Chapter 6
System Design: Decomposing the System
Design

“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
Why is Design so Difficult?

- **Analysis**: Focuses on the application domain
- **Design**: Focuses on the solution domain
  - Design knowledge is a moving target
  - The reasons for design decisions are changing very rapidly
    - Halftime knowledge in software engineering: About 3-5 years
    - Cost of hardware rapidly sinking
- “Design window”: Time in which design decisions have to be made
The Scope of System Design

- 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 NFRs
   - 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
   - Conc. Processes

8. Boundary Conditions
   - Initialization
   - Termination
   - Failure
How the Analysis Models influence System Design

• Nonfunctional Requirements
  => Definition of Design Goals

• Functional model
  => Subsystem Decomposition

• Object model
  => Hardware/Software Mapping, Persistent Data Management

• Dynamic model
  => Identification of Concurrency, Global Resource Handling, Software Control

• Finally: Subsystem Decomposition
  => Boundary conditions
From Analysis to System Design

Nonfunctional Requirements
1. Design Goals
   Definition
   Trade-offs

Functional Model
2. System Decomposition
   Layers vs Partitions
   Coherence/Coupling

Dynamic Model
3. Concurrency
   Identification of
   Threads

Object Model
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
Handlung
   Access Control List
   vs Capabilities
   Security

7. Software
   Control
   Monolithic
   Event-Driven
   Conc. Processes

8. Boundary
   Conditions
   Initialization
   Termination
   Failure
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

Client (Customer):
- Low cost
- Increased productivity
- Backward compatibility
- Traceability of requirements
- Rapid development
- Flexibility

End User:
- Functionality
- User-friendliness
- Usability
- Ease of learning
- Fault tolerant
- Robustness

Developer/Maintainer:
- Minimum # of errors
- Modifiability, Readability
- Reusability, Adaptability
- Well-defined interfaces

Runtime Efficiency:
- Portability
- Good documentation

Reliability:
- Minimum # 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

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

- **Services are defined during system design**
Example: Services provided by the ARENA Subsystems

- **Tournament**: Manages tournaments, promotions, and applications
- **User Management**: Administers user accounts
- **User Directory**: Stores user profiles (contact info & subscriptions)
- **User Interface**: Advertisement
  - Manages advertisement banners & sponsorships
- **Component Management**: Adds games, styles, and expert rating formulas
- **Session Management**: Maintains state during matches
- **Tournament Statistics**: Stores results of archived tournaments

**Services are described by subsystem interfaces**
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

- **API** of Notification Subsystem
  - Implementation in Java
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
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
  • Layer A “calls” Layer B (runtime dependency)
    • Example: A web browser calls a web server

• 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
  • Runtime dependencies are associations with dashed lines
  • Compile time dependencies are associations with solid lines.
Example of a Subsystem Decomposition

Layer 1

Layer 2

Layer 3

Partition relationship

Layer Relationship “depends on”

Layer Relationship “calls”
Example of a Bad Subsystem Decomposition
Good Design: The System as set of Interface Objects

Subsystem Interface Objects

- User Interface
- User Management
- Advertisement
- Tournament Statistics
- Tournament
- Component Management
- Session Management
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.
Opaque Layering in ARENA

- Interface
  - ArenaServer
  - ArenaClient
- Application Logic
  - UserManagement
  - GameManagement
  - AdvertisementManagement
  - TournamentManagement
  - Notification
- Storage
  - ArenaStorage
- Application Logic
- Interface
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
- Closed architectures are more portable
- Open architectures are more efficient
- Layered systems often have a chicken and egg problem

How do you open the symbol table when you are debugging the File System?
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.
Coupling and Coherence of Subsystems

**Good Design**

- **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 associations
  - **Low coherence**: Lots of miscellaneous and auxiliary classes, 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
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 many **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 server does not need to know the interface of the client
- The response in general is immediate
- End users interact only with the client

```
*                *  
| requester      | provider |

+service1()     
+service2()     
+serviceName()  
```
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

**Location-Transparency**
Server might itself be distributed, but provides a single "logical" service to the user

**High Performance**
Client optimized for interactive display-intensive tasks; Server optimized for CPU-intensive operations

**Scalability**
Server can handle large # of clients

**Flexibility**
User interface of client supports a variety of end devices (PDA, Handy, laptop, wearable computer)

**Reliability**
Server should be able to survive client and communication problems
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

```
1. updateData

application1:DBUser

database:DBMS

2. changeNotification

application2:DBUser
```
Peer-to-Peer Architectural Style

Generalization of Client/Server Architectural Style

“Clients can be servers and servers can be clients”

Introduction a new abstraction: Peer

```
Peer

service1()

*  

service2()

*  

serviceN()

requester

provider

application1:DBUser

1. updateData

application2:DBUser

2. changeNotification

database:DBMS
```
Example: Peer-to-Peer Architectural Style

- 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
  - Services: sendBit() and receiveBit()
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

Middleware

Application

Presentation

Session

Transport

Network

DataLink

Physical

Abstraction provided By Middleware

Object

Socket

TCP/IP

Ethernet

Wire
Repository Architectural Style

- Subsystems access and modify data from a single data structure called the **repository**
- Subsystems are loosely coupled (interact only through the repository)
- Control flow is dictated by the repository through triggers or by the subsystems through locks and synchronization primitives

![Diagram showing the relationship between Subsystem and Repository with method signatures: createData(), setData(), getData(), searchData()](image-url)
Examples of Repository Architectural Style

- Hearsay II speech understanding system ("Blackboard architecture")
- Database Management Systems
- Modern Compilers
Model-View-Controller (MVC) 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

```
<table>
<thead>
<tr>
<th>Controller</th>
<th>initiator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
</tr>
<tr>
<td>Model</td>
<td>repository</td>
</tr>
<tr>
<td>View</td>
<td>notifier</td>
</tr>
</tbody>
</table>
```

- **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
Example of a File System Based on the MVC Architectural Style
2. User types new filename

1. Views subscribe to event

3. Request name change in model

4. Notify subscribers

5. Updated views

Sequence of Events (Collaborations)

Controller

InfoView

Model

FolderView
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
  - Client Server, Peer-to-Peer, Repository, Model-View-Controller, …