System Design: Decomposing the System
“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.”

- C.A.R. Hoare
System Design

- **Analysis**: Focuses on the application domain

- **Design**: Focuses on the solution domain

- Identify design goals
  - Identify and prioritize qualities of system that should be optimized.
  - **Design the initial subsystem decomposition**
    - Decompose the system into smaller parts based on use case and analysis model
    - Standard architectural styles as starting point
  - **Refine the subsystem decomposition**
    - Refine until all goals satisfied
System Design: Purpose

● Bridging the gap between desired and existing system in a manageable way

● Use Divide and Conquer
  – Model the new system to be developed as a set of subsystems
Construction of the System
  – System Design
  – Object Design and
  – Implementation

System Design
  – First Step: Decomposing the system into manageable parts

Example: Floor Plan
  – Residential House
  – Floor Plan: Location of Walls, doors and windows
  – Functional Requirements:
    ✅ The kitchen in the vicinity of dining room and garage.
    ✅ The bathroom should be close to the bedrooms
  – Standard Utilization
    ✅ Dimensions of the room and location of the doors

Constraints:
  ✅ This house should have 2 bedrooms, a kitchen, and living room
  ✅ The overall distance the occupants walk should be minimized
  ✅ The use of daylight should be maximized
Example of iterative floor plan design. Three successive versions show how we minimize walking distance and take advantage of sunlight.
## Mapping of Architectural and Software Engineering Concepts

<table>
<thead>
<tr>
<th></th>
<th>Architectural Concept</th>
<th>Software Engineering Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Components</strong></td>
<td>Rooms</td>
<td>Subsystems</td>
</tr>
<tr>
<td><strong>Interfaces</strong></td>
<td>Doors</td>
<td>Services</td>
</tr>
<tr>
<td><strong>Nonfunctional</strong></td>
<td>Living area</td>
<td>Response time</td>
</tr>
<tr>
<td>requirements</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Functional</strong></td>
<td>Residential house</td>
<td>Use case</td>
</tr>
<tr>
<td>requirements</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Costly rework</strong></td>
<td>Moving walls</td>
<td>Change of subsystem interfaces</td>
</tr>
</tbody>
</table>
An Overview of System Design

- **Analysis**
  - A set of non functional requirements and constraints
  - A use case model
  - An object model
  - A sequence diagrams

- **System Design**
Activities of System Design

- Analysis
  - nonfunctional requirements
  - dynamic model
  - analysis object model

- System design
  - design goals
  - subsystem decomposition

- Object design
  - object design model
An Overview of System Design

System Design

1. Design Goals
2. System Decomposition
3. Concurrency
4. Hardware/Software Mapping
5. Data Management
6. Global Resource Handling
7. Software Control
8. Boundary Conditions
An Overview of System Design

Requirements Analysis to System Design

- Nonfunctional requirements =>
  - Activity 1: Design Goals Definition
- Functional model =>
  - Activity 2: System decomposition (Selection of subsystems based on functional requirements, cohesion, and coupling)
- Object model =>
  - Activity 4: Hardware/software mapping
  - Activity 5: Persistent data management
- Dynamic model =>
  - Activity 3: Concurrency
  - Activity 6: Global resource handling
  - Activity 7: Software control
- Subsystem Decomposition
  - Activity 8: Boundary conditions
Subsystem Decomposition and their Properties

Subsystem (UML: Package)
- Collection of classes, associations, operations, events and constraints that are interrelated
- Seed for subsystems: UML Objects and Classes.
Subsystem Decomposition Example

- Subsystem decomposition for an Accident Management System
  - Subsystems: UML packages, dashed arrows dependency
Choosing Subsystems

- **Criteria for subsystem selection**: Most of the interaction should be within subsystems, rather than across subsystem boundaries *(High cohesion).*
  - Does one subsystem always call the other for the service?
  - Which of the subsystems call each other for service?

- **Primary Question**:
  - What kind of service is provided by the subsystems (subsystem interface)?

- **Secondary Question**:
  - Can the subsystems be hierarchically ordered (layers)?

- What kind of model is good for describing layers and partitions?
Coupling and Cohesion

- **Goal:** Reduction of *complexity while change occurs*
- **Cohesion** measures the dependence among classes
  - High cohesion: The classes in the subsystem perform similar tasks and are related to each other (via associations)
  - Low cohesion: Lots of miscellaneous and auxiliary classes, no associations
- **Coupling** measures dependencies between subsystems
  - High coupling: Changes to one subsystem will have high impact on the other subsystem (change of model, massive recompilation, etc.)
  - Low coupling: A change in one subsystem does not affect any other subsystem
- **Subsystems:** maximum cohesion and minimum coupling:
  - How can we achieve high cohesion?
  - How can we achieve loose coupling?
Example of reducing the couple of subsystems

Alternative 1: Direct access to the Database subsystem
Alternative 2: Indirect access to the Database through a Storage subsystem
Layering and Partitioning: techniques to achieve low coupling.
– Large system decomposition into subsystems; layers and partitions.

Layer: 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

Partition: Divide a system into several independent (or weakly-coupled) subsystems that provide services on the same level of abstraction.
Subsystem Decomposition into 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)
Relationships between Subsystems

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

- **Partition relationship**
  - The subsystems have mutual but not deep knowledge about each other
  - Partition A “Calls” partition B and partition B “Calls” partition A
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**
Properties of Layered Systems

- Layered systems are *hierarchical*. They are desirable because hierarchy reduces complexity (by low coupling).
- Closed architectures are more portable.
- Open architectures are more efficient.
- If a subsystem is a layer, it is often called a virtual machine.
Software Architecture

- **Software architecture** is process of designing the global organization of a software system, including:
  - Dividing software into subsystems.
  - Deciding how these will interact.
  - Determining their interfaces.

  The architecture is the core of the design, so all software engineers need to understand it.

  The architecture will often constrain the overall efficiency, reusability and maintainability of the system.
Common elements

- Architecture defines major components.
- Architecture defines component relationships (structures) and interactions.
- Architecture omits content information about components that does not pertain to their interactions.
- Behavior of components is a part of architecture insofar as it can be discerned from the point of view of another component.
- Every system has an architecture (even a system composed of one component).
- Architecture defines the rationale behind the components and the structure.
- Architecture definitions do not define what a component is.
- Architecture is not a single structure -- no single structure is the architecture.
Architecture established establishes the context for design and implementation

- Architectural decisions are the most fundamental decisions; changing them will have significant ripple effects.
Architecture defined

- IEEE 1471-2000
  - Software architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution

- Software architecture encompasses the set of significant decisions about the organization of a software system
  - Selection of the structural elements and their interfaces by which a system is composed
  - Behavior as specified in collaborations among those elements
  - Composition of these structural and behavioral elements into larger subsystems
  - Architectural style that guides this organization
Architecture defined

- Software architecture also involves
  - Functionality
  - Usability
  - Resilience
  - Performance
  - Reuse
  - Comprehensibility
  - Economic and technology constraints and tradeoffs
  - Aesthetic concerns
Architectural style defined

- Style is the classification of a system’s architecture according to those with similar patterns

- A pattern is a common solution to a common problem; patterns may be classified as idioms, mechanisms, or frameworks
Software Architectural Styles

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

- Specification of the system decomposition is critical.

- Patterns for software architecture
  - Client/Server
  - Peer-To-Peer
  - Repository
  - Model/View/Controller
  - Pipes and Filters
Client/Server Architectural Style

- One or many servers provides services to instances of subsystems, called clients.
- Client calls on the server, which performs some service and returns the result
  - Client knows the *interface* of the server *(its service)*
  - Server does not need to know the interface of the client
- Response in general immediately
- Users interact only with the client
Client/Server Architectural Style

- Often used in database systems:
  - Front-end: User application (client)
  - Back end: Database access and manipulation (server)

- Functions performed by client:
  - Customized user interface
  - Front-end processing of data
  - Initiation of server remote procedure calls
  - Access to database server across the network

- Functions performed by the database server:
  - Centralized data management
  - Data integrity and database consistency
  - Database security
  - Concurrent operations (multiple user access)
  - Centralized processing (for example archiving)
Design Goals for Client/Server Systems

- **Service Portability**
  - Server can be installed on a variety of machines and operating systems and functions in a variety of networking environments

- **Transparency, Location-Transparency**
  - The server might itself be distributed (why?), but should provide a single "logical" service to the user

- **Performance**
  - Client should be customized for interactive display-intensive tasks
  - Server should provide CPU-intensive operations

- **Scalability**
  - Server should have spare capacity to handle larger number of clients

- **Flexibility**
  - The system should be usable for a variety of user interfaces and end devices (eg. WAP Handy, wearable computer, desktop)

- **Reliability**
  - System should survive node or communication link problems
Problems with Client/Server Architectural Styles

- Layered systems do not provide peer-to-peer communication

- Peer-to-peer communication is often needed

- 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

![Diagram showing Peer-to-Peer architecture with service requests and database interactions]
Peer-to-Peer Architectural Style
Example of a Peer-to-Peer Architectural Style

- ISO’s OSI Reference Model
  - ISO = International Standard Organization
  - OSI = Open System Interconnection

- Reference model defines 7 layers of network protocols and strict methods of communication between the layers.

- Closed software architecture
OSI model Packages and their Responsibility

- The **Physical** layer represents the hardware interface to the network. It allows to `send()` and `receive bits` over a `channel`.
- The **Datalink** layer allows to send and receive `frames` without error using the services from the Physical layer.
- The **Network** layer is responsible for that the data are reliably transmitted and routed within a network.
- The **Transport** layer is responsible for reliably transmitting from end to end. (This is the interface seen by Unix programmers when transmitting over TCP/IP sockets)
- The **Session** layer is responsible for initializing a connection, including authentication.
- The **Presentation** layer performs data transformation services, such as byte swapping and encryption
- The **Application** layer is the system you are designing (unless you build a protocol stack). The application layer is often layered itself.
Another View at the ISO Model

• A closed software architecture
• Each layer is a UML package containing a set of objects
Middleware Allows Focus On The Application Layer

Diagram:
- Application
- Presentation
- Session
- Transport
- Network
- DataLink
- Physical
- CORBA
- TCP/IP
- Ethernet
- Object
- Socket
- Wire
Three-tier architectural style.

- Interface layer: boundary objects
- Application logics: control an entity objects
- Storage layer; realizes the query, retrieval
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 datastructure, the Controller subsystem explicitly dictate the control flow
Example of a File System Based on the MVC Architectural Style
MVC

- Previous slide displays two views of the file system
  - CBSE, information about file

- Sequence of Events
  - The **InfoView** and the **FolderView** both subscribe for changes to the **File** models they display
  - The user types the new name of the file
  - The **Controller**, the object responsible for interacting with the user during file name changes, sends a request to the **Model**
  - The **Model** changes the file name and notifies all subscribers of the change
  - Both **InfoView** and **FolderView** are updated, so the user sees a consistent change
2. User types new filename

3. Request name change in model

1. Views subscribe to event

4. Notify subscribers

5. Update views

Sequence of Events
Repository Architectural Style

- Subsystems access and modify data from a single data structure

- Subsystems are loosely coupled (interact only through the repository)

- Control flow is dictated by central repository (triggers) or by the subsystems (locks, synchronization primitives)
Examples of Repository Architectural Style

- Modern Compilers: generates ParseTree and a SymbolTable
- SourceLevelDebugger: uses ParseTree & SymbolTable
System Design Activities: From Objects to Subsystems

- Goal: to convert the *analysis model* into a *design model* that takes into account the non-functional requirements.

- Steps (starting from an analysis model):
  - Identify the design goals
  - Design an initial system decomposition

- Use *MyTrip* - a route planning system as an example
Example: MyTrip Planner

- Problem Statement:
  - Using MyTrip, a driver can plan a trip from a home computer by contacting a trip-planner service on the Web. The trip is saved for later retrieval on the server. The trip-planning service must support more than one driver - PlanTrip use case
  - The driver then goes to the car and starts the trip, while the onboard computer gives directions based on trip information from the planning service and her current position indicated by an onboard GPS system - ExecuteTrip use case
PlanTrip Use Case

- **Use Case Name:** Plan Trip

- **Flow of Events:**
  1. The Driver activates her computer and logs into the trip-planning Web service
  2. The Driver enters constraints for the trip as a sequence of destinations
  3. Based on a database of maps, the planning service computes the shortest way of visiting the destinations in the ordered specified. The result is a sequence of segments binding a series of crossings and a list of directions
  4. The Driver can revise the trip by adding or removing destinations
  5. The Driver saves the planned trip by name in the planning service database for later retrieval
Use Case Name: ExecuteTrip

Flow of Events:
1. The Driver starts her car and logs into the onboard route assistant
2. Upon successful login, the Driver specifies the planning service and the name of the trip to be executed
3. The onboard route assistant obtains the list of destinations, directions, segments, and crossings from the planning service.
4. Given the current position, the route assistant provides the Driver with the next step of directions.
5. The Driver arrives at the destinations and shuts down the route assistant.
Crossing: A geographical point where several Segments meet.

Destination: A Destination represents a Location where the driver wishes to go.

Direction: Given a Crossing and an adjacent Segment, a Direction describes in natural language how to steer the car onto the given Segment.

Location: The position of a car as known by the onboard GPS system.

PlanningService: Web server that can supply a trip, linking a number of destinations in the form of a sequence of Crossings and Segments.

RouteAssistant: A RouteAssistant gives Directions to the driver, given the current Location and upcoming Crossing.

Segment: Represents the road between two Crossings.

Trip: Sequence of Directions between two Destinations.
Identifying Design Goals

- Definition of design goals is first step of system design -
  - Identifies qualities that should be focused on.

- Design goals: Non-functional requirements
Performance Criteria

- Speed and space requirements
  - **Response time:** How soon is a user’s request acknowledged?
  - **Throughput:** How many tasks can the system handle?
  - **Memory:** How much space is required for the system to run?
Dependability Criteria

- Determine how much effort to minimizing system crashes and their consequences:
  - **Robustness**: Ability to survive invalid user input
  - **Reliability**: Difference between specified and observed behavior
  - **Availability**: Percentage of time that the system can use for normal tasks
  - **Fault tolerance**: Ability to operate under erroneous conditions
  - **Security**: Ability to withstand malicious tasks
  - **Safety**: Ability to avoid endangering human lives
Cost Criteria

- Include the cost to develop the system, to deploy it and to administer it - includes design considerations as well as managerial considerations.
  
  - **Development cost**: Cost of developing the initial system
  
  - **Deployment cost**: Cost of installing the system and training the users
  
  - **Upgrade cost**: Cost of translating the data from previous system - backward compatibility
  
  - **Maintenance cost**: Cost required for bug fixes
  
  - **Administration cost**: Cost to administer the system
**Maintenance Criteria**

- Determine how difficult it is to change the system after deployment

- **Extensibility:** How easy is it to add functionality or new classes?

- **Modifiability:** How easy is it to change the functionality of the system?

- **Adaptibility:** How easy is it to port the system to different application domains?

- **Portability:** How easy is it to port it to different platforms?

- **Readability:** How easy is to understand the system?

- **Traceability of Requirements:** How easy is it to map the code to specific requirements?
End User Criteria

- Include qualities that are desirable from a users' point of view but have not yet been covered under performance and dependability criteria
  
  - **Utility**: How well does the system support the work of the user
  - **Usability**: How easy is it for the user to use the system
Design Tradeoffs

- Only a small subset of these criteria can be simultaneously taken into account
  - Cannot develop software that is safe, secure and cheap!

- Need to prioritize and tradeoff design goals against each other
## Example Tradeoffs

<table>
<thead>
<tr>
<th>Tradeoff</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space vs Speed</strong></td>
<td>If software does not meet response time or throughput requirements, more memory can be expended to speed up the software. Or if space is an issue then data can be compressed at the cost of speed.</td>
</tr>
<tr>
<td><strong>Delivery time vs functionality</strong></td>
<td>If development runs behind schedule, a manager can choose to deliver less functionality on time or full functionality at a later time.</td>
</tr>
<tr>
<td><strong>Delivery time vs Quality</strong></td>
<td>If testing runs behind schedule, manager can deliver software with known bugs on time, or deliver software later with fewer bugs.</td>
</tr>
<tr>
<td><strong>Delivery time vs Staffing</strong></td>
<td>If development runs behind schedule the manager can add more resources to the project to increase project productivity - option available only early in the project cycle!</td>
</tr>
</tbody>
</table>
Nonfunctional Requirements

- (Specified by Client during requirement elicitation)
  - *MyTrip* is in contact with the planning service via a wireless modem. Assume that the wireless modem functions properly at the initial destination
  - Once the trip has been started, *MyTrip* should give correct directions even if the modem fails to maintain a connection with the *PlanningService*.
  - *MyTrip* should minimize connection costs to reduce operating costs
  - Replanning is possible only if the connection to *PlanningService* is possible,
  - The *PlanningService* can support at least 50 different drivers and 1,000 trips.
Design Goals for MyTrip

- Design goals are identified by the non-functional requirements. What are the design goals for the MyTrip system?

  - **Reliability:** MyTrip should be reliable (NF: 2)
  - **Fault Tolerance:** MyTrip should be fault tolerant to loss of connectivity with the routing service (NF: 2)
  - **Security:** MyTrip should be secure, not allow other drivers to access a driver’s trip (deduced)
  - **Modifiability:** MyTrip should be modifiable to use different routing services (anticipation of change)
Identifying Subsystems

- A similar process to Abbot’s analysis for objects - heuristics are often applied for grouping objects

- Is typically iterative - revised as new issues are discovered or as new functionality is added
  - Initial changes can be drastic - best handled through brainstorming sessions

- Initial subsystem is derived
  - From the functional requirements
  - And by grouping together functionally related objects
Heuristics for Grouping Objects into Subsystems

- Assign objects identified in one use case to the same subsystem

- Create a dedicated subsystem for objects used for moving data between subsystems

- Minimize the number of associations crossing the subsystem boundaries

- All objects in the same subsystem should be functionally related.
PlanTrip Use Case

• Use Case Name: Plan Trip

• Flow of Events:
  1. The Driver activates her computer and logs into the trip-planning Web service
  2. The Driver enters constraints for the trip as a sequence of destinations
  3. Based on a database of maps, the planning service computes the shortest way of visiting the destinations in the ordered specified. The result is a sequence of segments binding a series of crossings and a list of directions
  4. The Driver can revise the trip by adding or removing destinations
  5. The Driver saves the planned trip by name in the planning service database for later retrieval
ExecuteTrip Use Case

- Use Case Name: ExecuteTrip

- Flow of Events:
  1. The Driver starts her car and logs into the onboard route assistant
  2. Upon successful login, the Driver specifies the planning service and the name of the trip to be executed
  3. The onboard route assistant obtains the list of destinations, directions, segments, and crossings from the planning service.
  4. Given the current position, the route assistant provides the Driver with the next step of directions.
  5. The Driver arrives at the destinations and shuts down the route assistant.
MyTrip Subsystems

Planning Subsystem: The Planning Subsystem is responsible for a Trip connecting a sequence of Destinations. The Planning Subsystem is also responsible for responding to replan requests from Routing Subsystems.

Routing Subsystem: The Routing Subsystem is responsible for downloading a Trip from the PlanningService and executing it by giving it to the driver based on location.
Encapsulating Subsystems with the Façade Design Pattern

- Subsystem decomposition
  - reduces complexity of the solution domain by minimizing coupling among subsystems

- Façade design pattern
  - Allows further reduction of dependencies between classes by encapsulating a subsystem with a simple, unified interface.
Façade - Motivation

client classes

subsystem classes

Facade

69
Consequences of Façade Pattern

- **Consequences**
  - **Benefits**
    - It hides the implementation of the subsystem from clients, making the subsystem easier to use.
    - It promotes weak coupling between the subsystem and its clients. This allows you to change the classes that comprise the subsystem without affecting the clients.
    - It reduces compilation dependencies in large software systems.
    - It simplifies porting systems to other platforms, because it's less likely that building one subsystem requires building all others.
    - It does not prevent sophisticated clients from accessing the underlying classes.
  - **Liabilities**
    - It does not prevent clients from accessing the underlying classes!
Façade Pattern - Example
Summary

- **System Design**
  - Reduces the gap between requirements and the (virtual) machine
  - Decomposes the overall system into manageable parts

- **Design Goals Definition**
  - Describes and prioritizes the qualities that are important for the system
  - Defines the value system against which options are evaluated

- **Subsystem Decomposition**
  - Results into a set of loosely dependent parts which make up the system