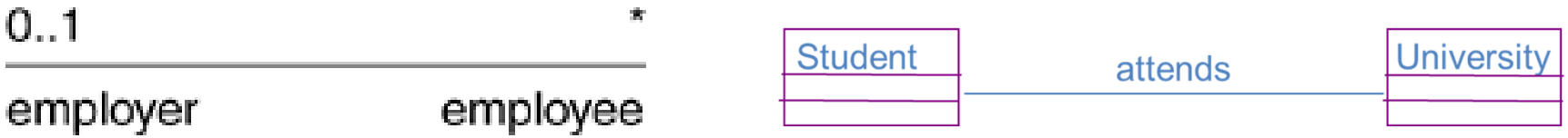


Figure 2-15. Associations

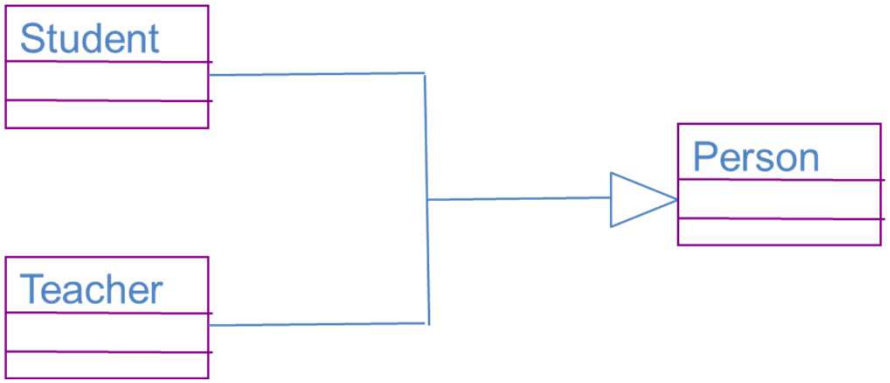
Relationships in the UML: **Generalization**

>> A **generalization** is a **specialization/generalization relationship** in which the **specialized** element (the **child**) builds on the specification of the **generalized** element (the **parent**).

>> The child shares the structure and the behavior of the parent.

>> Graphically, a generalization relationship is rendered as a **solid line with a hollow arrowhead** pointing to the parent, as in Figure 2-16.

Figure 2-16. Generalizations



Relationships in the UML: **Realization**

>> A ***realization*** is a **semantic relationship** between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out.

>> You'll encounter realization relationships in two places: between interfaces and the classes or components that realize them, and between use cases and the collaborations that realize them.

>> Graphically, a realization relationship is rendered as a **cross** between a generalization and a dependency relationship, as in Figure 2-17.

Figure 2-17. Realizations

Diagrams in the UML

>> A **diagram** is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).

>> The UML includes nine such diagrams.

Structural Diagrams

Represent the static aspects of a system.

- Class Diagram
- Object Diagram
- Component Diagram
- Deployment Diagram

Behavioral Diagrams

Represent the dynamic aspects of a system.

- Use case Diagram
- Sequence Diagram (Interaction)
- Collaboration Diagram
- Statechart Diagram
- Activity Diagram

Diagrams in the UML

>> Generally, the UML includes thirteen kinds of diagrams:

1. Class diagram
2. Object diagram
3. Component diagram
4. Composite structure diagram (Hybrid)
5. Use case diagram
6. Sequence diagram
7. Communication diagram
8. State diagram
9. Activity diagram
10. Deployment diagram
11. Package diagram
12. Timing diagram
13. Interaction overview diagram (Hybrid)

Diagrams in the UML: **Class Diagram**

>> A **Class** diagram shows a set of classes, interfaces, and collaborations and their relationships.

>> Class diagrams address the static design view of a system.

>> Class diagram that includes active classes address the static process view of a system.

>> Component diagrams are variants of class diagrams.

Diagrams in the UML: **Object Diagram**

>> An **object** diagram shows a set of objects and their relationships.

>> Object diagrams represent static snapshots on instances of the things found in class diagrams.

>> These designs address the static design or process view of a system from the perspective of real or prototypical cases.

Diagrams in the UML: **Component Diagram**

>> A **component** diagram shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors.

>> Component diagrams address the static design implementation view of a system.

>> They are important for building large systems from smaller parts.

Diagrams in the UML: **Use Case Diagram**

>> A **Use case** diagram shows a set of use cases and actors and their relationships.

>> Use case diagrams address the static use case view of a system.

>> These diagrams are especially important in organizing and modeling the behaviors of a system.

Diagrams in the UML: **Interaction Diagram**

>> Both **sequence** diagrams and **communication/collaboration** diagrams are kinds of **interaction** diagrams.

>> An **interaction** diagram shows an interaction, consisting of a set of objects or roles, including the messages that may be dispatched among them.

>> Interaction diagrams address the dynamic view of a system.

>> A **sequence** diagram is an interaction diagram that emphasizes the time-ordering of messages.

>> A **communication/collaboration** diagram is an interaction diagram that emphasizes the structural organization of the objects or roles that send and receive messages.

Diagrams in the UML: **Interaction Diagram**

>> **Sequence** diagrams and **communication** diagrams represent similar basic concepts, but each diagram emphasizes a different view of the concepts.

>> **Sequence** diagrams emphasize temporal ordering, and **communication** diagrams emphasize the data structure through which messages flow.

>> A **timing** diagram shows the actual times at which messages are exchanged.

Diagrams in the UML: **State/Statechart Diagram**

>> A **state** diagram shows a state machine, consisting of states, transitions, events, and activities.

>> A state diagrams shows the dynamic view of an object.

>> They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

Diagrams in the UML: **Activity Diagram**

>> An **activity** diagram shows the structure of a process or other computation as the flow of control and data from step to step within the computation.

>> Activity diagrams address the dynamic view of a system.

>> They are especially important in modeling the function of a system and emphasize the flow of control among objects.

Diagrams in the UML: **Deployment Diagram**

>> A **deployment** diagram shows the configuration of run-time processing nodes and the components that live on them.

>> Deployment diagrams address the static deployment view of an architecture.

>> A node typically hosts one or more artifacts. They are related to component diagrams in that a node typically encloses one or more components.

Diagrams in the UML: **Artifact Diagram**

>> An **artifact** diagram shows the physical constituents of a system on the computer.

>> Artifacts include files, databases, and similar physical collections of bits.

>> Artifacts are often used in conjunction with deployment diagrams (UML treats artifact diagrams as a variety of deployment diagram).

>> Artifact diagrams model the static implementation view of a system.

>> Artifact diagrams are essentially class diagrams that focus on a system's artifacts. Artifacts show the classes and components that they implement.

Diagrams in the UML: **Package Diagram**

>> A **package** diagram shows the decomposition of the model itself into organization units and their dependencies.

Diagrams in the UML: **Timing Diagram**

>> A **timing** diagram is an interaction diagram that shows actual times across different objects or roles, as opposed to just relative sequences of messages.

Diagrams in the UML: **Interaction Overview Diagram**

>> An **interaction overview** diagram is a hybrid of an **activity** diagram and a **sequence** diagram.

Architecture

>> Visualizing, specifying, constructing, and documenting a software-intensive system demands that the system be viewed from a number of perspectives.

>> **Different stakeholders** - end users, analysts, developers, system integrators, testers, technical writers, and project managers- each bring different agendas to a project, and each looks at that system in different ways at different times over the project's life.

>> A **system's architecture** is perhaps the most important artifact that can be used to **manage** these **different viewpoints** and thus control the iterative and incremental development of a system throughout its life cycle.

Architecture

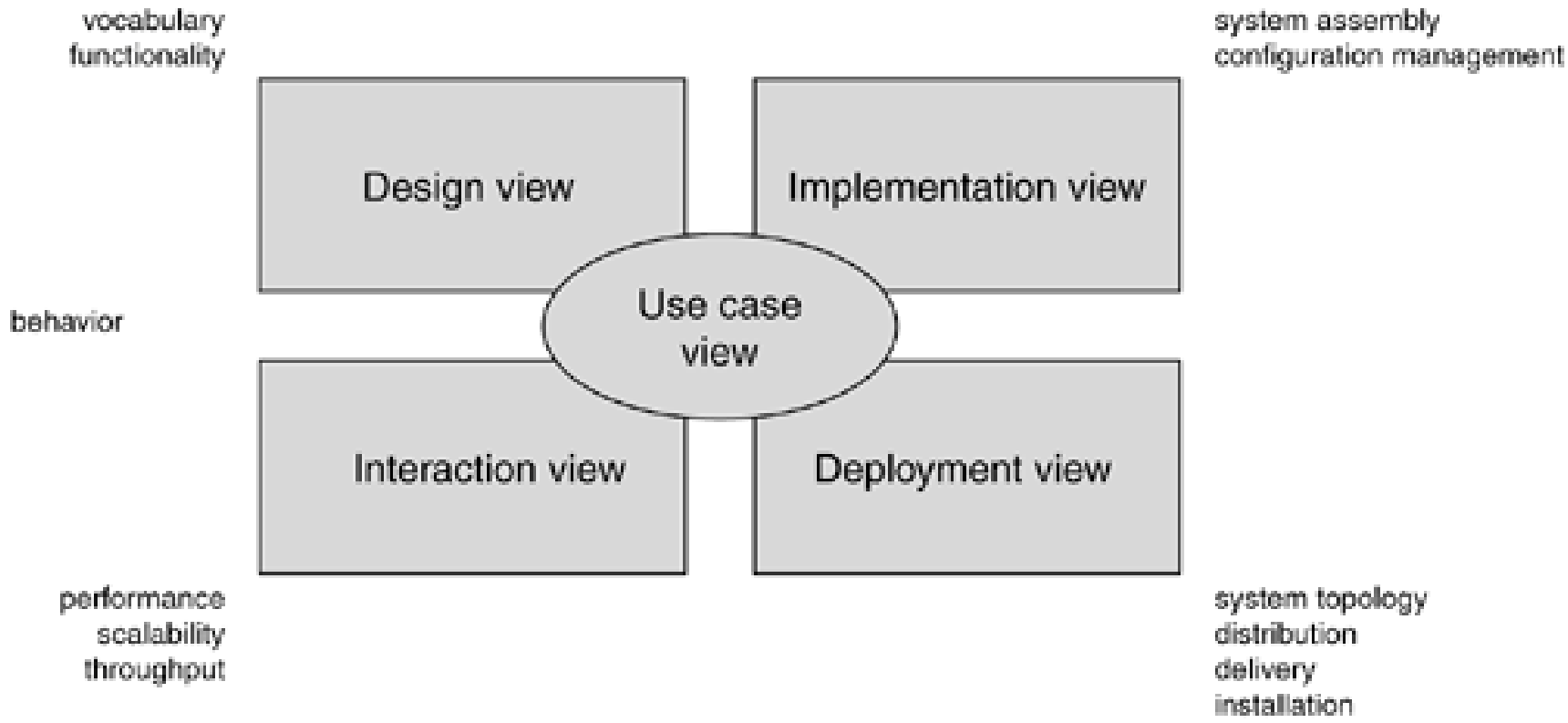


Figure 2-23. Modeling a System's Architecture

Architecture: **Use Case View**

>> The **use case view** of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers.

>> This view doesn't really specify the organization of a software system. Rather, it exists to specify the forces that shape the system's architecture.

>> With the UML, the **static aspects** of this view are captured in use case diagrams; the **dynamic aspects** of this view are captured in interaction diagrams, state diagrams, and activity diagrams.

Architecture: **Design View**

>> The **design view** of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution.

>> This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users.

>> With the UML, the **static aspects** of this view are captured in class diagrams and object diagrams; the **dynamic aspects** of this view are captured in interaction diagrams, state diagrams, and activity diagrams.

Architecture: **Process/Interaction View**

>> The **interaction view** of a system shows the flow of control among its various parts, including possible concurrency and synchronization mechanisms.

>> That means, the **Process/Interaction view** of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms.

>> This view primarily addresses the performance, scalability, and throughput of the system.

>> With the UML, the **static** and **dynamic aspects** of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that control the system and the messages that flow between them.

Architecture: **Implementation View**

>> The **implementation view** of a system encompasses the artifacts that are used to assemble and release the physical system.

>> This view primarily addresses the configuration management of the system's releases, made up of somewhat independent files that can be assembled in various ways to produce a running system.

>> It is also concerned with the mapping from logical classes and components to physical artifacts.

>> With the UML, the **static aspects** of this view are captured in artifact diagrams; the **dynamic aspects** of this view are captured in interaction diagrams, state diagrams, and activity diagrams.

Architecture: **Deployment View**

>> The **deployment view** of a system encompasses the nodes that form the system's hardware topology on which the system executes.

>> This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system.

>> With the UML, the **static aspects** of this view are captured in deployment diagrams; the **dynamic aspects** of this view are captured in interaction diagrams, state diagrams, and activity diagrams.

Software Development Life Cycle

>> The UML is largely **process-independent**, meaning that it is not tied to any particular **software development life cycle**.

>> However, to get the most benefit from the UML, you should consider a process.

>> This process can be broken into **phases**. A phase is the span of time between two major milestones of the process, when a well-defined set of objectives are met, artifacts are completed, and decisions are made whether to move into the next phase.

>> An **example** of **software development life cycle** is shown in **Figure 2-24** with four phases: **inception**, **elaboration**, **construction**, and **transition**. In the diagram, workflows are plotted against these phases, showing their varying degrees of focus over time.

- This kind of software development approach is called **Rational Unified Process** (**RUP**).

Software Development Life Cycle: **RUP**

>> The **Rational Unified Proces (RUP)** is an agile software development method, in which the life cycle of a project, or the development of software, is divided into four phases (**inception**, **elaboration**, **construction**, and **transition**). Various activities take place during these phases.

>> RUP is *iterative*, because all of the process's core activities repeat throughout the project.

>> The process is **agile** because various components can be adjusted, and phases of the cycle can be repeated until the software meets requirements and objectives.

Software Development Life Cycle: **RUP**

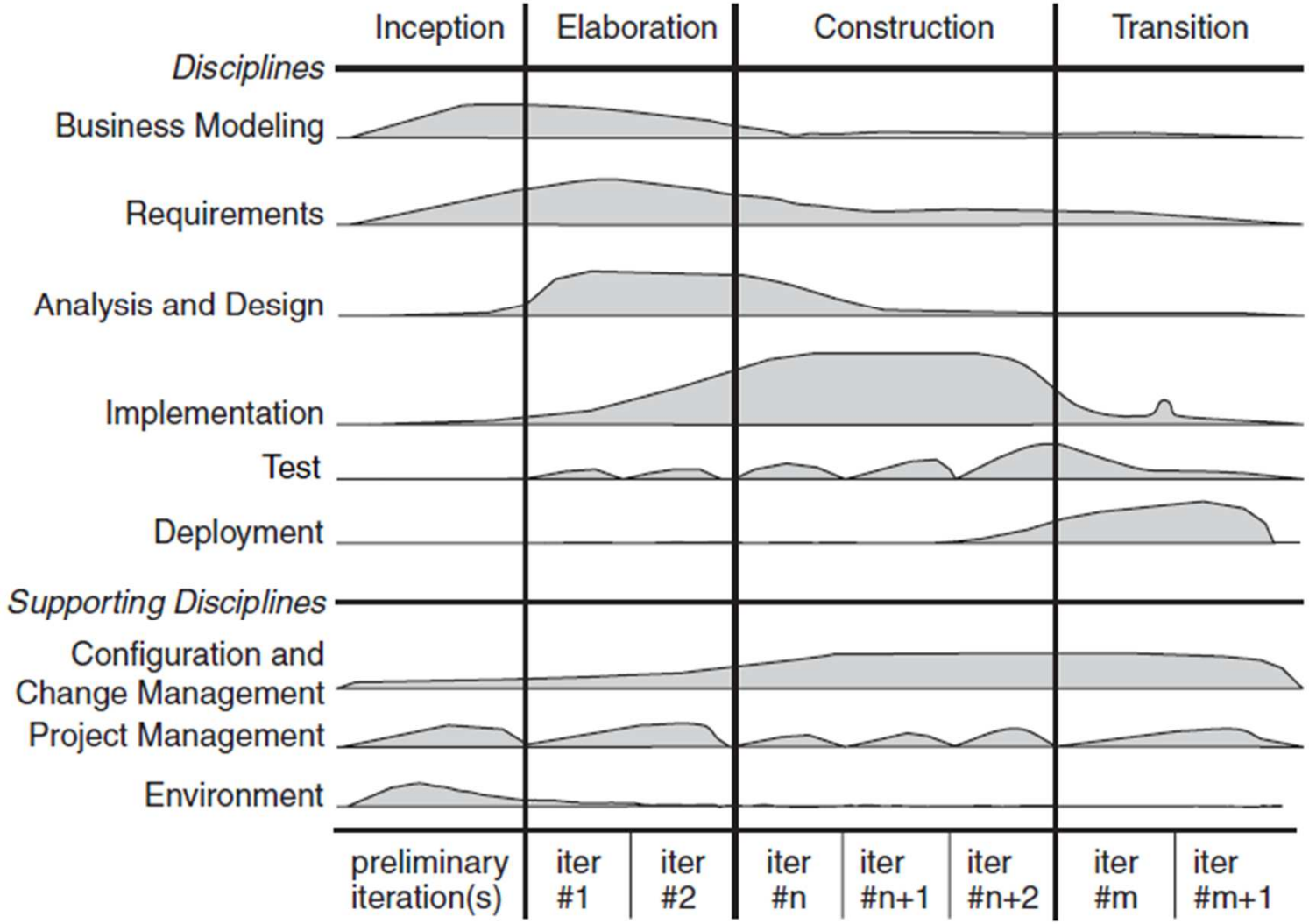


Figure 2-24: Software Development Life Cycle: **RUP**

Software Development Life Cycle: **RUP**

>> **Inception** is the first phase of the process, when the seed idea for the development is brought up to the point of being—at least internally—sufficiently well-founded to warrant entering into the elaboration phase.

>> **Elaboration** is the second phase of the process, when the product requirements and architecture are defined. In this phase, the requirements are articulated, prioritized, and baselined. A system's requirements may range from general vision statements to precise evaluation criteria, each specifying particular functional or nonfunctional behavior and each providing a basis for testing.

Software Development Life Cycle: **RUP**

>> **Construction** is the third phase of the process, when the software is brought from an executable architectural baseline to being ready to be transitioned to the user community. Here also, the system's requirements and especially its evaluation criteria are constantly reexamined against the business needs of the project, and resources are allocated as appropriate to actively attack risks to the project.

>> **Transition** is the fourth phase of the process, when the software is delivered to the user community. Rarely does the software development process end here, for even during this phase, the system is continuously improved, bugs are eradicated, and features that didn't make an earlier release are added.

Software Development Life Cycle: **RUP**

>> One element that distinguishes this process and that cuts across all four phases is an **iteration**.

>> An **iteration** is a distinct set of work tasks, with a baselined plan and evaluation criteria that results in an executable system that can be run, tested, and evaluated. Because the **iteration** yields an executable product, progress can be judged, and risks can be reevaluated after each iteration.

>> This means that the **software development life cycle** can be characterized as involving a continuous stream of executable releases of the system's **architecture** with a midcourse correction after each iteration to mitigate potential risk.

>> It is this emphasis on architecture as an important artifact that drives the **UML** to focus on **modeling** the **different views** of a system's **architecture**.

References

→ **Chapter 2**

The Unified Modeling Language User Guide

By Grady Booch, James Rumbaugh and Ivar Jacobson