Chapter 6
System Design: Decomposing the System
Requirements Elicitation & Analysis results in;
• Non-functional requirements and constraints
• Use-case model (functional model)
• Object Model
• Dynamic model

The activities of System design;
• Identify design goals
• Design the initial subsystem decomposition
• Refine the subsystem decomposition to address design goals
Design is Difficult

• There are two ways of constructing a software design (Tony Hoare):
  • One way is to make it so simple that there are obviously no deficiencies
  • The other way is to make it so complicated that there are no obvious deficiencies."

• Corollary (Jostein Gaarder):
  • If our brain would be so simple that we can understand it, we would be too stupid to understand it.

Sir Antony Hoare, *1934
- Quicksort
- Hoare logic for verification
- CSP (Communicating Sequential Processes): modeling language for concurrent processes (basis for Occam).

Jostein Gardner, *1952, writer
Uses metafiction in his stories:
Fiction which uses the device of fiction
- Best known for: „Sophie's World“.
Why is Design so Difficult?

• **Analysis:** Focuses on the application domain
• **Design:** Focuses on the solution domain
  • The solution domain is changing very rapidly
    • Halftime knowledge in software engineering: About 3-5 years
  ➢ Design knowledge is a moving target
The Scope of System Design

• Bridge the gap
  • between a problem and an 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.

Problem

System Design

System
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.

System Design

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

System Design I (This Lecture)

0. Overview of System Design
1. Design Goals
2. Subsystem Decomposition, Software Architecture

System Design II (Next Lecture)

3. Concurrency: Identification of parallelism
4. Hardware/Software Mapping:
   Mapping subsystems to processors
5. Persistent Data Management: Storage for entity objects
6. Global Resource Handling & Access Control:
   Who can access what?
7. Software Control: Who is in control?
8. Boundary Conditions: Administrative use cases.
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 Parallelism

Object Model
4. Hardware/Software Mapping
   - Special Purpose Systems
   - Buy vs Build
   - Allocation of Resources
   - Network Connectivity

5. Data Management
   - Persistent Objects
   - File system vs Database

Functional Model
8. Boundary Conditions
   - Initialization
   - Termination
   - Failure

Dynamic Model
7. Software Control
   - Monolithic vs Event-Driven
   - vs Concurrent Processes

6. Global Resource Handling
   - Access Control List
   - vs Capabilities
   - Security
Example of Design Goals

Design goals guide the decisions to be made by developers

- 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

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

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

Runtime Efficiency
- Reliability
- Portability
- Good documentation

Flexibility
- Client
- End User
- Developer/Maintainer
Typical Design Trade-offs

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

- Security vs Usability
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
  - A subsystem is characterized by the services it provides to other subsystems

- **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.

- **Advantage of System Decomposition?**
Example: Services provided by the ARENA Subsystems

**Tennis**

- **Tournament**
  - Manages tournaments, promotions, applications

- **User Management**
  - Administers user accounts

- **User Directory**
  - Stores user profiles (contact info & subscriptions)

- **Statistics**
  - Stores results of archived tournaments

- **Session Management**
  - Maintains state during matches

- **Component Management**
  - Adds games, styles, and expert rating formulas

- **Advertisement**
  - Manages advertisement banners & sponsorships

**Services are described by subsystem interfaces**
Conway’s Law

Conway’s Law

"Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization’s communication structure."
Melvin E. Conway, 1967

The structure of software reflects the organizational structure that produced it

* https://en.wikipedia.org/wiki/Conway%27s_law
Conway’s Law – Example

• Consider a large system $S$ that the government wants to build. The government hires Company X to build system $S$. Say company X has three engineering groups, $E_1$, $E_2$ and $E_3$ that participate in the project.

• Conway’s law suggest that it is likely that the resultant system will consist of 3 major subsystems ($S_1$, $S_2$, and $S_3$), each built by one of the engineering groups.

• Further, the resultant interfaces among subsystems ($S_1$-$S_2$, $S_1$-$S_3$, $S_2$-$S_3$) will reflect the quality and nature of the real-world interpersonal communications among respective engineering groups ($E_1$-$E_2$, $E_1$-$E_3$, $E_2$-$E_3$)
Packages

- A **package** in the Unified Modeling Language is used to group elements (as **subsystem**).
- A package may contain other packages, thus providing for a hierarchical organization of packages.
- Pretty much all UML elements can be grouped into packages.
Package Diagram

To simplify complex class diagrams, you can group classes into packages. A package is a collection of logically related UML elements.

A package diagram depicts the dependencies between the packages.

The dotted arrows are dependencies. One package depends on another if changes in the other could possibly force changes in the first.

http://edn.embarcadero.com/article/31863
Subsystem Decomposition - Example

Accident Management System

FieldOfficerInterface

DispatcherInterface

Notification

IncidentManagement
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
Coupling and Dependency

Uncoupled

Loosely Couple: Some Dependencies

Highly Couple: Many Dependencies

‘ripple’ effect
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)?
Examples of Reducing the coupling of subsystems

Alternative 1: Direct access to the Database subsystem

What can be improved?
Example of Reducing the couple of subsystems

Alternative 2: Indirect access to the Database through a Storage subsystem

Diagram:
- Storage
  - ResourceManagement
  - IncidentManagement
  - MapManagement
  - Database
Decision Tracking System

- Decision Tracking system for recording design problems, discussions, decisions etc.

What can be improved?
Alternative System Decomposition

```
RationaleSubsystem
   
   Criterion assesses Alternative
   *
   DesignProblem solvableBy based-on
   resolvedBy Decision

PlanningSubsystem
   
   SubTask implementedBy
   *
   ActionItem
   Task
   
   subtasks
```
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
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.
3-Layer Architectural Style

A 3-Layer Architectural Style is a hierarchy of 3 layers usually called presentation, application and data layer.

Presentation Layer (Client Layer)

Application Layer (Middleware, Business Logic)

Data Layer

Existing System

Operating System, Libraries

Appeared first in 1965, proposed by Dijkstra, in the design of the T.H.E. system.
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:
  - Runtime dependencies are associations with dashed lines
  - Compile time dependencies are associations with solid lines.
Example of a Subsystem Decomposition

Partition relationship

Layer 1

Layer 2

Layer 3

Relationship „depends on“

Relationship „calls“
Building Systems as a Set of Layers

A system is a hierarchy of layers, each using language primitives offered by the lower layers.
Closed Architecture (Opaque Layering)

- Each layer can only call operations from layer below

**Design goals:** Maintainability, flexibility
Opaque Layering in ARENA

Interface

Application Logic

ArenaServer

ArenaClient

UserManagement

AdvertisementManagement

GameManagement

TournamentManagement

Notification

Storage

ArenaStorage

Storage

Application Logic

Interface

opaque layering in arena

arena server

arena client

user management

advertisement management

game management

tournament management

notification

storage

arena storage
Open Architecture (Transparent Layering)

- Each layer can call operations from any layer below

Design goal: Runtime efficiency
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 other 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.

```
Client
   *.requester
   *provider

Server
   +service1()
   +service2()
   +serviceName()
```
Client/Server Architectures

- Well suited for distributed systems that manage large amounts of data
- 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
- A measure of success with which the observed behavior of a system confirms 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

```
application1:DBUser
1. updateData
```

```
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”

How do we model this statement? With Inheritance?

Introduction a new abstraction: Peer

“Clients and servers can be both peers”
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

- **Model**: encapsulates core data and functionality; independent of specific output representations or input behavior
- **View**: Display information to the user; obtains data from the model; multiple views of the same model are possible.
- **Controller**: Receive input events which are translated to model and view services; user interacts solely through controller
- Well suited for interactive systems, especially when multiple views of the same model are needed.
Model-View-Controller Example

Political Elections

Core Data

Black: 30%
Red: 20%
Green: 17%
Others: 18%

Other
Green
Red
Black

input
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**

```
Controller

initiator

* 1 repository

View

subscriber

* 1 notifier

Model

Better understanding with a Collaboration Diagram
```
Example: Modeling the Sequence of Events in MVC

4.0 User types new filename

5.0 Request name change in model

1.0 Subscribe

6.0 Notify subscribers

7.0 Show updated views

UML Collaboration Diagram

UML Class Diagram
3-Layer Architectural Style

A 3-Layer Architectural Style is a hierarchy of 3 layers usually called presentation, application and data layer.
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.
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
Summary

• System Design
  • An activity that reduces the gap between the problem and a solution

• 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, Model-View-Controller.