Bahria University-Karachi Campus

Software Design & Architecture

Lecture 2 of 16

Engr. Majid Kaleem

WEEKLY AGENDA

TENTATIVE WEEKLY DATES	TENTATIVE TOPICS
1	INTRODUCTION TO THE COURSE; DEFINING SOFTWARE ARCHITECTURE & DESIGN CONCEPTS
2	**DESIGN PRINCIPLES; OBJECT-ORIENTED DESIGN WITH UML**
3	SYSTEM DESIGN & SOFTWARE ARCHITECTURE; OBJECT DESIGN, MAPPING DESIGN TO CODE
4	FUNCTIONAL DESIGN; UI DESIGN; WEB APPLICATIONS DESIGN **ASSIGNMENT & QUIZ #1**
5	MOBILE APPLICATION DESIGN; PERSISTENCE LAYER DESIGN
6	CREATIONAL DESIGN PATTERNS
7	STRUCTURAL DESIGN PATTERNS **ASSIGNMENT & QUIZ #2**
8	BEHAVIORAL DESIGN PATTERNS
← MID TERM EXAMINATIONS →	
9	INTERACTIVE SYSTEMS WITH MVC ARCHITECTURE; SOFTWARE REUSE
10	ARCHITECTURAL DESIGN ISSUES; ARCHITECTURE DESCRIPTION LANGUAGES (ADLS)
11	ARCHITECTURAL STYLES/PATTERNS & DESIGN QUALITIES
12	ARCHITECTURAL STYLES/PATTERNS & DESIGN QUALITIES **ASSIGNMENT & QUIZ #3**
13	QUALITY TACTICS; ARCHITECTURE DOCUMENTATION
14	ARCHITECTURAL EVALUATION TECHNIQUES
15	MODEL DRIVEN DEVELOPMENT **ASSIGNMENT (PRESENTATIONS) & QUIZ #4**
16	REVISION WEEK
← FINAL TERM EXAMINATIONS →	

SOFTWARE DESIGN PRINCIPLES

- Software design principles are concerned with providing means to handle the ***complexity*** of the design process effectively.
- Effectively managing the complexity will not only reduce the *effort needed for design* but can also reduce the *scope of introducing errors* during design.

SOFTWARE DESIGN PRINCIPLES

- *Following are the principles of Software Design:*

1. **Problem partitioning** – *address the problem as a whole or in parts*
2. **Abstraction** – *focus on the big picture and not on details*
3. **Modularity** – *specify the division of software into separate modules*
4. **Top-down & Bottom-up strategy:**
   - ***Top-down Approach:*** *This approach starts with the identification of the main components and then decomposing them into their more detailed sub-components.*
   - ***Bottom-up Approach:*** *A bottom-up approach begins with the lower details and moves towards the hierarchy.*

WHAT IS MODELING?

- **Modeling:** drawing a flowchart listing; the steps carried out by an application.
- **Why do we use modeling?**
- Defining a model makes it easier to break up a complex application or a huge system into simple, discrete pieces that can be individually studied.
- We can focus more easily on the smaller parts of a system and then understand the "big picture."
- **The reasons behind modeling can be summed up in two words:**
  - Readability
  - Reusability

WHAT IS MODELING?

- **Readability:** brings clarity—ease of understanding.
- Understanding a system is the first step in either building or enhancing a system.
- This involves knowing what a system is made up of, how it behaves, and so forth.
- Depicting a system to make it readable involves capturing the structure of a system and the behavior of the system.

WHAT IS MODELING?

- **Reusability:** is the byproduct of making a system readable.
- After a system has been modeled to make it easy to understand, we tend to identify similarities or redundancy in terms of functionality, features, or structure.
- UML provides the ability to capture the characteristics of a system by using notations.
- UML provides a wide array of simple notations for documenting systems based on object-oriented design principles.
- These notations are called the diagrams of UML.

WHAT IS UML?

- The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.
- The UML is a very important part of developing object oriented software and the software development process.
- The UML uses graphical notations to express the design of software projects.
- Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

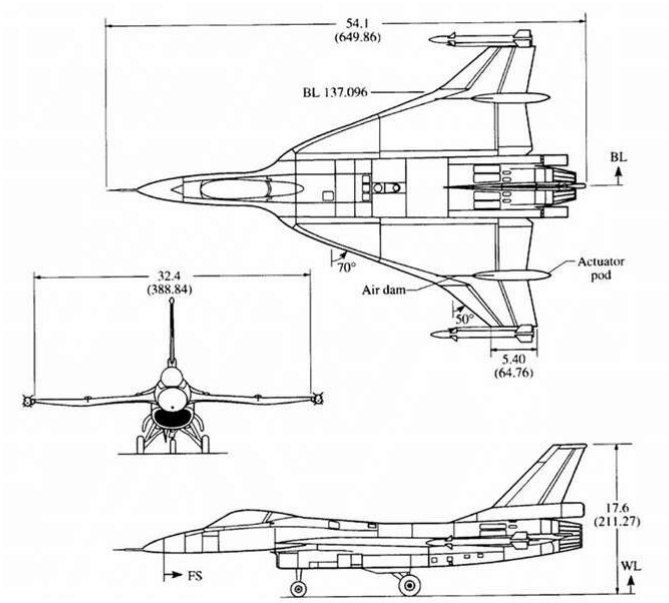
GOALS OF UML

- Provide users with 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.

WHY USE UML?

- The industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market.
- These techniques include component technology, visual programming, and frameworks.
- Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale.
- They recognize the need to solve the architectural problems, such as physical distribution, concurrency, security, load balancing and fault tolerance.
- The Unified Modeling Language (UML) was designed to respond to these needs.

WHY USE UML?



WHY USE UML?

RATIONAL ROSE

- **Rational Rose** is an object-oriented Unified Modeling Language (UML) software design tool intended for visual modeling and component construction of enterprise-level software applications.

MODELING CONCEPTS SPECIFIED BY UML

- System development focuses on three overall different system models:

1. **Functional**: These are Use Case diagrams, which describe system functionality from the point of view of the user.
2. **Object**: These are Class Diagrams, which describe the structure of the system in terms of objects, attributes, associations, and operations.
3. **Dynamic**: Interaction Diagrams, State Machine Diagrams, and Activity Diagrams are used to describe the internal behavior of the system.

- These system models are visualized through two different types of diagrams: **structural** and **behavioral**.

UML Diagram Classification

- A software system can be said to have three distinct characteristics: *static*, *dynamic*, and *implementation*.
  1. **Static:** the structural aspect of the system, define what *parts* the system is made up of.
  2. **Dynamic:** The behavioral features of a system; for example, the ways a system *behaves* in response to certain events or actions are the dynamic characteristics of a system.
  3. **Implementation:** The implementation characteristic of a system is an entirely new feature that describes the different elements required for *deploying* a system.

UML Diagram Classification

- The UML diagrams that fall under each of these categories are:
  - **Static**
    - Use case diagram
    - Class diagram
  - **Dynamic**
    - Object diagram
    - State diagram
    - Activity diagram
    - Sequence diagram
    - Collaboration diagram
  - **Implementation**
    - Component diagram
    - Deployment diagram

UML Modeling Types

- **Structural Modeling**
- Structural modeling captures the static features of a system. They consist of the following:
  - Classes diagrams
  - Objects diagrams
  - Deployment diagrams
  - Package diagrams
  - Composite structure diagram
  - Component diagram

UML Modeling Types

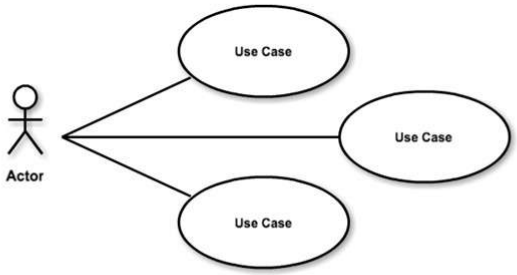
- **Behavioral Modeling**
- Behavioral model describes the interaction in the system. It represents the interaction among the structural diagrams.
- Behavioral modeling shows the dynamic nature of the system. They consist of the following:
  - Activity diagrams
  - Interaction diagrams
  - Use case diagrams

UML Modeling Types

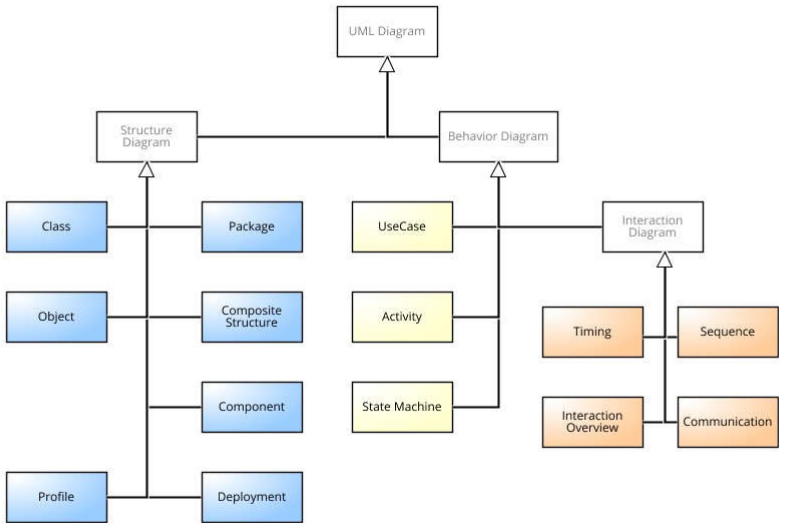
- **Architectural Modeling**
- Architectural model represents the overall framework of the system.
- It contains both structural and behavioral elements of the system.
- Architectural model can be defined as the blueprint of the entire system.
- It consists of the following:
  - Package diagram.

Classification of UML 2.5 Diagrams

- https://www.uml-diagrams.org/uml-25-diagrams.html



CLASSIFICATION OF UML 2.5 DIAGRAMS



STRUCTURE DIAGRAMS

- Structure diagrams represent the **structure**, they are used extensively in documenting the software **architecture** of software systems.

1. **Class diagram**: describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.
2. **Component diagram**: describes how a software system is split up into components and shows the dependencies among these components.
3. **Composite structure diagram**: describes the internal structure of a classifier and the collaborations that this structure makes possible.
4. **Deployment diagram**: describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.

STRUCTURE DIAGRAMS

5. **Object diagram**: shows a complete or partial view of the structure of an example modeled system at a specific time.
6. **Package diagram**: describes how a system is split up into logical groupings by showing the dependencies among these groupings.
7. **Profile diagram**: operates at the metamodel level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype.

BEHAVIOR DIAGRAMS

- Behavior diagrams illustrate the **behavior** of a system, they are used extensively to describe the **functionality** of software systems.

1. **Activity diagram**: describes the business and operational step-by-step workflows of components in a system.
2. **Use Case diagram**: describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.
3. **State machine diagram**: describes the states and state transitions of the system.

BEHAVIOR DIAGRAMS

- Interaction diagrams, a subset of behavior diagrams, emphasize the **flow** of **control** and **data** among the things in the system being modeled:

1. **Sequence diagram**: shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those messages.
2. **Communication diagram**: shows the interactions between objects or parts in terms of sequenced messages.
3. **Interaction overview diagram**: provides an overview in which the nodes represent communication diagrams.
4. **Timing diagrams**: a specific type of interaction diagram where the focus is on timing constraints.


```
If(anyQuestions)
{
  askNow();
}
else
{
  thankYou();
  submitAttendance();
  endClass();
}
```