System Design: Decomposing the System

The Scope of System Design

- **Problem**
  - Bridge the gap between a problem and an existing system in a manageable way
  - How?
  - Use Divide & Conquer:
    1) Identify design goals
    2) Model the new system design as a set of subsystems
    3-8) Address the major design goals.

System Design: Eight Issues

1. Identify Design Goals
   - Additional NEFs
   - Trade-offs

2. Subsystem Decomposition
   - Layers vs Partitions
   - Coherence & Coupling

3. Identify Concurrency
   - Identification of Parallelism
   - (Processes, Threads)

4. Hardware/Software Mapping
   - Identification of Nodes
   - Special Purpose Systems
   - Buy vs Build
   - Network Connectivity

5. Persistent Data Management
   - Storing Persistent Objects
   - Filesystem vs Database

6. Global Resource Handling
   - Access Control
   - ACL vs Capabilities
   - Security

7. Software Control
   - Monolithic
   - Event-Driven
   - Concurrent Processes

8. Boundary Conditions
   - Initialization
   - Termination
   - Failure

Analysis Sources: Requirements and System Model

- **Nonfunctional Requirements**
- **Functional Model**
- **Dynamic Model**
- **Object Model**

How the Analysis Models influence System Design

- **Nonfunctional Requirements**
  - => Definition of Design Goals
  - => Subsystem Decomposition
  - => Object model
  - => Hardware/Software Mapping, Persistent Data Management
  - => Dynamic model
- **Dynamic model**
  - => Identification of Concurrency, Global Resource Handling, Software Control
- **Finally:** Hardware/Software Mapping
  - => Boundary conditions

From Analysis to System Design

- **Nonfunctional Requirements**
  - 1. Design Goals
    - Definition
    - Trade-offs
  - 2. System Decomposition
    - Layers vs Partitions
    - Coherence/Coupling
  - 3. Concurrency
    - Identification of Threads
  - 4. Hardware/Software Mapping
    - Special Purpose Systems
    - Buy vs Build
    - Allocation of Resources
    - Connectivity
  - 5. Data Management
    - Persistent Objects
    - Filesystem vs Database
  - 6. Global Resource Handling
    - Access Control
    - ACL vs Capabilities
    - Security
Example 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 number of errors
- Readability
- Ease of learning
- Ease of remembering
- Ease of use
- Increased productivity
- Low-cost
- Flexibility

Stakeholders have different Design Goals

- Low cost
- Increased productivity
- Backward compatibility
- Traceability of requirements
- Rapid development
- Flexibility
- Reliability
- Good documentation
- Portability
- Usability
- Low cost of errors
- Modifiability
- Readability
- Reusability
- Adaptability
- Well-defined interfaces

Typical Design Trade-offs

- Functionality v. Usability
- Cost v. Robustness
- Efficiency v. Portability
- Rapid development v. Functionality
- Cost v. Reusability
- Backward Compatibility v. Readability

Subsystem Decomposition

- **Subsystem**
  - Collection of classes, associations, operations, events and constraints that are closely interrelated with each other
  - The objects and classes from the object model are the “seeds” for the subsystems
  - In UML subsystems are modeled as packages
  - Services are defined during system design.

- **Service**
  - A set of named operations that share a common purpose
  - The origin (“seed”) for services are the use cases from the functional model

Subsystem Interfaces vs API

- **Subsystem interface**: Set of fully typed UML operations
  - Specifies the interaction and information flow from and to subsystem boundaries, but not inside the subsystem
  - Refinement of service, should be well-defined and small
  - **Subsystem interfaces are defined during object design**

- **Application programmer’s interface (API)**
  - The API is the specification of the subsystem interface in a specific programming language
  - APIs are defined during implementation

  - The terms subsystem interface and API are often confused with each other
  - The term API should not be used during system design and object design, but only during implementation.
Example: Notification subsystem

- **Service provided by Notification Subsystem**
  - LookupChannel()
  - SubscribeToChannel()
  - SendNotice()
  - UnsubscribeFromChannel()

- **Subsystem Interface of Notification Subsystem**
  - Set of fully typed UML operations
  - Left as an Exercise

- **API of Notification Subsystem**
  - Implementation in Java
  - Left as an Exercise.

Subsystem Interface Object

- Good design: The subsystem interface object describes all the services of the subsystem interface

- **Subsystem Interface Object**
  - The set of public operations provided by a subsystem

Subsystem Interface Objects can be realized with the façade pattern (=> lecture on design patterns).

Properties of Subsystems: Layers and Partitions

- A **layer** is a subsystem that provides a service to another subsystem with the following restrictions:
  - A layer only depends on services from lower layers
  - A layer has no knowledge of higher layers
  - A layer can be divided horizontally into several independent subsystems called **partitions**
    - Partitions provide services to other partitions on the same layer
    - Partitions are also called “weakly coupled” subsystems.

Relationships between Subsystems

- Two major types of Layer relationships
  - Layer A "depends on" Layer B (compile time dependency)
    - Example: Build dependencies (make, ant, maven)
  - Layer A "calls" Layer B (runtime dependency)
    - Example: A web browser calls a web server
    - Can the client and server layers run on the same machine?
      - Yes, they are layers, not processor nodes
      - Mapping of layers to processors is decided during the Software/hardware mapping!

- **Partition relationship**
  - The subsystems have mutual knowledge about each other
    - A calls services in B; B calls services in A (Peer-to-Peer)

  **UML convention:**

Example of a Subsystem Decomposition

- **Partition relationship**
- **Layer Relationship**
  - Layer 1
  - Layer 2
  - Layer 3

ARENA Subsystem Decomposition

- User Interface
- Advertisement
- Tournament
- Component Management
- Session Management
- Tournament Statistics
- User Management
- User Directory
Example of a Bad Subsystem Decomposition

Good Design: The System as set of Interface Objects

Virtual Machine
- A virtual machine is a subsystem connected to higher and lower level virtual machines by "provides services for" associations
- A virtual machine is an abstraction that provides a set of attributes and operations
- The terms layer and virtual machine can be used interchangeably
  - Also sometimes called "level of abstraction".

Building Systems as a Set of Virtual Machines
A system is a hierarchy of virtual machines, each using language primitives offered by the lower machines.

Closed Architecture (Opaque Layering)
- Each virtual machine can only call operations from the layer below
  Design goals: Maintainability, flexibility.

Open Architecture (Transparent Layering)
- Each virtual machine can call operations from any layer below
  Design goal: Runtime efficiency
Properties of Layered Systems

- Layered systems are hierarchical. This is a desirable design, because hierarchy reduces complexity.
- Low coupling
- Closed architectures are more portable.
- Open architectures are more efficient.
- Layered systems often have a chicken-and-egg problem.

Coupling and Coherence of Subsystems

- Goal: Reduce system complexity while allowing change.
- Coherence measures dependency among classes.
  - High coherence: The classes in the subsystem perform similar tasks and are related to each other via many associations.
  - Low coherence: Lots of miscellaneous and auxiliary classes, almost no associations.
- Coupling measures dependency among subsystems.
  - High coupling: Changes to one subsystem will have high impact on the other subsystem.
  - Low coupling: A change in one subsystem does not affect any other subsystem.

How to achieve high Coherence

- High coherence can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries.
- Questions to ask:
  - Does one subsystem always call another one for a specific service?
  - Yes: Consider moving them together into the same subsystem.
  - Which of the subsystems call each other for services?
  - Can this be avoided by restructuring the subsystems or changing the subsystem interface?
  - Can the subsystems even be hierarchically ordered (in layers)?

How to achieve Low Coupling

- Low coupling can be achieved if a calling class does not need to know anything about the internals of the called class (Principle of information hiding, Parnas).
- Questions to ask:
  - Does the calling class really have to know any attributes of classes in the lower layers?
  - Is it possible that the calling class calls only operations of the lower level classes?

Architectural Style vs Architecture

- Subsystem decomposition: Identification of subsystems, services, and their association to each other (hierarchical, peer-to-peer, etc).
- Architectural Style: A pattern for a subsystem decomposition.
- Software Architecture: Instance of an architectural style.

Examples of Architectural Styles

- Client/Server
- Peer-To-Peer
- Repository
- Model/View/Controller
- Three-tier, Four-tier Architecture
- Service-Oriented Architecture (SOA)
- Pipes and Filters
Client/Server Architectural Style

- One or more servers provide services to instances of subsystems, called clients.
- Each client calls on the server, which performs some service and returns the result.
- The clients know the interface of the server.
- The response in general is immediate.
- End users interact only with the client.

Client

requester

provider

+service1()
+service2()
+serviceN()

Server

Client/Server Architectures

- Often used in the design of database systems.
  - Front-end: User application (client).
  - Back-end: Database access and manipulation (server).
- Functions performed by client:
  - Input from the user (customized user interface).
  - Front-end processing of input data.
- Functions performed by the database server:
  - Centralized data management.
  - Data integrity and database consistency.
  - Database security.

Design Goals for Client/Server Architectures

**Service Portability**
- Server runs on many operating systems and many networking environments.
- Server might itself be distributed, but provides a single "logical" service to the user.

**Location-Transparency**
- Client optimized for interactive display-intensive tasks; Server optimized for CPU-intensive operations.

**High Performance**
- Server can handle large # of clients.

**Scalability**
- Server should be able to survive client and communication problems.

**Reliability**
- A measure of success with which the observed behavior of a system conforms to the specification of its behavior (Chapter 11: Testing).

Problems with Client/Server Architectures

- Client/Server systems do not provide peer-to-peer communication.
- Peer-to-peer communication is often needed.
- Example:
  - Database must process queries from application and should be able to send notifications to the application when data have changed.

Relationship Client/Server & Peer-to-Peer

**Problem statement**: "Clients can be servers and servers can be clients."
**Which model is correct?**
- **Model 1**: "A peer can be either a client or a server".
- **Model 2**: "A peer can be a client as well as a server".

Peer-to-Peer Architectural Style

**Generalization of Client/Server Architectural Style**
"Clients can be servers and servers can be clients."

*Introduction to a new abstraction: Peer*

*"Clients and servers can be both peers."

How do we model this statement? With Inheritance?

**Proposal 1**: "A peer can be either a client or a server".

**Proposal 2**: "A peer can be a client as well as a server."

ISO's OSI Reference Model
- ISO = International Standard Organization
- OSI = Open System Interconnection
- Reference model which defines 7 layers and communication protocols between the layers

OSI Model Layers and Services
- The Application layer is the system you are building (unless you build a protocol stack)
  - The application layer is usually layered itself
- The Presentation layer performs data transformation services, such as byte swapping and encryption
- The Session layer is responsible for initializing a connection, including authentication

OSI Model Layers and their Services
- The Transport layer is responsible for reliably transmitting messages
  - Used by Unix programmers who transmit messages over TCP/IP sockets
- The Network layer ensures transmission and routing
  - Services: Transmit and route data within the network
- The Datalink layer models frames
  - Services: Transmit frames without error
- The Physical layer represents the hardware interface to the network

An Object-Oriented View of the OSI Model
- The OSI Model is a closed software architecture (i.e., it uses opaque layering)
- Each layer can be modeled as a UML package containing a set of classes available for the layer above

Middleware Allows Focus On Higher Layers
- Abstraction provided by Middleware

Providing Consistent Views

- **Problem:** In systems with high coupling changes to the user interface (boundary objects) often force changes to the entity objects (data)
  - The user interface cannot be reimplemented without changing the representation of the entity objects
  - The entity objects cannot be reorganized without changing the user interface
- **Solution:** Decoupling! The model-view-controller architectural style decouples data access (entity objects) and data presentation (boundary objects)
  - The Data Presentation subsystem is called the **View**
  - The Data Access subsystem is called the **Model**
  - The **Controller** subsystem mediates between View (data presentation) and Model (data access)
- Often called **MVC**.

Model-View-Controller Architectural Style

- **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

![Class Diagram](image)

Better understanding with a Collaboration Diagram

UML Collaboration Diagram

- A **Collaboration Diagram** is an instance diagram that visualizes the interactions between objects as a flow of messages. Messages can be events or calls to operations
- Communication diagrams describe the static structure as well as the dynamic behavior of a system:
  - The static structure is obtained from the UML class diagram
  - Collaboration diagrams reuse the layout of classes and associations in the class diagram
  - The dynamic behavior is obtained from the dynamic model (UML sequence diagrams and UML statechart diagrams)
  - Messages between objects are labeled with a chronological number and placed near the link the message is sent over
- Reading a collaboration diagram involves starting at message 1.0, and following the messages from object to object

Example: Modeling the Sequence of Events in MVC

![UML Class Diagram](image)

Virtual Machines in 3-Layer Architectural Style

**3-Layer-Architectural Style**

**3-Tier Architecture**

- **Definition: 3-Layer Architectural Style**
  - An architectural style, where an application consists of 3 hierarchically ordered subsystems
  - A user interface, middleware and a database system
  - The middleware subsystem services data requests between the user interface and the database subsystem
- **Definition: 3-Tier Architecture**
  - A software architecture where the 3 layers are allocated on 3 separate hardware nodes
  - **Note:** Layer is a type (e.g. class, subsystem) and Tier is an instance (e.g. object, hardware node)
  - Layer and Tier are often used interchangeably.
Example of a 3-Layer Architectural Style

- Three-Layer architectural style are often used for the development of Websites:
  1. The **Web Browser** implements the user interface
  2. The **Web Server** serves requests from the web browser
  3. The **Database** manages and provides access to the persistent data.

Example of a 4-Layer Architectural Style

- 4-Layer-architectural styles (4-Tier Architectures) are usually used for the development of electronic commerce sites. The layers are:
  1. The **Web Browser**, providing the user interface
  2. A **Web Server**, serving static HTML requests
  3. An **Application Server**, providing session management (for example the contents of an electronic shopping cart) and processing of dynamic HTML requests
  4. A back end **Database**, that manages and provides access to the persistent data
    - In current 4-tier architectures, this is usually a relational Database management system (RDBMS).

MVC vs. 3-Tier Architectural Style

- The **MVC** architectural style is nonhierarchical (triangular):
  - View subsystem sends updates to the Controller subsystem
  - Controller subsystem updates the Model subsystem
  - View subsystem is updated directly from the Model subsystem
- The **3-tier** architectural style is hierarchical (linear):
  - The presentation layer never communicates directly with the data layer (opaque architecture)
  - All communication must pass through the middleware layer

- History:
  - MVC (1970-1980): Originated during the development of modular graphical applications for a single graphical workstation at Xerox Parc

Pipes and Filters

- A pipeline consists of a chain of processing elements (processes, threads, etc.), arranged so that the output of one element is the input to the next element
  - Usually some amount of buffering is provided between consecutive elements
  - The information that flows in these pipelines is often a stream of records, bytes or bits.

Pipes and Filters Architectural Style

- An architectural style that consists of two subsystems called pipes and filters
  - **Filter**: A subsystem that does a processing step
  - **Pipe**: A Pipe is a connection between two processing steps
  - Each filter has an input pipe and an output pipe.
    - The data from the input pipe are processed by the filter and then moved to the output pipe
  - Example of a Pipes-and-Filters architecture: Unix
    - Unix shell commands: ls -a | cat

Additional Readings

- **E.W. Dijkstra (1968)**
  - The structure of the T.H.E Multiprogramming system, Communications of the ACM, 18(8), pp. 453-457
- **D. Parnas (1972)**
  - On the criteria to be used in decomposing systems into modules, CACM, 15(12), pp. 1053-1058
- **L.D. Erman, F. Hayes-Roth (1980)**
- **J.D. Day and H. Zimmermann (1983)**
- **Jostein Gaarder (1991)**
  - Sophie's World: A Novel about the History of Philosophy.
Summary

• **System Design**
  • An activity that reduces the gap between the problem and an existing (virtual) machine

• **Design Goals Definition**
  • Describes the important system qualities
  • Defines the values against which options are evaluated

• **Subsystem Decomposition**
  • Decomposes the overall system into manageable parts by using the principles of cohesion and coherence

• **Architectural Style**
  • A pattern of a typical subsystem decomposition

• **Software architecture**
  • An instance of an architectural style

\[\text{Client-Server, Peer-to-Peer, Model-View}\]