


DA-IICT




IT 314: Software Engineering

Object-Oriented System Design

Saurabh Tiwari

1

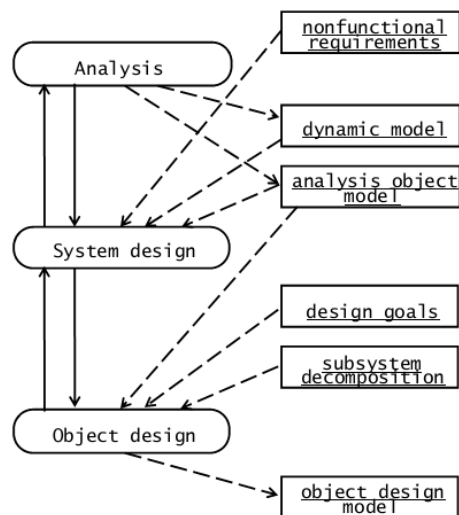


Recap (Analysis Modeling)

1. Analyze the problem statement
 - Identify functional requirements
 - Identify nonfunctional requirements
 - Identify constraints (pseudo requirements)
2. Build the functional model:
 - Develop use cases to illustrate functional requirements
3. Use case realizations by building **dynamic models**:
 - Develop **sequence diagrams** to illustrate the interaction between **domain objects**
 - Develop **state diagrams** for **domain objects** with interesting behavior
4. Build the domain object model:
 - Develop **class diagrams** (at domain level) showing the structure of the system

Designing the System

The activities of System Design



Analysis Model

- Analysis model = **first cut** at design model
- In theory: refine analysis model into design model in a structure-preserving way
 - But may not be possible due to **design/implementation** constraints (Platforms, NFRs, Reuse, etc.)
 - Design is **not algorithmic** activity
- What is in design but not analysis:
 - Decisions on **performance and distribution requirements**; optimizations etc.
 - Decisions on **choosing architecture**
 - Decisions on **data structures** and **persistent** (database) storage
 - Decision on **Reuse**
 - Analysis should prepare for design by giving a **thorough understanding of the requirements**

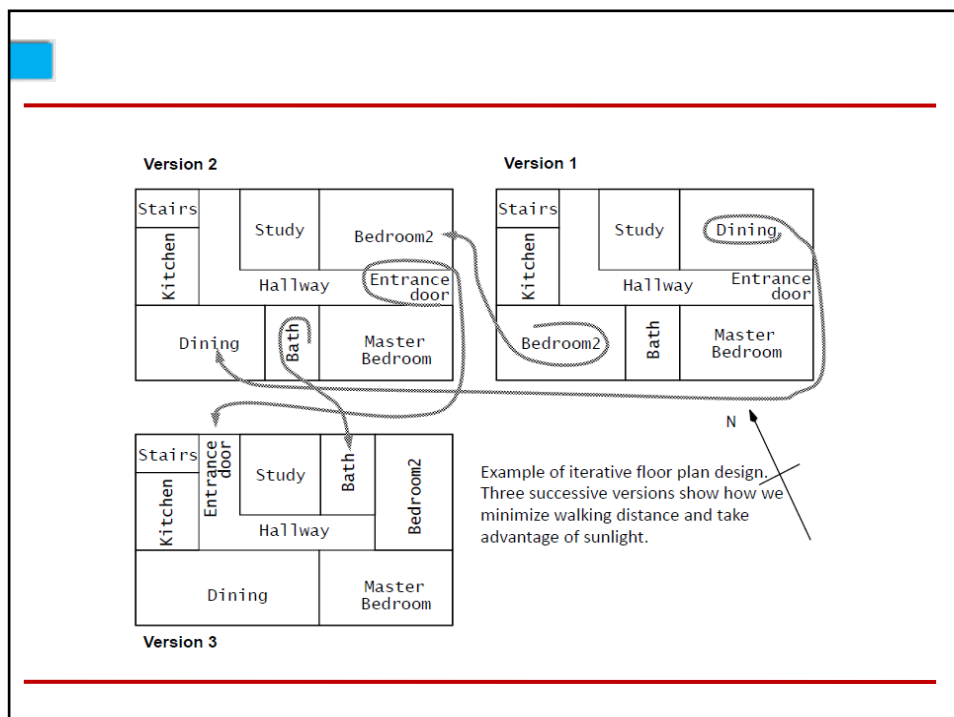
Analysis vs. Design model

Conceptual model (avoids implementation issues)	Physical model (blueprint of implementation)
Applicable to several designs	Specific for one implementation
Three conceptual stereotypes (entity, boundary, and control)	Any number of physical stereotypes depending on programming language
Less formal	More formal
Less expensive to develop	More expensive to develop
Few layers	Many layers
May not be maintained	Should be maintained

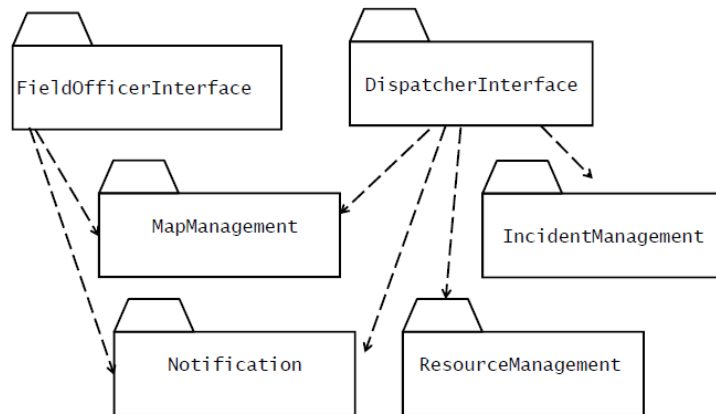
Jacobson/Booch/Rumbaugh

System Design

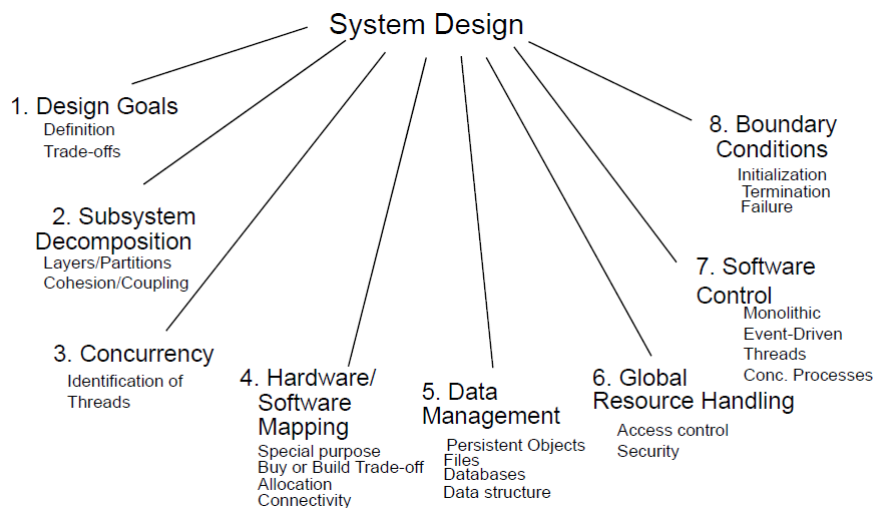
- Identify **design** goals
- Design the **initial subsystem decomposition**
 - **Analysis model** and use cases
 - Use standard **Architectural Styles**
- Refine the subsystem decomposition



Example: FRIEND



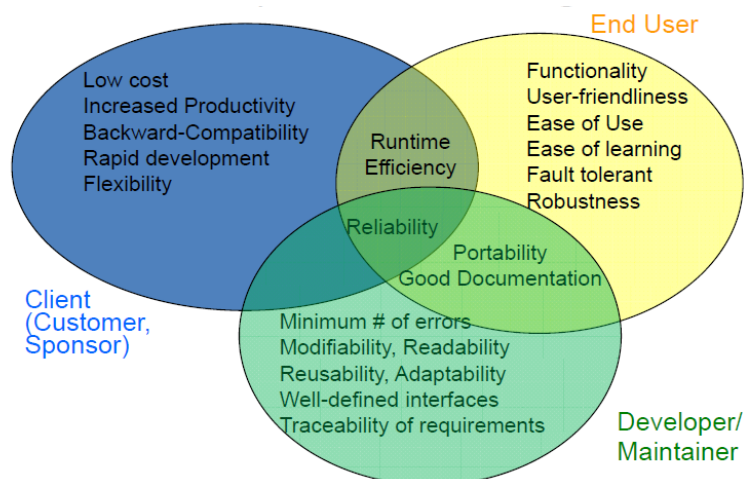
Subsystem Design



List of Design Goals

- Reliability
- Modifiability
- Maintainability
- Understandability
- Adaptability
- Reusability
- Efficiency
- Portability
- Traceability of requirements
- Fault tolerance
- Backward-compatibility
- Cost-effectiveness
- Robustness
- High-performance
- Good documentation
- Well-defined interfaces
- User-friendliness
- Reuse of components
- Rapid development
- Minimum # of errors
- Readability
- Ease of learning
- Ease of remembering
- Ease of use
- Increased productivity
- Low-cost
- Flexibility

Relationship between Design Goals



Typical design Trade-Offs

- Functionality vs. Usability
 - Cost vs. Robustness
 - Efficiency vs. Portability (or Modifiability)
 - Rapid development vs. Functionality
 - Cost vs. Reusability
 - Backward Compatibility vs. Readability
-

Design the System

Subsystem decomposition

- Subsystems
 - Subsystem Structure
 - Subsystem Interfaces
-

Subsystem Decomposition

Subsystem (UML: Package)

- Collection of classes, associations, operations, events and constraints that are interrelated
- Seed for subsystems: Objects and Classes.

(Subsystem) Services:

- Group of operations provided by the subsystem
- Seed for services: Subsystem use cases

(Services are specified by) Subsystem interface:

- Specifies interaction and information flow from/to subsystem boundaries, but not inside the subsystem.
 - Should be clear, well-defined and small.
 - Often called API: Application programmer's interface (but this term should be used during implementation, not during Subsystem Design)
-

Choosing Subsystem

Criteria for subsystem selection:

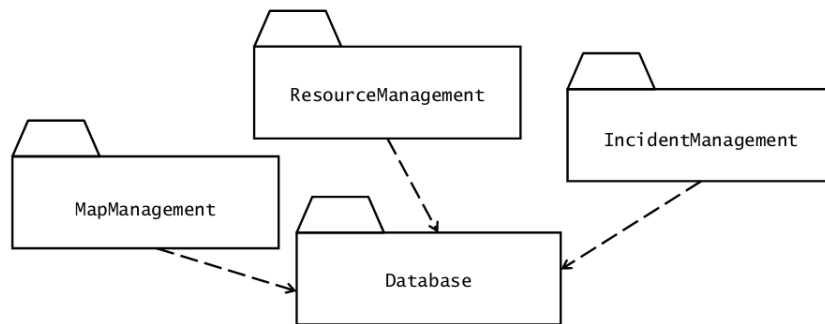
Most of the interaction should be within subsystems, rather than across subsystem boundaries

(High cohesion and Low Coupling).

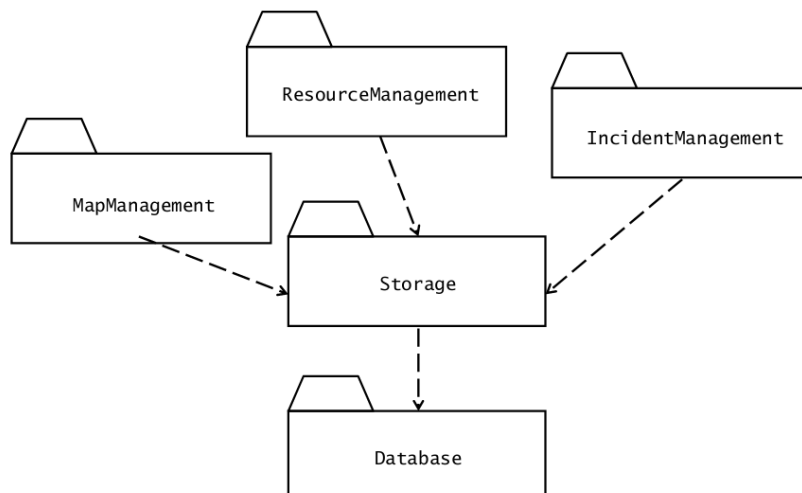
- Does one subsystem always call the other for the service?
 - Which of the subsystems call each other for the service?
-

Example design alternatives: FRIEND

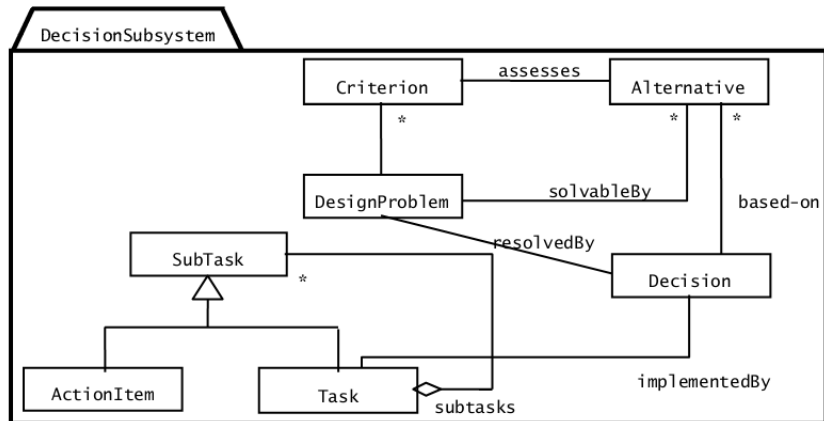
Alternative 1: Direct access to the Database subsystem



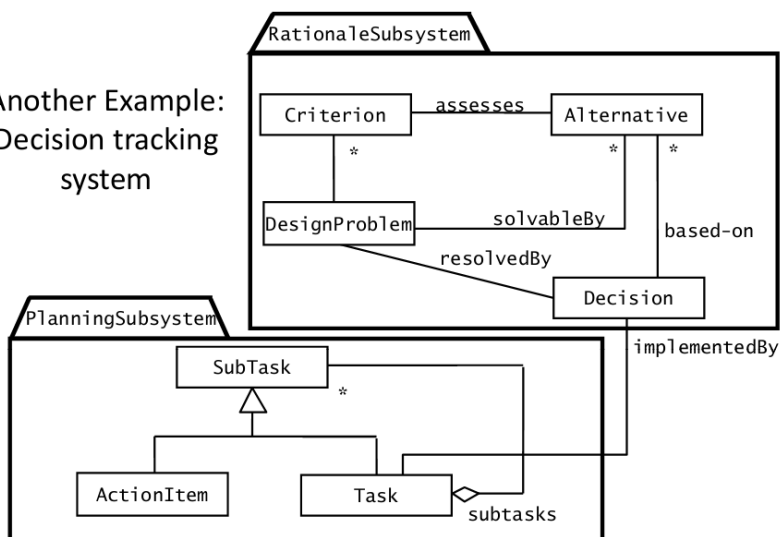
Example design alternatives: FRIEND



Another Example: Decision tracking system



Another Example:
Decision tracking
system



Choosing Subsystems

Primary Question:

What kind of service is provided by the subsystems (subsystem interface) ?

Secondary Question:

Organization: Can the subsystems be hierarchically ordered (**layers**), and /or **partitioned**?

Deployment: How the subsystem is going to be deployed?

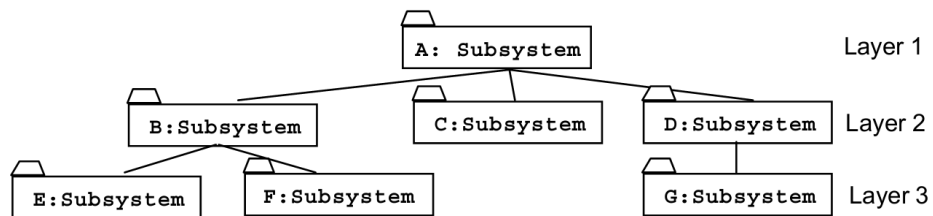
Q. What kind of model is good for describing **layers** and **partitions**?

Subsystem Decomposition: Partitions and Layers

A large system is usually decomposed into subsystems using both, **layers** and **partitions**

- A **Partition** divide a system into several independent (or weakly-coupled) subsystems that provide services on the same level of abstraction.
 - A **layer** is a subsystem that provides subsystem services to a higher layers (level of abstraction)
 - A layer can only depend on lower layers
 - A layer has no knowledge of higher layers
-

Subsystem Decomposition: Partitions and Layers



Subsystem Decomposition Heuristics:

- No more than 7 ± 2 subsystems

More subsystems increase cohesion but also complexity (more services)

- No more than 4 ± 2 layers, use 3 layers (good)

Relationship between Subsystems

Partition relationship

- The subsystem have mutual but not deep knowledge about each other
- Partition A “Calls” partition B and partition B “Calls” partition A

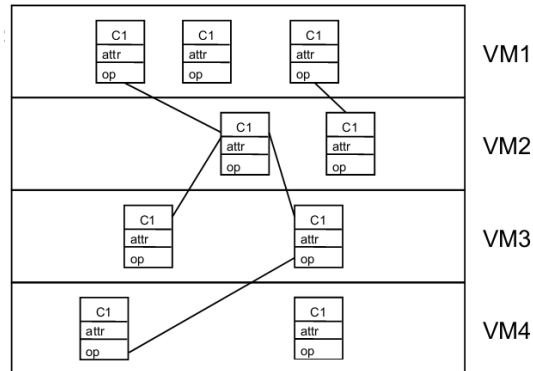
Layer relationship

- Layer A “Calls” Layer B (runtime)
- Layer A “Depends on” Layer B (“make” dependency, compile time)

Closed Architecture (Opaque Layering)

Any layer can only invoke operations from the immediate layer below

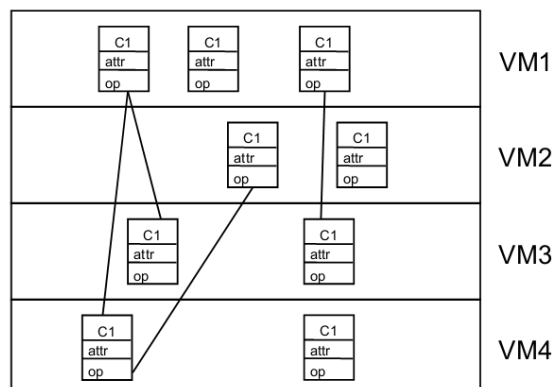
Design Goal: High maintainability, flexibility



Open Architecture (Transparent Layering)

Any layer can invoke operations from any layers below

Design Goal: Runtime, efficiency



Software Architectural Styles

Subsystem decomposition

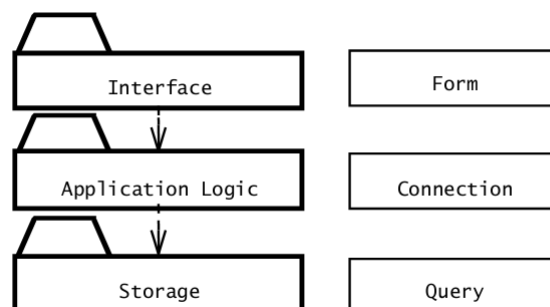
- Identification of **subsystems**, **services**, and **their relationship** to each other.

Specification of the system decomposition is critical.

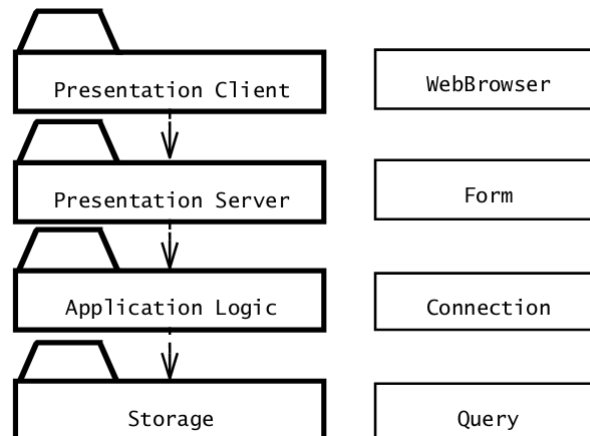
Organization of services (Patterns for software architecture)

- Client/Server
- Peer-To-Peer
- Repository
- Model/View/Controller
- Pipes and Filters
- ...and others

C/S: Three-tier architectural style



Four-tier architectural style



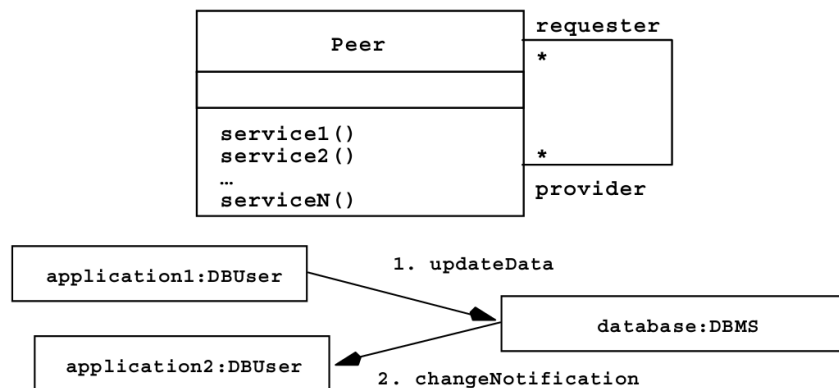
Issues with C/S

Layered systems (in dedicated client-server mode) do not provide peer-to-peer communication

Example: Database receives queries from application but also sends notifications to application when data have changed

Peer-to-Peer Architectural Style

Generalization of Client/Server Architecture
 Clients can be servers and servers can be clients
 More difficult because of possibility of deadlocks



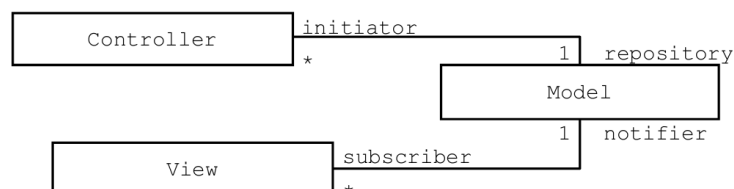
Model/View/Controller

Subsystems are classified into 3 different types

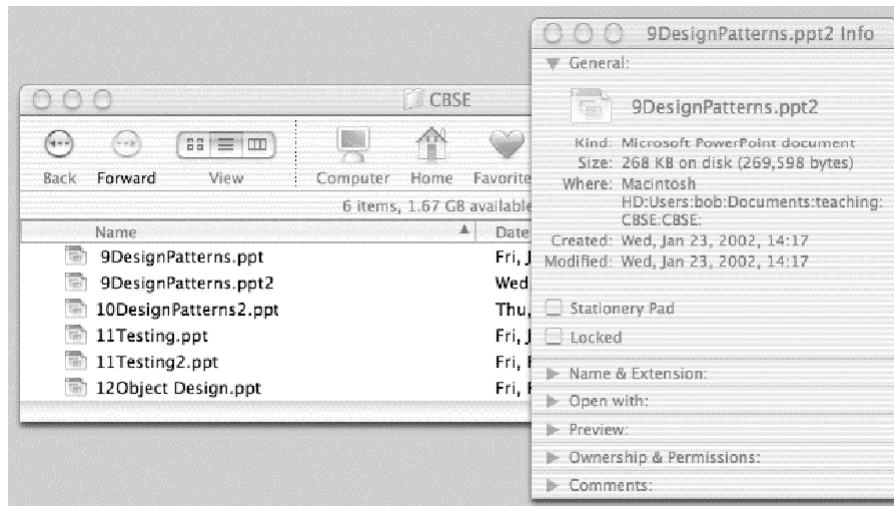
- Model subsystem: Responsible for application domain knowledge
- View subsystem: Responsible for displaying application domain objects to the user
- Controller subsystem: Responsible for sequence of interactions with the user and notifying views of changes in the model.

MVC is a special case of a repository architecture:

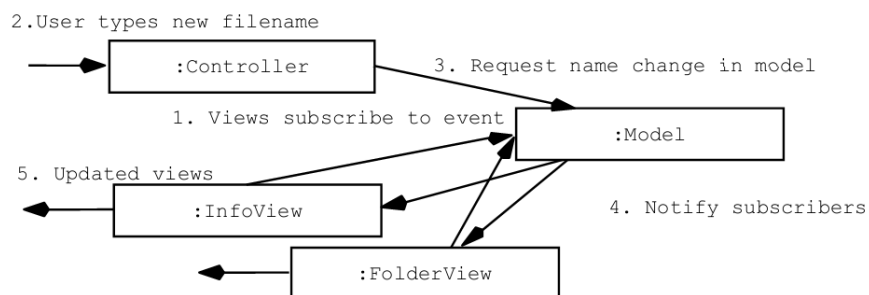
- Model subsystem implements the central data structure, the Controller subsystem explicitly dictate the control flow



Example: MVC Architectural Style



Sequence of Events (Collaborations)





Summary

Design is the process of adding details to the requirements analysis and making implementation decisions

- An evolutionary activity
- Consists of
 - Sub-system Design (Choosing an Architecture)
 - Interface Design
 - Object Design (Solution domain)

Next Lectures...
Object Design
Specifying Interfaces
Package diagram
Design Patterns
