CHAPTER 3:
PRINCIPLES OF
SOFTWARE ARCHITECTURE
1. RE Fundamentals
2. Architecture Fundamentals
3. Architecture Views
Session I: RE Fundamentals
SESSION’S AGENDA

1. **Software Architecture Process**
   - ✓ Understand and Evaluate Requirements
   - ✓ Design the Architecture
   - ✓ Evaluate the Architecture
   - ✓ Document the Architecture
   - ✓ Monitor and control implementation

2. **Requirements Engineering (RE) Process**
   - ✓ Elicitation
   - ✓ Analysis
   - ✓ Specification
   - ✓ Validation

Software Engineering Design: Theory and Practice
SOFTWARE ARCHITECTURE PROCESS

On a larger scale, the process for creating software architectures can be executed using the following tasks:

1. **Understand and evaluate requirements**
2. **Design the architecture**
3. **Evaluate the architecture**
4. **Document the architecture**
5. **Monitor and control implementation**

Software architects spend a great deal of time working with software requirements.

- Even after requirements are specified, software architects find themselves going back and forth between requirements and design.
- In some cases, architects are completely immersed in the requirements phase, playing a key role in the specification of requirements.
UNDERSTAND AND EVALUATE REQUIREMENTS

- RE definition - The discipline within software engineering that is concerned with the systematic approach to requirements specification, mainly through the following activities:
  1. Elicitation
  2. Analysis
  3. Specification
  4. Validation

- Similar to the design phase, the requirements phase can be broken down into well defined activities
UNDERSTAND AND EVALUATE REQUIREMENTS

1. Elicitation
   ✓ Activity that deals with identifying stakeholders, uncovering what the customer needs and wants, and with determining the non-functional requirements.
   ✓ Begins by identifying all sources of information that can be used to generate requirements.
     ▪ These vary from project to project and can provide bias information to shape the system in a way that addresses their particular needs.
   ✓ Different sources can have similar but different visions for the system.
     ▪ Common sources of requirements include:
       – Stakeholders
       – Goals
       – Domain knowledge
       – Operational and organizational environment

➢ Elicitation Techniques
   ✓ Interviews
   ✓ Facilitated meetings
   ✓ Observation
   ✓ Scenarios
RECAP ON USE CASE DOCUMENTATION = Scenario in EA

1. Use Case Name
2. Short Description of the Use Case
3. Preconditions
4. Normal Flow/Scenario
5. Alternative Flow/Scenarios
6. Post conditions
7. Other Requirements
Scenarios are NOT thrown away after design! They are eventually transformed into Unit Test Cases!

Steps included in the scenario, classified as operator action and system response.

Quality requirements are discovered using scenarios

Important: Scenarios need to be approved!
UNDERSTAND AND EVALUATE REQUIREMENTS

- On eliciting requirements using *Scenarios*
  - A popular approach for eliciting requirements.
  - Allow designers to present stakeholders with storylines about different behaviors that the system is expected to provide.
  - These storylines are born out of the perceived expected behavior by the designer and refined and validated through stakeholders’ reviews.
  - They provide a valuable means for:
    - Establishing a framework for eliciting requirements
    - Identifying major system functions and details of the software
    - Providing initial insight into the required testing of the software.

- In UML, scenarios are grouped by use cases.
  - For each use case, one or more scenarios—one for the main flow of events and other for alternate scenarios—are created to document the expected system behavior and deviations from its main flow of events.
  - Scenarios represent instances of use cases, so there is one-to-many relationship between use case and scenarios.
  - Since there are no universally accepted method for documenting scenarios, they can be found in the following format:
    - Paragraphs, numbered list, tabular or graphical form, etc.
  - Without scenarios, use cases provide limited benefits.
UNDERSTAND AND EVALUATE REQUIREMENTS

1. Analysis

✓ Requirements are analyzed in their raw form to address issues such as requirements that are contradicting, incomplete, vague, or just wrong [1].
✓ Allows architects to clear the air in regard to what needs to be done before devising more detailed designs.
✓ The following tasks can be performed during analysis:
   i. Requirement classification
   ii. Requirement prioritization
   iii. Requirement negotiation
   iv. Conceptual modeling
UNDERSTAND AND EVALUATE REQUIREMENTS

i. Requirement classification

✓ Performed for identifying the nature of each requirement
  ▪ Functional vs. non-functional
  ▪ Product vs. process
  ▪ Imposed vs. derived

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional vs. Non-Functional</td>
<td>Classification that differentiates between requirements that specify the functional aspects of the system vs. the ones that place constraints on how the functional aspects are achieved.</td>
</tr>
<tr>
<td>Product vs. Process</td>
<td>Requirement placed on the system product vs. requirements placed on the process employed to build such product.</td>
</tr>
<tr>
<td>Imposed vs. Derived</td>
<td>Requirements imposed by stakeholders vs. requirements that are derived by the development team.</td>
</tr>
</tbody>
</table>
UNDERSTAND AND EVALUATE REQUIREMENTS

ii. Requirement prioritization and negotiation
   ✓ Helps identify the most important functions of the software system.
   ✓ When done properly, it can help refine the projected schedule by determining which requirements need to be processed first.
   ✓ Can be used to determine different builds of the software
   ✓ Can help during negotiation when conflicts between requirements are identified

iii. Conceptual modeling
   ✓ Conceptual models are created to further identify the requirements by understanding their context, discovering the bounds of the software system, and conceptualizing how the system interacts with its environment.
   ✓ In many projects, this is where architectural design begins, since system decomposition is essential to developing effective conceptual models.
UNDERSTAND AND EVALUATE REQUIREMENTS

iii. Specification and validation

✓ Activity where the results of elicitation and analysis are formally captured and documented in an appropriate format for the use and review of all stakeholders.

✓ The format of the specification varies depending on the developing organization or project; however, it is typically produced as a document, or its electronic equivalent, and is referred as the software requirements specification.

➢ When specifying requirements, it is important that each requirement exhibit certain characteristics desired for designing successful systems. Requirements must be:
  a. Specific
  b. Correct
  c. Complete
  d. Consistent
  e. Attainable
  f. Verifiable
UNDERSTAND AND EVALUATE REQUIREMENTS

a. On being specific,
  ✓ Requirements need to be specified in a clear, concise, and exclusive manner.
  ✓ Clear requirements are not open to interpretation; unclear or ambiguous requirements lead to incorrect designs, incorrect implementations, and deceptive validation during test.
  ✓ Concise requirements are brief and to the point and are therefore easier to understand.
  ✓ Exclusive requirements specify one, and only one thing, making them easier to verify.

For example, consider the following statement:
The software needs to provide an easy-to-use interface; that is, it must be usable.

This statement provides important information to begin thinking about what the customer wants and expects from the software system.

However, in its current form, this statement is too generic to use as basis for design, construction, and verification of the software system.

For example, a console-based interface may be highly usable for developers, but not for customers, which may prefer a graphical user interface!

Designing based on different interpretations of this statement may entail sacrificing other functions that may be important to customers and users!
Example of specific and nonspecific requirements:

<table>
<thead>
<tr>
<th>Specific</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>The software shall search the database.</td>
</tr>
<tr>
<td>Yes</td>
<td>The software shall search for a product using the product ID.</td>
</tr>
<tr>
<td>No</td>
<td>The software shall be secure.</td>
</tr>
<tr>
<td>Yes</td>
<td>The software shall authenticate users with user ID and password.</td>
</tr>
<tr>
<td>No</td>
<td>The software shall be secure and fast.</td>
</tr>
<tr>
<td>Yes</td>
<td>The software shall authenticate users with user ID and password.</td>
</tr>
<tr>
<td>Yes</td>
<td>Server acknowledgment message shall be sent within 1/2 second from the time a request is received.</td>
</tr>
</tbody>
</table>

Secure and fast?

Search the database for what?

What does this mean?

These two requirements are specific and provide the information required to determine what secure and fast mean!
b. On being **correct**,  
   ✓ Requirements need to be correct in the sense that they must **accurately describe a desired system function**.  
   ✓ In some cases, correctness of requirements is easily identified; in others, it is not.  
   ✓ Laplante [1] presents an example based on requirements for a computer security system for which it requires users to **log on using a unique combination of user ID and password**.  
      - In this case, when users attempt to log on using an already existing user name or password, the system is required to reject the attempt, therefore giving insight into someone else’s logon information.  
   ✓ Incorrect requirements can lead to incorrect or undesired behavior.

---

**Display error, per specification**

Attempt to register

Yes, that is right. The system has given away someone else’s logon information!
c. On being *complete*,
   - Requirements must be complete both individually and as collective set.
     - This means that each requirement should be **specified thoroughly** so that it absolutely describes the functions required to meet some need.
     - Collectively, requirements need to provide complete specification of the software’s required functionality in the software requirements specification (SRS).
   - Incomplete requirements lead to incomplete designs, which in turn leads to incomplete construction of the software system.
   - Requirements that are complete help clarify questions during construction and testing by providing information necessary to disambiguate or prevent misinterpretations of required functionality.
   - Completeness is hard because it is not always obvious or it is sometimes too difficult to determine when information is missing [1].

<table>
<thead>
<tr>
<th>Complete</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>The software shall generate product reports.</td>
</tr>
<tr>
<td>Yes</td>
<td>The software shall generate product reports consisting of product description, picture, and price.</td>
</tr>
<tr>
<td></td>
<td>Product reports shall be in PDF format.</td>
</tr>
</tbody>
</table>

Notice how the original requirement is broken into two complete requirements!

This is a good requirement in the sense that it is specific and correct.

However, if we know the requirements for product reports, then a complete version of this requirement specifies this information. This help disambiguate the notion of a product report so that it can be implemented easily in code.
UNDERSTAND AND EVALUATE REQUIREMENTS

d. On being **consistent**,  
   ✓ Requirements are consistent when they **do not preclude** the design or construction of other requirements. = atomic @ fine-grained  
      ▪ Eg: (not consistent)  
         The software shall be able to display the weekly report at least in 5 seconds and in PDF format

e. On being **attainable**,  
   ✓ Requirements that are unattainable **serve no purpose**.  
   ✓ Attainability can be determined for both product and process.  
   **Can you meet this requirement today?**

<table>
<thead>
<tr>
<th>Attainable</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>The software shall execute on all future operating systems.</td>
</tr>
<tr>
<td>Yes</td>
<td>The software shall execute on the Microsoft Windows 7 platform.</td>
</tr>
</tbody>
</table>

f. On being **verifiable**,  
   ✓ Perhaps the most obvious **desirable characteristic of requirements**.  
   ✓ Requirements that cannot be verified cannot be claimed as met. = **testable**  
   ✓ Inability to verify requirements point to a serious flaw early on in the development process

<table>
<thead>
<tr>
<th>Verifiable</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>The system shall maximize communication speed.</td>
</tr>
<tr>
<td>Yes</td>
<td>The system's data rate shall be no less than 1Mbps.</td>
</tr>
</tbody>
</table>
SUMMARY…

➢ Fundamentals concepts of RE; process:
   i. Elicitation
   ii. Analysis
   iii. Specification
   iv. Validation

➢ Information to successfully create good requirements; quality attributes:
   a. Specific
   b. Correct
   c. Complete
   d. Consistent
   e. Attainable
   f. Verifiable
Session II:
Architecture Fundamentals
SESSION’S AGENDA

1. Fundamentals of software architecture
   ✓ What is software architecture?
   ✓ Why is it important?

2. Key tasks in software architecture
   i. Stakeholders concerns
   ii. Identify architectural views, styles, and patterns
   iii. Identify major component’s and interfaces
   iv. Evaluating and validating the Architecture
   v. Introducing policies for design synchronicity

3. Problem-solving in architecture
   - Recap from Chapter 1
1. FUNDAMENTALS OF SOFTWARE ARCHITECTURE

- Software architecture **definition**:
  - The *foundational software design* activity that evaluates and translates *software requirements* (both functional and non-functional) into a *collection of design elements* that specify structural and behavioral aspects of the *major components of the system*, together with their provided quality and interrelationships required to *support the detailed design and construction* of software systems.
  - The product resulting from such activity.

- From this definition, a few things are of interest and need further explanation:
  - Foundational design
  - Transforming requirements
  - Collection of design elements
  - Major system components together with their provided quality
  - Support detailed design and construction
1. **Fundamentals of Software Architecture**

- On “...foundational software design...”
  - Software architecture provides the groundwork essential for meeting requirements
    - This applies to both functional and non-functional requirements.
  - This suggest that architecture is not an optional activity or activity performed as a means of documenting software systems long after they are implemented.
    - New development efforts should approach software architecture as a forward engineering activity that leads to the implementation of systems and not as a reverse engineering mechanism for documentation.
  - As foundational design, it is where designing for quality begins. Not considering quality attributes of the system during the software architecture activity can be a grave mistake!

- On “...translates software requirements...”
  - Requirements is a tricky business! Inexperienced engineers tend not to see the many traps behind the requirements effort.
  - Assuming that every requirement is captured and well understood, design elements need to be created so that there is a mapping between requirements and design element.
    - One design element (e.g., UML component) can be assigned one or more requirements.
    - When we do this, we transform one or more requirements from textual form into a graphical form that represents (in the design domain) the services that need to be provided by the system.
    - This allows us to map requirements to design elements and provide the means for tracing requirements through the development life-cycle.
1. **Fundamentals of Software Architecture**

- More on “...translates software requirements...”
  - To create design elements from requirements, it is assumed that requirements are understood. Sometimes this is not the case.
    - For example, some may think that “The system shall perform fast.” specifies a requirement that can be used to create design elements.
    - Such statements create problems for designers. These problems need to be resolved before we can translate from requirement to design domain.
    - This suggest that architects must be proficient in activities related to requirements engineering. We will cover such situations later on in the course.

- On “...collection of design elements...”
  - This suggest that no one structure or diagram can fully describe the software architecture.
    - Think about this: can you evaluate a system’s usability and performance with one diagram?
  - This suggest that architectures are composed of multiple structures.
  - We will see examples of this later on...
On “...major components of the system, together with their provided quality...”

- This suggests that software architecture works at a distinct level of abstraction that differs from other forms of design, such as detailed design.
- This means that architectural work focuses on the major components, the quality properties, and services that these components exhibit and provide to other components.

---

Example A

I provide an Algorithm

I Optimizer

Example B

I provide an Algorithm

The architectural effort requires designers to focus not only on decomposition, but also on the quality provided by identified components! This is equivalent to tagging a component with important information, so that its quality is known by clients using services from the component.

Client needing a fast algorithm

O(n^3)! Yikes!

Component Quality:
Performance: O(n^3)
Reusability: Low...

---

Component
<<component>>

performance:
O(n^3)

Component
<<component>>

performance:
O(n^3)
1. Fundamentals of Software Architecture

- More on “…major components of the system, together with their provided quality…”
  - Expected quality requirements identified during architecture trickles down all the way to the implementation of components.
    - This provides developers with enough information to produce code that meets the system’s functional and non-functional requirements.

Important: We are focusing here on quality requirements, but the same applies to functional requirements.
1. **Fundamentals of Software Architecture**

- On “…support detailed design and construction…”
  - Although architecture focuses on the quality properties of the system, it must also result in a design that supports efficient detailed design and construction of the system.
    - Even though Architects do not need to be proficient in a particular programming language, they benefit greatly from having proficiency in general programming design concepts.
  - This suggests that architecture alone cannot guarantee the quality of the system!
    - Since work performed during subsequent activities and phases significantly shapes the system’s quality, software architecture can only play the initial (indispensable) role of establishing the design quality framework for the rest of the development process.

---

Architect spends time and effort defining what security means to customers!
2. **Key Tasks in Architectural Design**

- From the software architecture definition, we have been able to derive key tasks that need to be performed during software architecture.
  - However, defining the structure of software systems requires consideration of many other project-specific aspects and how those aspects relate to the organization’s goals.
  - Formally, key tasks that need to be performed during the software architecture design effort include:
    i. Identifying stakeholders concern
    ii. Identifying appropriate architectural views
    iii. Identifying architectural styles and patterns
    iv. Identifying influences of architectural decisions in organizations
    v. Identifying the system’s major components and interfaces
    vi. Evaluating and validating the architecture
    vii. Establishing policies for ensuring architectural design synchronicity
2. KEY TASKS IN ARCHITECTURAL DESIGN

i. Identifying stakeholders concerns

✓ Stakeholders are persons, groups, or organizations that have a direct or indirect stake in the system.
  ▪ They include systems engineers, software engineers, hardware engineers, project management, customers, testing teams, quality assurance teams, members of the configuration management team, etc.

✓ A stakeholder’s concern provides high-level information about desired characteristics of the software system.
  ▪ The software architect must ensure that the software to be developed addresses the concerns of all stakeholders.
  ▪ Stakeholders’ concerns serve as driving force behind architectural decisions

✓ The software architect must identify and understand the different ways stakeholders influence the system.
  ▪ These need to be elicited before any design effort can begin.

**Important:**
Stakeholders’ concerns serve as driving force for architectural decisions
2. Key Tasks in Architectural Design

- So far, we’ve vaguely mentioned the concepts of **quality** and stakeholders’ concerns.
  - These high-level concerns are often related to the desired quality of the system.
- Let’s formally define some important quality attributes of software systems.

<table>
<thead>
<tr>
<th>Quality</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability</td>
<td>The degree of complexity involved when learning or using the system.</td>
</tr>
<tr>
<td>Modifiability</td>
<td>The degree of complexity involved when changing the system to fit current or future needs.</td>
</tr>
<tr>
<td>Security</td>
<td>The system’s ability to protect and defend its information or information system.</td>
</tr>
<tr>
<td>Performance</td>
<td>The system’s capacity to accomplish useful work under time and resource constraints.</td>
</tr>
<tr>
<td>Reliability</td>
<td>The system’s failure rate.</td>
</tr>
<tr>
<td>Portability</td>
<td>The degree of complexity involved when adapting the system to other software or hardware environments.</td>
</tr>
<tr>
<td>Testability</td>
<td>The degree of complexity involved when verifying and validating the system’s required functions.</td>
</tr>
<tr>
<td>Availability</td>
<td>The system’s uptime.</td>
</tr>
<tr>
<td>Interoperability</td>
<td>The system’s ability to collaborate with other software or hardware systems.</td>
</tr>
</tbody>
</table>
2. **Key Tasks in Architectural Design**

- Notice that these quality attributes also describe **high-level information** about desired characteristics of the software system.
- In their current form, they are not sufficient to develop the system.
- For a system to exhibit any of these qualities, design decisions must be made to support the achievement of these qualities. These design decisions are referred by Bass, Clements, and Kazman as **Tactics** [1].
According to Bass, Clements, and Kazman, a tactic is a design decision that influences the control of a quality attribute response [1].

**Important:**
In many cases, these quality goals fall through the requirement phase, leaving the architect responsible for specifying requirements to meet these quality attributes. During this process, tactics are identified for each desired quality attribute.
2. **Key Tasks in Architectural Design**

a. Tactics for *Security* [1]
   - **Resisting Attacks**
     - Authenticating users, Limit exposure, Limit access, only on need-to-know basis, etc.
   - **Detecting Attacks**
     - Intrusion detection

b. Tactics for *Testability*
   - **Event logging**
     - Log data and operations throughout the system. Allow testers to enable/disable this feature so that when enabled, events are displayed in the console to give insight into the system’s operations and data.

c. Tactics for *Modifiability* [1]
   - **Localize changes**
     - Modularization, abstraction, encapsulation
   - **Prevention of ripple effects**
     - Encapsulation, reduce coupling

d. Tactics for *Availability*
   - **Redundancy, Task monitor, Watchdog timer**, etc.

e. Tactics for *Performance* [1]
   - Increase *computation efficiency*, reduce *computational overhead*, introduce *concurrency*, etc.
2. **Identifying appropriate architectural views**

- In complex software systems, there can be numerous stakeholders with different backgrounds.
  - These stakeholders have different perception about the system, which influence the way they evaluate the system’s design.
- For this reason, architectural designs must support different architectural views used to evaluate the design from a particular stakeholder’s perspective.
- An architectural view is a representation of the system.
  - Different representations are required to evaluate certain properties of the system.

**View 1**
- This view is appropriate to evaluate the system’s usability.

**View 2**
- This view is appropriate to evaluate the system’s reusability.

We’ll see more relevant views during session 3.
2. KEY TASKS IN ARCHITECTURAL DESIGN

3. Identifying Architectural Styles and Patterns
   ✓ The concept of architectural styles and patterns are fundamental to the efficient creation of software architectures.
   ✓ They provide an overall strategy for designing a family of software systems.
   ✓ They provide reusable architectural solutions, documented in a way that is easily understood and applied.
     ▪ For this reason, from the logical perspective, this is one of the first tasks performed during architecture.
   ✓ Decisions based on styles and patterns benefit from years of documented experience.
   ✓ Today, numerous styles and patterns exist so architects must be aware of these so that they can identify and determine the appropriateness of a particular style or pattern for their system’s design.

   Important:
   Don’t get stuck on terminology! As you will see later on, Architectural Styles and Architectural Patterns refer to similar concept. During Detailed Design, you will also hear about Design Patterns!

4. Identifying System Interfaces
   ✓ Interfaces are defined for components residing within single physical nodes within a single or different process space, or for components residing on different physical nodes.

5. Identifying Impact of Architectural Decisions in Organization ~ Influence factors
   i. Impact on customer base
   ii. Impact on budget and schedule
   iii. Impact on resource availability
2. **KEY TASKS IN ARCHITECTURAL DESIGN**

6. **Evaluating and Validating the Architecture**
   - ☑ Long and iterative process
   - ☑ Failure to do so can have significant impact in effort and cost incurred to develop the system.
     - It is well known that defects found earlier on in the development process take much less effort to correct than if found at later stages.
   - ☑ The result should determine if the software architecture is *sufficiently* complete to support the development of the system.
   - ☑ We’ll have more to say about this later on in the course…

7. **Introduce Policies for Design Synchronicity**
   - ☑ Design synchronicity is a measurement of the degree of how well the software implementation reflects its design, both in architectural form and detailed design form.
   - ☑ Obviously, we want high synchronicity, but this is not always the case.
     - It is very easy to deviate from design, especially on multi-year efforts.
   - ☑ For any software architecture effort to result in successful implementation, all subsequent phases and activities must be synchronized with the architecture.
     - To do this, a well-defined and understood process must be in place.
     - This includes the maintenance phase, which can go on for years after a software has been deployed!
3. **Problem Solving During Architecture**

- **Other project related constraints**
  - Schedule Constraints,
  - Cost Constraints,
  - Quality Constraints

- **These define the problem and product constraints**
  - Requirements,
  - System Goals,
  - Scenarios

- **For example, we will reuse our home-grown class framework to leverage our expertise and develop faster!**

- **Another example, we will buy and integrate xyz library into our product instead of developing it ourselves**

- **Tasks performed to process the requirements**
  - Example: Design Review

- **These dictate how, when, and where activities are performed!**

- **Development team, modeling software, IDEs, hardware, etc.**

- **Unfortunately, these are rarely complete on the first shot!**

- **Controls Examples:**
  1. Design reviews must be scheduled with one week of anticipation
  2. Software Quality Assurance personnel must be present during review.
  3. Review of material containing classified information must occur inside a vault

- **Resources**
  - **Evaluate Constraints**
  - **Document**
  - **Interpret Problem**
  - **Collaborative Brainstorming**
  - **Evaluate**
  - **Architectural Design**
  - **Activities**
  - **Controls**
BRING ME PROBLEM, NOT SOLUTIONS

http://softwarearchitecturezen.blogspot.com/2012/10/bring-me-problems-not-solutions.html
SUMMARY…

- Fundamentals concepts of software architecture
  - Foundational design
  - Transforming requirements
  - Collection of design elements
  - Major system components together with their provided quality
  - Support detailed design and construction

- Key tasks during software architecture:
  1. Identifying stakeholders concerns
  2. Identify architectural views, styles, and patterns
  3. Identify major component’s and interfaces
  4. Evaluating and validating the Architecture
  5. Introducing policies for design synchronicity
Session III: Architecture Views
SESSION’S AGENDA

1. Designing the Architecture with the 4+1 view model
   i. Designing the logical architecture
      ▪ Example using a component diagram
   ii. Designing the process architecture
      ▪ Example using a communication diagram
   iii. Designing the development architecture
      ▪ Example using artifacts
   iv. Designing the physical architecture
      ▪ Example using deployment diagrams
As discussed before, designing architectures require the selection of particular perspectives for design that are appropriate for describing the system to be developed.

- To this end, several models have been created that suggest popular views that are useful in the design of most systems.
- These models propose addressing the system’s architectural design from perspectives that are common to most software systems.
- They provide a *systematic and disciplined* approach to architecture creation.
- Two popular models are the 4+1 view model and Siemens' 4 views model.

**The 4+1 View Model**

- Concentrates on four main views:
  - Logical, process, development, and physical
- To ensure consistency among all 4 views, the user’s perspective is captured through several use cases, as part of the user view.
- 4+1 is used as the basis for the **popular rational unified process**
I. Designing the Logical Architecture

- The logical view is used to decompose systems into logical components that represent the structural integrity that supports functional and non-functional requirements. The logical view can be modeled using:
  - Component diagrams
  - Class diagrams
  - Box-and-line diagrams

- Using this view the static structure of the system, including components, interfaces, and their associations are modeled. This view is appropriate for:
  - Designing for portability
  - Designing maintainability (maintaining existing features)
  - Designing for extensibility (adding new features)
  - Designing for reusability
  - Resource allocation, project structuring and planning
I. DESIGNING THE LOGICAL ARCHITECTURE

Video controller communicates with the actual video collection hardware to command it!

No interface defined! Direct association suggests that these components are tightly coupled, therefore making it more difficult to modify the system later on!

SiteController communicates with the outside world. It requires a schedule and provides collected data.

Important: This process is recursive, i.e., each component can be further decomposed!

Well-defined interface! This allows different user interfaces to be exchangeable easier, e.g., from console-to graphical user interface-based. on!
The process view is used to represent the dynamic or behavioral aspects of software systems, where the main units of analysis are processes and threads.

With this view, the system is decomposed into processes and threads to address design issues that deal with the dynamic flow of control between architectural elements.

The process view can be modeled with UML using:

- Sequence diagrams
- Communication diagrams

This view is appropriate for evaluating important characteristics of the system, such as concurrency, fault tolerance, and the system’s integrity. Specific quality attributes that can be evaluated with this view include:

- Performance
- Availability
II. DESIGNING THE PROCESS ARCHITECTURE

Threads in the system using asynchronous communication

Based on the received message, the SiteManager initiates data collection using the sensors.

Based on the received message, the SiteManager sends an immediate ACK response, to indicate that the message is received. After that, it updates the user interface to display the receipt of the message.

The Communication Manager executes under the SiteManager's context

Based on the received message, the SiteManager stops data collection using the video camera.

The SiteManager uses the CommMgr to receive a message. After receipt, the SiteManager sends an immediate ACK response, to indicate that the message is received. After that, it updates the user interface to display the receipt of the message.
III. DESIGNING THE DEVELOPMENT ARCHITECTURE

- The development view of software systems represents the software development configuration aspects of the software system.
  - The main units of decomposition are actual physical files and directories.
- This view can be used to analyze the system from the perspective of how logical components map to physical files and directories.
  - These analyses can be employed to address concerns that deal with:
    - Ease of development
    - Reusability
    - Compile-time dependencies of the system
    - Configuration management
    - Allocation of work to teams
    - Cost evaluation and planning
    - Project monitoring
    - Portability
III. DESIGNING THE DEVELOPMENT ARCHITECTURE

Version 1.0 is used by a specific customer that does not require video capture capabilities.

Version 2.0 is used by a specific customer that requires video capture capabilities.

Notice how versions change with the addition of new functionality!
IV. DESIGNING THE PHYSICAL ARCHITECTURE

- The physical view represents the deployment aspects of software systems.
  - The main elements of analysis are nodes, connections between nodes, and the mapping of artifacts to these nodes.
  - This view can be used to model the run-time dependencies of the system.

- The physical view can be modeled in UML using:
  - Deployment diagrams

- The physical view is appropriate to evaluate the system’s:
  - Availability
    - For example, specifying the need for redundant nodes to increase the system’s availability
  - Performance
    - For example, required processor speed, bandwidth, etc. to meet performance requirements.
  - Scalability
DESIGNING THE PHYSICAL ARCHITECTURE

![Diagram of physical architecture design]

Software Engineering Design: Theory and Practice
EXAMPLE OF PARTIAL ARCHITECTURE OF CLIENT COLLECTION SYSTEM

Logical View

Development View

Process View

Physical View

Software Engineering Design: Theory and Practice 54
4+1 view model from stakeholder perspectives
ONE LAST NOTE...

- Do NOT make the assumption that a particular UML classifier belongs exclusively to a particular view of the 4+1 model!!
  - Although some classifiers are certainly “the main characters” in particular views, they can serve as “supporting characters in other views.”
    - For example, components are used mostly in the logical view, BUT,
    - They can also be used in the physical view to model how they are manifested by artifacts!

- UML Artifacts are tricky, because they can be used as the dominant design unit (i.e., “main character”) in both development and physical view:
  - In the physical view, they model deployment artifacts.
    - These are artifacts necessary to form an executable system (e.g., .dll, .jar, .class, .ini file, etc.)
    - These are essential to model the run-time dependencies of the system (e.g., .NET 4.1)
  - In the development view, they model work product artifacts
    - These are artifacts used to create deployment artifacts, i.e., they are necessary to build the software (e.g., source code files [.h, .cpp, .java], data files])
    - These artifacts do not directly exist in the physically deployed system; they are the work products of development used to build the system.
    - These are essential to model the compile-time dependencies to build the system as well as versioning and configuration management of the system.

- Keep in mind that when using UML artifacts, modeling occurs in the physical dimension, which can be addressed in both development and physical views of the 4+1 model. Artifacts do not exist in the logical dimension, as opposed to other UML classifiers, such as components.
In this session, we presented the 4+1 view model for designing software architectures, which includes

i. Logical view
ii. Development View
iii. Process View
iv. Physical View
v. User View