Chapter 7:
Moving on to Design

Learning Objectives

- Understand the verification and validation of the analysis models.
- Understand the transition from analysis to design.
- Understand the use of factoring, partitions, and layers.
- Be able to create package diagrams.
- Be familiar with the custom, packaged, and outsource design alternatives.
- Be able to create an alternative matrix.

Introduction

- Analysis determines the business needs
- Design activities focus on how to build the system
  - Major activity is to evolve the models into a design
  - Goal is to create a blueprint for the design that makes sense to implement
  - Determine how and where data will be stored
  - Determine how the user will interface with the system (user interface, inputs and outputs)
  - Decide on the physical architecture
- Analysis and design phases are highly *interrelated* and may require much "going back and forth"
  - Example: prototyping may uncover additional information

The Design Process

- Verify and validate the analysis models
- Evolve the analysis models into design models
- Create packages and utilize package diagrams
- Decide upon a design strategy

Verifying & Validating the Analysis Models

- Do the analysis models accurately represent the problem domain?
  - Test the fidelity of each model
  - Example: activity diagrams, use-case descriptions and use-case diagrams should all describe the same functional requirements
- Balance the models to ensure consistency between them

Balancing Functional & Structural Models

- A class on a class diagram must be associated with at least one use-case
- An activity in an activity diagram and an event in a use-case description should be related to one or more operations on a class diagram
- An object node on an activity diagram must be associated with an instance or an attribute on a class diagram
- An attribute or an association/aggregation relationship on a class diagram should be related to the subject or object of a use-case

Balancing Functional & Behavioral Models

- Sequence & communication diagrams must be associated with a use-case
- Actors on sequence & communication diagrams or CRUDE matrices must be associated with actors within a use-case
- Messages on sequence & communication diagrams, transitions on behavioral state machines and entries in a CRUDE matrix must relate to activities on an activity diagram and events in a use-case
- All complex objects in activity diagrams must be represented in a behavioral state machine

Balancing Structural & Behavioral Models

- Objects in a CRUDE matrix must be associated with classes
- Behavioral state machine must be associated with objects on a class diagram
- Objects in sequence and communication diagrams must be associated with objects on a class diagram
- Messages on sequence and communication diagrams and transitions on behavioral state machines must be associated with operations in a class
- States in a behavioral state machine must match the different values of an attribute of an object

Evolving the Analysis Models into Design Models

- Analysis models focused on functional requirements
- Design models must include non-functional requirements as well
  - System performance
  - System environment issues
    - Distributed vs. centralized processing
    - User interface
    - Database
- The system must be maintainable and affordable, efficient and effective
- Utilize factoring, partitions & collaborations, and layers

Factoring

- Creating modules that account for similarities and differences between units of interest
- New classes formed through a:
  - Generalization (a-kind-of) relationship, or a
  - Aggregation (has-parts) relationship
- Abstraction—create a higher level class (e.g., create an Employee class from a set of job positions)
- Refinement—create a detailed class (e.g., create a secretary or bookkeeper from the Employee class)

Partitions and Collaborations

- Partition: create a sub-system of closely collaborating classes
  - Base partitions on patterns of activity (e.g., collaborations found in a communication diagram)
  - Greater coupling among classes may identify partitions (e.g., more messages passes between objects suggests that they belong in the same partition)
- Identifying partitions and collaborations determines which classes should be grouped together

Layers

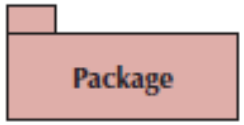
- System environment information must now be added
- Use layers to represent and separate elements of the software architecture
  - Easier to understand a complex system
  - Example:
    - Model-view-controller (MVC) architecture
    - Separates application logic from user interface
  - Proposed layers:
    - Foundation (e.g., container classes)
    - Problem domain (e.g., encapsulation, inheritance, polymorphism)
    - Data management (e.g., data storage and retrieval)
    - User interface (e.g., data input forms)
    - Physical architecture (e.g., specific computers and networks)

Packages and Package Diagrams

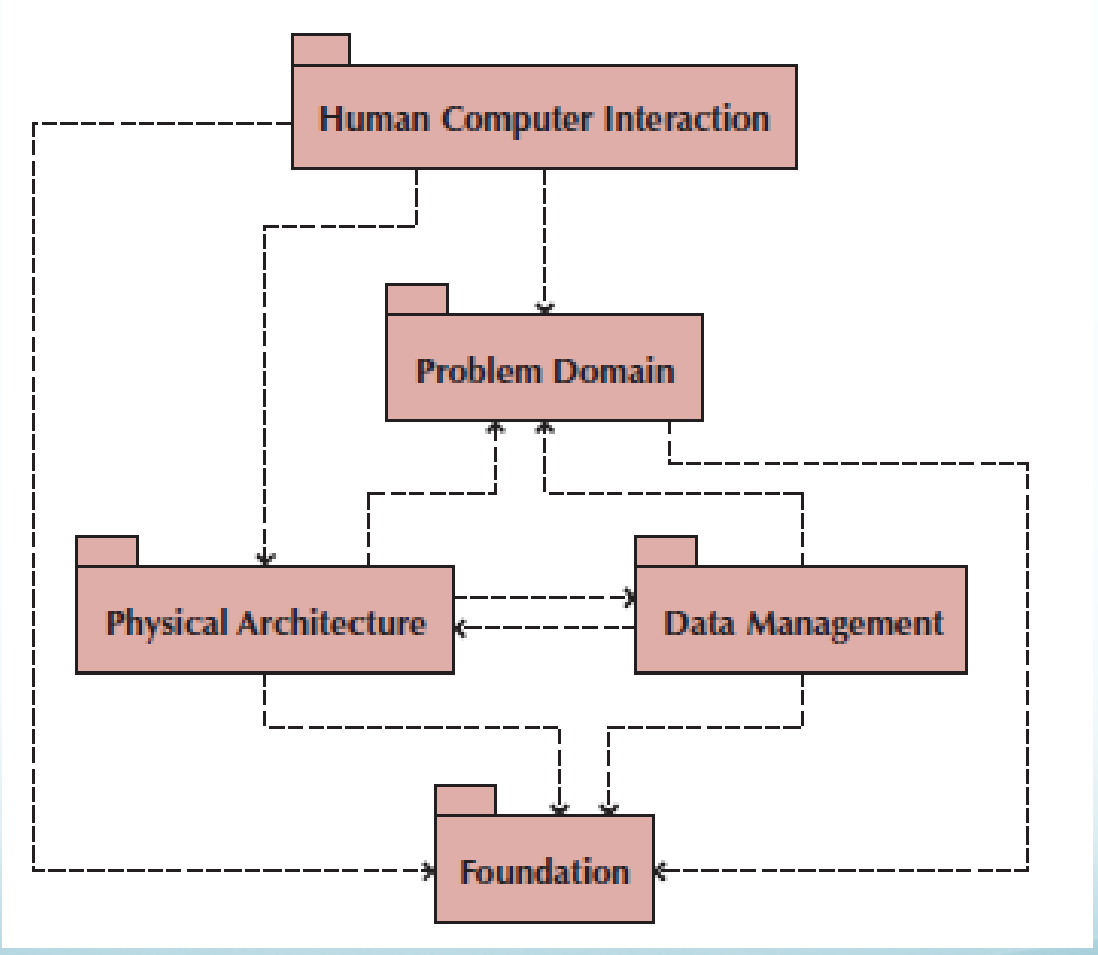
- Packages group together similar components (e.g., use-cases, class diagrams)
- Package diagrams show the packages and their relationships
  - Aggregation & association relationships are possible
  - Packages may be dependent upon one another
    - If one package is modified, others that depend on it may also require modification

Package

- A general construct that groups units together
- Used to reduce complexity of models
- A package diagram shows packages only

A package: ■ Is a logical grouping of UML elements ■ Is used to simplify UML diagrams by grouping related elements into a single higher-level element.	
A dependency relationship: ■ Represents a dependency between packages: If a package is changed, the dependent package also could have to be modified. ■ Has an arrow drawn from the dependent package toward the package on which it is dependent.	

Sample Package Diagram



Guidelines for Building Package Diagrams

- Use them to logically organize your design
- Observe semantic relationships
  - Vertical positioning indicates inheritance
  - Horizontal positioning indicates aggregation and association
- Dependency relationships should also observe semantic relationships
- For use-case package diagrams, include the actors
- Use simple but descriptive names for each package
- Make packages cohesive

Building Package Diagrams

- Set the context
- Cluster classes together based on shared relationships
- Create packages from the clusters
- Identify dependency relationships among packages
- Lay out and draw the diagram including only the packages and their dependencies
- Verify and validate the package diagram

Design Strategies

- Custom development—build it in house from scratch
- Purchase packaged software
  - Office suites (e.g., word processors, spreadsheets, etc.)
  - Enterprise systems (e.g., SAP, PeopleSoft)
- Hire an external vendor (outsource)

Custom Development

- Allows for meeting highly specialized requirements
- Allows flexibility and creativity in solving problems
- Easier to change components
- Builds personnel skills
- May excessively burden the IT staff
- May add significant risk

Packaged Software

- Software already written (e.g., accounting software)
- May be more efficient
- May be more thoroughly tested and proven
- May range from components to tools to enitre enterprise systems
- Must accept functionality provided
- May require change in how the firm does business
- May require significant “customization” or “workarounds”

System Integration

- Building a new system by combining packages, legacy systems, and new software
  - Not uncommon to purchase off the shelf software and outsource its integration to existing systems
- Key challenge is integrating data
  - May require data transformations
  - New package may need to write data in the same format as a legacy system
- Develop "object wrappers"
  - Wraps the legacy system with an API to allow newer systems to communicate with it
  - Protects the investment in the legacy system

Outsourcing

- Hire an external firm to create the system
  - Requires extensive two-way coordination, information exchange and trust
  - Disadvantages include loss of control, compromise confidential information, transfer of expertise
  - Carefully choose your vendor
  - Carefully prepare the contract and method of payment
- Contract types:
  - Time-and-arrangement: pay for all time and expenses
  - Fixed-price: pay an agreed upon price
  - Value-added: pay a percentage of benefits

Selecting a Design Strategy

	Use Custom Development When…	Use a Packaged System When…	Use Outsourcing When…
Business Need	The business need is unique.	The business need is common.	The business need is not core to the business.
In-house Experience	In-house functional and technical experience exists.	In-house functional experience exists.	In-house functional or technical experience does not exist.
Project Skills	There is a desire to build in-house skills.	The skills are not strategic.	The decision to outsource is a strategic decision.
Project Management	The project has a highly skilled project manager and a proven methodology.	The project has a project manager who can coordinate the vendor's efforts.	The project has a highly skilled project manager at the level of the organization that matches the scope of the outsourcing deal.
Time frame	The time frame is flexible.	The time frame is short.	The time frame is short or flexible.

Developing the Actual Design

- Determine tools and skills needed for in-house development
- Identify existing packages that satisfy the users' needs
- Locate companies who can build it under contract
- Create an alternative matrix to organize the pros and cons of each possible choice
  - Incorporate technical, economic and organizational feasibility
  - Utilize an RFP or RFI to obtain cost & time estimates from potential vendors

Summary

- Verifying and Validating the Analysis Models
- Evolving the Analysis Models into Design Models
- Packages and Package Diagrams
- Design Strategies
- Developing the Actual Design