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
    - What I teach today will be out of date in 3 years
    - Cost of hardware rapidly sinking
- “Design window”:
  - Time in which design decisions have to be made
- Technique
  - Time-boxed prototyping
The Purpose of System Design

- Bridging the gap between desired and existing system in a manageable way
- Use Divide and Conquer
  - We model the new system to be developed as a set of subsystems
System Design

1. Design Goals
   - Definition
   - Trade-offs

2. System Decomposition
   - Layers/Partitions
   - Cohesion/Coupling

3. Concurrency
   - Identification of Threads

4. Hardware/Software Mapping
   - Special purpose
   - Buy or Build Trade-off
   - Allocation
   - Connectivity

5. Data Management
   - Persistent Objects
   - Files
   - Databases
   - Data structure

6. Global Resource Handling
   - Access control
   - Security

7. Software Control
   - Monolithic
   - Event-Driven
   - Threads
   - Conc. Processes

8. Boundary Conditions
   - Initialization
   - Termination
   - Failure
Overview

System Design

0. Overview of System Design
1. Design Goals
2. Subsystem Decomposition
3. Concurrency
4. Hardware/Software Mapping
5. Persistent Data Management
6. Global Resource Handling and Access Control
7. Software Control
8. Boundary Conditions
How to use the results from the Requirements Analysis for 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
How do we get the Design Goals?

Let’s look at a small example

_Current Situation:_
- Computers must be used in the office

_What we want:_
- A computer that can be used in mobile situations.
Establish New Design Goals

- Mobile Network Connection
- Multiple Output Devices
- Location-Based
- Multimodal Input (Users Gaze, Users Location, …)
- Vague input
Sharpen the Design Goals

Location-based input

◆ Input depends on user location
◆ Input depends on the direction where the user looks (“egocentric systems”)

Multi-modal input

◆ The input comes from more than one input device

Dynamic connection

◆ Contracts are only valid for a limited time

Is there a possibility of further generalizations?

Example: location can be seen as a special case of context

◆ User preference is part of the context
◆ Interpretation of commands depends on context
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

**Client**
- Low cost
- Increased Productivity
- Backward-Compatibility
- Traceability of requirements
- Rapid development
- Flexibility

**End User**
- Functionality
- User-friendliness
- Ease of Use
- Ease of learning
- Fault tolerant
- Robustness

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

**Runtime Efficiency**
- Portability
- Good Documentation
**Typical Design Trade-offs**

- Functionality vs. Usability
- Cost vs. Robustness
- Efficiency vs. Portability
- Rapid development vs. Functionality
- Cost vs. Reusability
- Backward Compatibility vs. Readability
Nonfunctional Requirements may give a clue for the use of Design Patterns

♦ Read the problem statement again
♦ Use textual clues (similar to Abbot’s technique in Analysis) to identify design patterns

♦ Text: “manufacturer independent”, “device independent”, “must support a family of products”
  ◆ Abstract Factory Pattern

♦ Text: “must interface with an existing object”
  ◆ Adapter Pattern

♦ Text: “must provide a policy independent from the mechanism”
  ◆ Strategy Pattern
Textual Clues in Nonfunctional Requirements

- *Text*: “complex structure”, “must have variable depth and width”  
  ◆ Composite Pattern
- *Text*: “must interface to an set of existing objects”  
  ◆ Façade Pattern
- *Text*: “must be location transparent”  
  ◆ Proxy Pattern
- *Text*: “must be extensible”, “must be scalable”  
  ◆ Observer Pattern
Section 2. System Decomposition

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

♦ (Subsystem) Service:
  ◆ Group of operations provided by the subsystem
  ◆ Seed for services: Subsystem use cases

♦ Service is specified by Subsystem interface:
  ◆ Specifies interaction and information flow from/to subsystem boundaries, but not inside the subsystem.
  ◆ Should be well-defined and small.
  ◆ Often called API: Application programmer’s interface, but this term should used during implementation, not during System Design
Services and Subsystem Interfaces

♦ **Service**: A set of related operations that share a common purpose
  - **Notification subsystem service**:
    - LookupChannel()
    - SubscribeToChannel()
    - SendNotice()
    - UnsubscribeFromChannel()
  - **Services are defined in System Design**

♦ **Subsystem Interface**: Set of fully typed related operations.
  - **Subsystem Interfaces are defined in Object Design**
  - **Also called application programmer interface (API)**
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?
Subsystem Decomposition Example
Definition: Subsystem Interface Object

♦ A Subsystem Interface Object provides a service
  ◆ This is the set of public methods provided by the subsystem
  ◆ The Subsystem interface describes all the methods of the subsystem interface object
♦ Use a Facade pattern for the subsystem interface object
System as a set of subsystems communicating via a software bus

A Subsystem Interface Object publishes the service (= Set of public methods) provided by the subsystem
A 3-layered Architecture

What is the relationship between Modeling and Authoring? Are other subsystems needed?
Another Example: ARENA Subsystem decomposition

- User Interface
- Advertisement
- Component Management
- Session Management
- Tournament
- Tournament Statistics
- User Management
- User Directory
Services provided by ARENA Subsystems

- **Advertisement**
  - Manages advertisement banners and sponsorships.

- **Component Management**
  - For adding games, styles, and expert rating formulas.

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

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

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

- **User Management**
  - Administers user accounts.

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

- **Interface**
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 should have as maximum cohesion and minimum coupling as possible:
  ◆ How can we achieve high cohesion?
  ◆ How can we achieve loose coupling?
Partitions and Layers

Partitioning and layering are techniques to achieve low coupling.

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

♦ **Partitions** vertically 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 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 subsystem have mutual but not deep knowledge about each other
  ◆ Partition A “Calls” partition B and partition B “Calls” partition A
Dijkstra: T.H.E. operating system (1965)

A system should be developed by an ordered set of virtual machines, each built in terms of the ones below it.
Virtual Machine

♦ A virtual machine is an abstraction
  ◆ It provides a set of attributes and operations.

♦ A virtual machine is a subsystem
  ◆ It is connected to higher and lower level virtual machines by "provides services for" associations.

♦ Virtual machines can implement two types of software architecture
  ◆ Open and closed architectures.
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.
- Layered systems often have a chicken-and-egg problem
  
  Example: Debugger opening the symbol table when the file system needs to be debugged

```
A: Debugger

D: File System

G: Op. System
```
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 interactions between applications and a database.]

1. updateData
2. changeNotification
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
ARENA: The Objectives

• Provide a **framework for tournament organizers**
  • to customize the number and sequence of matchers and the accumulation of expert rating points

• Provide a **framework for game developers**
  • for developing new games, or for adapting existing games into the ARENA framework

• Provide an infrastructure for **advertisers**.
ARENA Functional Requirements

• **Spectators** must be able to watch matches in progress without prior registration and without prior knowledge of the match

• The **operator** must be able to add new games.

• The **Player** must be able to:
  - ask for a match waiting for another player,
  - register for a tournament,
  - join to an already waiting match
ARENA Nonfunctional Requirements

- The system must support
  - parallel tournaments,
  - Each involving up to 64 players
  - and several hundreds of spectators.
  - The ARENA server **must be available 24 hours a day**

- The operator must be able to add new games without modifications to the existing system

- **ARENA must be able to dynamically interface to existing games** provided by other game developers.
ARENA Target Environment

Example:

- Users must be able to manage ARENA tournaments with any Web Browser
- The web page must be validated through the W3C Markup Validation Service
- Interaction with the ARENA Server must be via HTTP/1.1.

Development environment

- “Tournament management will be tested with Internet Explorer and Firefox”
- “The implementation language will be J2EE”
- “The IDE will be Eclipse 4.3”
ARENA Software Subsystem Decomposition

User Interface

Advertisement

Tournament

Component Management

Session Management

Tournament Statistics

User Management

User Directory

Object-Oriented Software Engineering: Using UML, Patterns, and Java
We first distinguish two main parts of the ARENA subsystem:
- the game organization part of the system, which is responsible for coordinating Users when organizing a Tournament, and
- the game playing part, in which Players conduct individual Matches in the scope of a Tournament.

For the game playing part, the client server architecture may not be sufficient for synchronous games in which the action of one player can trigger events for another player within a relatively short time.

Synchronous behavior could be simulated with polling; however, because of scalability and responsiveness goals, we select a peer-to-peer architecture in which MatchFrontEndPeer subsystems provide the user interface and a GamePeer maintains the state of the matches currently under way and enforces the game rules.
MatchFrontEndPeers may also communicate directly with each other for real-time games.

To achieve the game independence design goal, ARENA provides a framework for both the MatchFrontEndPeer and the GamePeer, while the bulk of the game logic is provided by customized game-dependent components.

Adding a game consists of developing adapters for existing games or ARENA-compliant components for new games. The TournamentManagement subsystem uses the GameManagement subsystem to initiate a GamePeer and to collect the results of the individual Matches.

MatchFrontEndPeer uses the AdvertisementManagement subsystem to retrieve Advertisements.
ARENA subsystem decomposition, game playing part (UML component diagram).
For the game organization part, a three-tier architectural style is suited in which an ArenaClient subsystem provides a front end for users to initiate all organization-related use cases (e.g. AnnounceTournament, ApplyForTournament, RegisterPlayer).

The ArenaServer subsystem is responsible for access control and concurrency control, and delegates to nested subsystems for the application logic.

Different subsystems are dedicated to the user management of users, advertisements, tournaments, and games.

The bottom tier is realized by the ArenaStorage subsystem, responsible for storing any persistent objects, except for those representing Match states.
ARENA subsystem decomposition, game organization part (UML component diagram, layers shown as UML packages).
ARENA Object Model

Game

League

Tournament

Round

Player

Match

Tournament Style

KOStyle

RoundRobin
ARENA Object Model (2)

- **Game**
  - **TicTacToe**
  - **Chess**

- **League**
  - **Tournament**
    - **Round**
      - **Match**

- **Player**

- **Move**

- **MatchPanel Factory**
  - creates **MatchPanel**
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 datastructure, the Controller subsystem explicitly dictate the control flow
## Monitoraggio Attività (Tutti i processi)

<table>
<thead>
<tr>
<th>Nome Processo</th>
<th>% CPU</th>
<th>Tempo CPU</th>
<th>Thread</th>
<th>Riattivazioni da stop</th>
<th>IDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>mdworker</td>
<td>39,7</td>
<td>2,33</td>
<td>19</td>
<td>2</td>
<td>56655</td>
</tr>
<tr>
<td>WindowServer</td>
<td>5,9</td>
<td>7:31:33,92</td>
<td>4</td>
<td>55</td>
<td>174</td>
</tr>
<tr>
<td>kernel_task</td>
<td>3,6</td>
<td>5:42:07,24</td>
<td>114</td>
<td>339</td>
<td>0</td>
</tr>
<tr>
<td>Twitter</td>
<td>2,6</td>
<td>10:26,63</td>
<td>15</td>
<td>3</td>
<td>487</td>
</tr>
<tr>
<td>Eclipse</td>
<td>1,8</td>
<td>2:13:07,44</td>
<td>42</td>
<td>161</td>
<td>1279</td>
</tr>
<tr>
<td>X-Lite</td>
<td>1,7</td>
<td>2:31:08,08</td>
<td>35</td>
<td>152</td>
<td>483</td>
</tr>
<tr>
<td>Finder</td>
<td>1,4</td>
<td>51:55,67</td>
<td>11</td>
<td>2</td>
<td>472</td>
</tr>
<tr>
<td>Dock</td>
<td>0,9</td>
<td>1:57:59,94</td>
<td>11</td>
<td>0</td>
<td>464</td>
</tr>
<tr>
<td>Eclipse</td>
<td>0,9</td>
<td>2:14:16,81</td>
<td>57</td>
<td>77</td>
<td>33196</td>
</tr>
<tr>
<td>Little Snitch Agent</td>
<td>0,8</td>
<td>1:17:11,50</td>
<td>3</td>
<td>13</td>
<td>359</td>
</tr>
</tbody>
</table>

Processes: 275 total, 2 running, 7 stuck, 266 sleeping, 1676 threads

Load Avg: 1.95, 2.19, 2.12
CPU usage: 2.7% user, 3.45% sys, 94.47% idle
SharedLibs: 10M resident, 17M data, 0B free
VM: 666G vspace, 1071M framework vspace, 3745518(0) swapins, 4294312(0) swapouts.
Networks: packets: 20704132/12G in, 22196276/12G out
in sintesi

• è una applicazione del pattern Observer alle interfaccie utente (GUI ma non necessariamente)
• lunga tradizione in smalltalk
• non è propriamente un design pattern ma un “architectural pattern”
  – perché i vari ruoli possono essere ricoperti da insiemi di classi anziché da singole classi

• intento
  disaccoppiare:
  – rappresentazione del modello di dominio (model)
  – interfaccia utente (view), non necessariamente GUI
  – controllo dell’interazione uomo-macchina (controller)
il pattern MVC

• struttura

qui, per semplificare, supponiamo che modello vista e controller possano essere implementati (o rappresentati) con un sola classe (o interfaccia)

• interazione

qui, per semplificare, supponiamo che il modello sia rappresentato da un solo oggetto, osservato con una sola view e controllato con un solo controller
controller: vari punti di vista

possiamo individuare due punti di vista rispetto al ruolo del controller
– punto di vista vicino al model
  • il controller dovrebbe raggruppare le azioni/transazioni “compesse” che possono essere effettuate sul modello
  • è del tutto indipendente da aspetti tecnologici legati all’interfaccia grafica
– punto di vista vicino alla view
  • il controller gestisce eventi (mouse clicks)
  • deve fornire una interfaccia che sia “compatibile” con il toolkit grafico utilizzato
  • può rappresentare il concetto di azione “elementare” effettuabile tramite l’interfaccia grafica
    – abilitazione/disabilitazione, nomi, tooltip, ecc.

• entrambe le esigenze possono essere soddisfatte
  – dividiamo le responsabilità in due categorie di oggetti distinti
  – ciascuna categoria soddisfa una delle esigenze

– categorie:
  • controller (propriamente detti)
    – tutto ciò che non è legato al toolkit grafico
  • azioni
    – tutto ciò che è legato a toolkit grafico
    – sono spesso classi molto piccole che delegano la maggior parte delle attività al controller
    – possono essere considerate degli adapter verso il controller
MVC: architettura target

- vari modi di interagire con il sistema
- View1 package
- ViewN package
- Library1
- Library2
- rappresentazione delle “procedure utente” (vedi use cases)
- rappresentazione del dominio
- Controller package
- Model package
- «modify»
- «query»
- «use»

MVC: cambiamenti al sistema

- impatto vs. probabilità
- impatto: medio
  probabilità: media, dipende dalla accuratezza dell’analisi
- impatto: alto
  probabilità: media, dipende dall’accuratezza dell’analisi e del porgetto
- impatto: basso
  probabilità: alta, è ciò che l’utente e i manager vedono!
MVC: dettagli di progetto
Sequence of Events (Collaborations)

2. User types new filename

1. Views subscribe to event

3. Request name change in model

4. Notify subscribers

5. Updated views

:File System Controller

:File System

:InfoView

:FolderView
Repository Architectural Style (*Blackboard Architecture, Hearsay II Speech Recognition System*)

♦ 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

- Hearsay II speech understanding system ("Blackboard architecture")
- Database Management Systems
- Modern Compilers
3. Concurrency

♦ Identify concurrent threads and address concurrency issues.
♦ Design goal: response time, performance.

♦ Threads
  ◆ A thread of control is a path through a set of state diagrams on which a single object is active at a time.
  ◆ A thread remains within a state diagram until an object sends an event to another object and waits for another event
  ◆ Thread splitting: Object does a nonblocking send of an event.
Concurrency (continued)

♦ Two objects are inherently concurrent if they can receive events at the same time without interacting

♦ Inherently concurrent objects should be assigned to different threads of control

♦ Objects with mutual exclusive activity should be folded into a single thread of control (Why?)
Concurrency Questions

♦ Which objects of the object model are independent?
♦ What kinds of threads of control are identifiable?
♦ Does the system provide access to multiple users?
♦ Can a single request to the system be decomposed into multiple requests? Can these requests be handled in parallel?
Implementing Concurrency

♦ Concurrent systems can be implemented on any system that provides
  ◆ physical concurrency (hardware)

or

◆ logical concurrency (software): Scheduling problem (Operating systems)
4. Hardware Software Mapping

♦ This activity addresses two questions:
  ◆ How shall we realize the subsystems: Hardware or Software?
  ◆ How is the object model mapped on the chosen hardware & software?
    ◇ Mapping Objects onto Reality: Processor, Memory, Input/Output
    ◇ Mapping Associations onto Reality: Connectivity

♦ Much of the difficulty of designing a system comes from meeting externally-imposed hardware and software constraints.
  ◆ Certain tasks have to be at specific locations
Mapping the Objects

♦ Processor issues:
  ◆ Is the computation rate too demanding for a single processor?
  ◆ Can we get a speedup by distributing tasks across several processors?
  ◆ How many processors are required to maintain steady state load?

♦ Memory issues:
  ◆ Is there enough memory to buffer bursts of requests?

♦ I/O issues:
  ◆ Do you need an extra piece of hardware to handle the data generation rate?
  ◆ Does the response time exceed the available communication bandwidth between subsystems or a task and a piece of hardware?
Mapping the Subsystems Associations: Connectivity

♦ Describe the *physical connectivity* of the hardware
  ◆ Often the physical layer in ISO’s OSI Reference Model
    ◇ Which associations in the object model are mapped to physical connections?
    ◇ Which of the client-supplier relationships in the analysis/design model correspond to physical connections?

♦ Describe the *logical connectivity* (subsystem associations)
  ◆ Identify associations that do not directly map into physical connections:
    ◇ How should these associations be implemented?
Typical Informal Example of a Connectivity Drawing

- Application Client
- Application Client
- Application Client
- Communication Agent for Application Clients
- Communication Agent for Application Clients
- Communication Agent for Data Server
- Communication Agent for Data Server
- Local Data Server
- Global Data Server
- Global Data Server
- Global Data Server
- OODBMS
- RDBMS

Physical Connectivity

Logical Connectivity

TCP/IP

Ethernet

LAN

Backbone Network
Logical vs Physical Connectivity and the relationship to Subsystem Layering

![Diagram showing logical and physical connectivity between layers](attachment:Logical_vs_Physical_Connectivity_Diagram.png)
Hardware/Software Mapping Questions

♦ What is the connectivity among physical units?
  ◆ Tree, star, matrix, ring

♦ What is the appropriate communication protocol between the subsystems?
  ◆ Function of required bandwidth, latency and desired reliability, desired quality of service (QOS)

♦ Is certain functionality already available in hardware?

♦ Do certain tasks require specific locations to control the hardware or to permit concurrent operation?
  ◆ Often true for embedded systems

♦ General system performance question:
  ◆ What is the desired response time?
Connectivity in Distributed Systems

♦ If the architecture is distributed, we need to describe the network architecture (communication subsystem) as well.

♦ Questions to ask
  ◆ What are the transmission media? (Ethernet, Wireless)
  ◆ What is the Quality of Service (QOS)? What kind of communication protocols can be used?
  ◆ Should the interaction asynchronous, synchronous or blocking?
  ◆ What are the available bandwidth requirements between the subsystems?
    ◆ Stock Price Change -> Broker
    ◆ Icy Road Detector -> ABS System
Drawing Hardware/Software Mappings in UML

♦ System design must model static and dynamic structures:
  ◆ Component Diagrams for static structures
    ◆ show the structure at design time or compilation time
  ◆ Deployment Diagram for dynamic structures
    ◆ show the structure of the run-time system

♦ Note the lifetime of components
  ◆ Some exist only at design time
  ◆ Others exist only until compile time
  ◆ Some exist at link or runtime
Component Diagram

- Component Diagram
  - A graph of components connected by dependency relationships.
  - Shows the dependencies among software components
    - source code, linkable libraries, executables
- Dependencies are shown as dashed arrows from the client component to the supplier component.
  - The kinds of dependencies are implementation language specific.
- A component diagram may also be used to show dependencies on a façade:
  - Use dashed arrow the corresponding UML interface.
Component Diagram Example

UML Component

Scheduler

reservations

Planner

update

GUI
Deployment Diagram

- Deployment diagrams are useful for showing a system design after the following decisions are made:
  - Subsystem decomposition
  - Concurrency
  - Hardware/Software Mapping

- A deployment diagram is a graph of nodes connected by communication associations:
  - Nodes are shown as 3-D boxes.
  - Nodes may contain component instances.
  - Components may contain objects (indicating that the object is part of the component)
Deployment Diagram Example

:HostMachine

:Scheduler

<<database>>
meetingsDB

:PC

:Planner

Compile Time Dependency

Runtime Dependency
5. Data Management

♦ Some objects in the models need to be persistent
  ◆ Provide clean separation points between subsystems with well-defined interfaces.

♦ A persistent object can be realized with one of the following
  ◆ Data structure
    ‣ If the data can be volatile
  ◆ Files
    ‣ Cheap, simple, permanent storage
    ‣ Low level (Read, Write)
    ‣ Applications must add code to provide suitable level of abstraction
  ◆ Database
    ‣ Powerful, easy to port
    ‣ Supports multiple writers and readers
File or Database?

- When should you choose a file?
  - Are the data voluminous (bit maps)?
  - Do you have lots of raw data (core dump, event trace)?
  - Do you need to keep the data only for a short time?
  - Is the information density low (archival files, history logs)?

- When should you choose a database?
  - Do the data require access at fine levels of details by multiple users?
  - Must the data be ported across multiple platforms (heterogeneous systems)?
  - Do multiple application programs access the data?
  - Does the data management require a lot of infrastructure?
Database Management System

♦ Contains mechanisms for describing data, managing persistent storage and for providing a backup mechanism
♦ Provides concurrent access to the stored data
♦ Contains information about the data (“meta-data”), also called data schema.
Issues To Consider When Selecting a Database

♦ Storage space
  ♦ Database require about triple the storage space of actual data

♦ Response time
  ♦ Mode databases are I/O or communication bound (distributed databases). Response time is also affected by CPU time, locking contention and delays from frequent screen displays

♦ Locking modes
  ♦ Pessimistic locking: Lock before accessing object and release when object access is complete
  ♦ Optimistic locking: Reads and writes may freely occur (high concurrency!) When activity has been completed, database checks if contention has occurred. If yes, all work has been lost.

♦ Administration
  ♦ Large databases require specially trained support staff to set up security policies, manage the disk space, prepare backups, monitor performance, adjust tuning.
---

**Object-Oriented Databases**

- Support all fundamental object modeling concepts
  - **Classes, Attributes, Methods, Associations, Inheritance**
- Mapping an object model to an OO-database
  - Determine which objects are persistent.
  - Perform normal requirement analysis and object design
  - Create single attribute indices to reduce performance bottlenecks
  - Do the mapping (specific to commercially available product).
    **Example:**
    - In ObjectStore, implement classes and associations by preparing C++
      declarations for each class and each association in the object model

---
Relational Databases

- Based on relational algebra

- Data is presented as 2-dimensional tables. Tables have a specific number of columns and arbitrary numbers of rows
  - **Primary key**: Combination of attributes that uniquely identify a row in a table. Each table should have only one primary key
  - **Foreign key**: Reference to a primary key in another table

- SQL is the standard language defining and manipulating tables.

- Leading commercial databases support constraints.
  - **Referential integrity**, for example, means that references to entries in other tables actually exist.
Data Management Questions

♦ Should the data be distributed?
♦ Should the database be extensible?
♦ How often is the database accessed?
♦ What is the expected request (query) rate? In the worst case?
♦ What is the size of typical and worst case requests?
♦ Do the data need to be archived?
♦ Does the system design try to hide the location of the databases (location transparency)?
♦ Is there a need for a single interface to access the data?
♦ What is the query format?
♦ Should the database be relational or object-oriented?
<table>
<thead>
<tr>
<th>Attributes</th>
<th>NoSQL Databases</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Database model</strong></td>
<td>Document-Stored</td>
</tr>
<tr>
<td>Data storage</td>
<td>Volatile memory</td>
</tr>
<tr>
<td>Protocol</td>
<td>Custom, binary (JSON)</td>
</tr>
<tr>
<td>Conditional entry updates</td>
<td>Yes</td>
</tr>
<tr>
<td>Mapreduce</td>
<td>Yes</td>
</tr>
<tr>
<td>Unicode</td>
<td>Yes</td>
</tr>
<tr>
<td>TTL for Entries</td>
<td>Yes</td>
</tr>
<tr>
<td>Compression</td>
<td>Yes</td>
</tr>
<tr>
<td>Integrity model</td>
<td>RASF</td>
</tr>
<tr>
<td>Atomicity</td>
<td>Conditional</td>
</tr>
<tr>
<td>Consistency</td>
<td>Yes</td>
</tr>
<tr>
<td>Isolation</td>
<td>No</td>
</tr>
<tr>
<td>Durability (data storage)</td>
<td>No</td>
</tr>
<tr>
<td>Transactions</td>
<td>No</td>
</tr>
<tr>
<td>Referential integrity</td>
<td>No</td>
</tr>
<tr>
<td>Revision control</td>
<td>No</td>
</tr>
<tr>
<td>Secondary Indexes</td>
<td>Yes</td>
</tr>
<tr>
<td>Indexing</td>
<td>Yes</td>
</tr>
<tr>
<td>Full text search</td>
<td>No</td>
</tr>
<tr>
<td>GeoSpatial Indexes</td>
<td>Yes</td>
</tr>
<tr>
<td>Graph support</td>
<td>No</td>
</tr>
<tr>
<td>Horizontal scalability</td>
<td>Yes</td>
</tr>
<tr>
<td>Replication</td>
<td>Yes</td>
</tr>
<tr>
<td>Replication mode</td>
<td>Master-Slave</td>
</tr>
<tr>
<td>Sharding</td>
<td>Yes</td>
</tr>
<tr>
<td>Shared nothing architecture</td>
<td>Yes</td>
</tr>
<tr>
<td>Value size max</td>
<td>16MB</td>
</tr>
<tr>
<td>Operating system</td>
<td>Cross-platform</td>
</tr>
<tr>
<td>Programming language</td>
<td>C++</td>
</tr>
</tbody>
</table>

**Diagram:**

*How to write a CV*

- Do you have any expertise in SQL?
  - No
  - geek &バー

Doesn't matter, "Expert in no SQL."

Leverage the NoSQL boom
6. Global Resource Handling

- Discusses access control
- Describes access rights for different classes of actors
- Describes how objects guard against unauthorized access
Defining Access Control

♦ In multi-user systems different actors have access to different functionality and data.

◆ During **analysis** we model these different accesses by associating different use cases with different actors.

◆ During **system design** we model these different accesses by examining the object model by determining which objects are shared among actors.

  - Depending on the security requirements of the system, we also define how actors are authenticated to the system and how selected data in the system should be encrypted.
Access Matrix

- We model access on classes with an access matrix.
  - The rows of the matrix represent the actors of the system
  - The column represent classes whose access we want to control.

- **Access Right:** An entry in the access matrix. It lists the operations that can be executed on instances of the class by the actor.
Access Matrix Implementations

♦ Global access table: Represents explicitly every cell in the matrix as a (actor, class, operation) tuple.
  ◆ Determining if an actor has access to a specific object requires looking up the corresponding tuple. If no such tuple is found, access is denied.

♦ Access control list associates a list of (actor, operation) pairs with each class to be accessed.
  ◆ Every time an object is accessed, its access list is checked for the corresponding actor and operation.
  ◆ Example: guest list for a party.

♦ A capability associates a (class, operation) pair with an actor.
  ◆ A capability provides an actor to gain control access to an object of the class described in the capability.
  ◆ Example: An invitation card for a party.

♦ Which is the right implementation?
Global Resource Questions

♦ Does the system need authentication?
♦ If yes, what is the authentication scheme?
  ◆ User name and password? Access control list
  ◆ Tickets? Capability-based
♦ What is the user interface for authentication?
♦ Does the system need a network-wide name server?
♦ How is a service known to the rest of the system?
  ◆ At runtime? At compile time?
  ◆ By port?
  ◆ By name?
7. Decide on Software Control

Choose implicit control (non-procedural, declarative languages)
  ◆ Rule-based systems
  ◆ Logic programming

Choose explicit control (procedural languages): Centralized or decentralized

Centralized control: Procedure-driven or event-driven
  ♦ Procedure-driven control
    ◆ Control resides within program code. Example: Main program calling procedures of subsystems.
    ◆ Simple, easy to build, hard to maintain (high recompilation costs)
  ♦ Event-driven control
    ◆ Control resides within a dispatcher calling functions via callbacks.
    ◆ Very flexible, good for the design of graphical user interfaces, easy to extend
Software Control (continued)

♦ Decentralized control
  ◆ Control resides in several independent objects.
  ◆ Possible speedup by mapping the objects on different processors, increased communication overhead.
  ◆ Example: Message based system.
Centralized vs. Decentralized Designs

Should you use a centralized or decentralized design?

- Take the sequence diagrams and control objects from the analysis model
- Check the participation of the control objects in the sequence diagrams
  - If sequence diagram looks more like a fork: Centralized design
  - The sequence diagram looks more like a stair: Decentralized design

Centralized Design

- One control object or subsystem ("spider") controls everything
  - Pro: Change in the control structure is very easy
  - Con: The single control object is a possible performance bottleneck

Decentralized Design

- Not a single object is in control, control is distributed, That means, there is more than one control object
  - Con: The responsibility is spread out
  - Pro: Fits nicely into object-oriented development
8. Boundary Conditions

♦ Most of the system design effort is concerned with steady-state behavior.

♦ However, the system design phase must also address the initiation and finalization of the system. This is addressed by a set of new uses cases called administration use cases
  
  ◆ Initialization
  
   ♦ Describes how the system is brought from an non initialized state to steady-state ("startup use cases").

  ◆ Termination

   ♦ Describes what resources are cleaned up and which systems are notified upon termination ("termination use cases").

  ◆ Failure

   ♦ Many possible causes: Bugs, errors, external problems (power supply).
   ♦ Good system design foresees fatal failures ("failure use cases").
Example: Administrative Use cases for MyTrip

- Administration use cases for MyTrip (UML use case diagram).
- An additional subsystems that was found during system design is the server. For this new subsystem we need to define use cases.
- ManageServer includes all the functions necessary to start up and shutdown the server.
ManageServer Use Case

PlanningService
Administrator

- ManageServer
- <<include>>
- StartServer
- <<include>>
- ShutdownServer
- <<include>>
- ConfigureServer
Boundary Condition Questions

8.1 Initialization
- How does the system start up?
  - What data need to be accessed at startup time?
  - What services have to be registered?
- What does the user interface do at start up time?
  - How does it present itself to the user?

8.2 Termination
- Are single subsystems allowed to terminate?
- Are other subsystems notified if a single subsystem terminates?
- How are local updates communicated to the database?

8.3 Failure
- How does the system behave when a node or communication link fails? Are there backup communication links?
- How does the system recover from failure? Is this different from initialization?
**Modeling Boundary Conditions**

- Boundary conditions are best modeled as use cases with actors and objects.
- Actor: often the system administrator
- Interesting use cases:
  - Start up of a subsystem
  - Start up of the full system
  - Termination of a subsystem
  - Error in a subsystem or component, failure of a subsystem or component
- Task:
  - Model the startup of the ARENA system as a set of use cases.
Object Design

- Object design is the process of adding details to the requirements analysis and making implementation decisions.
- The object designer must choose among different ways to implement the analysis model with the goal to minimize execution time, memory and other measures of cost.
- Requirements Analysis: Use cases, functional and dynamic model deliver operations for object model.
- Object Design: Iterates on the models, in particular the object model and refine the models.
- Object Design serves as the basis of implementation.
Object Design: Closing the Gap

- **System**
  - **Application objects**
  - **Solution objects**
    - **Custom objects**
    - **Off-the-shelf components**
  - Problem
    - Requirements gap
    - Object design gap
    - System design gap
  - Machine
Examples of Object Design Activities

♦ Identification of existing components
♦ Full definition of associations
♦ Full definition of classes (System Design => Service, Object Design => API)
♦ Specifying the contract for each component
♦ Choosing algorithms and data structures
♦ Identifying possibilities of reuse
♦ Detection of solution-domain classes
♦ Optimization
♦ Increase of inheritance
♦ Decision on control
♦ Packaging
A More Detailed View of Object Design Activities

Select Subsystem

Specification
- Identifying missing attributes & operations
- Specifying visibility
- Specifying types & signatures
- Specifying constraints
- Specifying exceptions

Reuse
- Identifying components
- Adjusting components
- Identifying patterns
- Adjusting patterns
Detailed View of Object Design Activities (ctd)
A Little Bit of Terminology: Activities

♦ Object-Oriented methodologies use these terms:
  ◆ System Design Activity
    ♦ Decomposition into subsystems
  ◆ Object Design Activity
    ♦ Implementation language chosen
    ♦ Data structures and algorithms chosen

♦ Structured analysis/structured design uses these terms:
  ◆ Preliminary Design Activity
    ♦ Decomposition into subsystems
    ♦ Data structures are chosen
  ◆ Detailed Design Activity
    ♦ Algorithms are chosen
    ♦ Data structures are refined
    ♦ Implementation language is chosen
    ♦ Typically in parallel with preliminary design, not a separate activity
Outline of Today

- Reuse Concepts
- The use of inheritance
- Implementation vs Interface Inheritance
- Delegation
- Components
- Documenting the Object Design
- JavaDoc
Reuse Concepts

- Application objects versus solution objects
- Specification inheritance and implementation inheritance
- The Liskov Substitution Principle
- Delegation (Section 8.3.3)
- Delegation and inheritance in design patterns
Application domain vs solution domain objects

♦ Application objects, also called domain objects, represent concepts of the domain that are relevant to the system.
  ◆ They are identified by the application domain specialists and by the end users.

♦ Solution objects represent concepts that do not have a counterpart in the application domain,
  ◆ They are identified by the developers
  ◆ Examples: Persistent data stores, user interface objects, middleware.
**Application Domain vs Solution Domain Objects**

**Requirements Analysis**
(Language of Application Domain)

- Incident Report

**Object Design**
(Language of Solution Domain)

- Incident Report
  - Text box
  - Menu
  - Scrollbar
Implementation of Application Domain Classes

♦ New objects are often needed during object design:
  ◆ The use of design patterns introduces new classes
  ◆ The implementation of algorithms may necessitate objects to hold values
  ◆ New low-level operations may be needed during the decomposition of high-level operations

♦ Example: The EraseArea() operation in a drawing program.
  ◆ Conceptually very simple
  ◆ Implementation
    ♦ Area represented by pixels
    ♦ Repair() cleans up objects partially covered by the erased area
    ♦ Redraw() draws objects uncovered by the erasure
    ♦ Draw() erases pixels in background color not covered by other objects
Observation about Modeling of the Real World

♦ [Gamma et al 94]:
♦ Strict modeling of the real world leads to a system that reflects today’s realities but not necessarily tomorrow’s.

♦ There is a need for reusable and flexible designs

♦ Design knowledge complements application domain knowledge and solution domain knowledge.
The use of inheritance

- Inheritance is used to achieve two different goals
  - Description of Taxonomies
  - Interface Specification

- Identification of taxonomies
  - Used during requirements analysis.
  - Activity: identify application domain objects that are hierarchically related
  - Goal: make the analysis model more understandable

- Service specification
  - Used during object design
  - Activity:
    - Goal: increase reusability, enhance modifiability and extensibility

- Inheritance is found either by specialization or generalization
Metamodel for Inheritance

- Inheritance is used during analysis and object design
Taxonomy Example

- Mammal
  - Tiger
  - Wolf
  - Wale
Implementation Inheritance

- A very similar class is already implemented that does almost the same as the desired class implementation.

Example: I have a List class, I need a Stack class. How about subclassing the Stack class from the List class and providing three methods, Push() and Pop(), Top()?

Problem with implementation inheritance:

Some of the inherited operations might exhibit unwanted behavior. What happens if the Stack user calls Remove() instead of Pop()?
Implementation Inheritance vs Interface Inheritance

♦ Implementation inheritance
  ◆ Also called class inheritance
  ◆ Goal: Extend an applications’ functionality by reusing functionality in parent class
  ◆ Inherit from an existing class with some or all operations already implemented

♦ Interface inheritance
  ◆ Also called subtyping
  ◆ Inherit from an abstract class with all operations specified, but not yet implemented
Delegation as alternative to Implementation Inheritance

- Delegation is a way of making composition (for example aggregation) as powerful for reuse as inheritance
- In Delegation two objects are involved in handling a request
  - A receiving object delegates operations to its delegate.
  - The developer can make sure that the receiving object does not allow the client to misuse the delegate object

![Diagram of Delegation](image)
Delegation instead of Implementation Inheritance

♦ **Inheritance**: Extending a Base class by a new operation or overwriting an operation.

♦ **Delegation**: Catching an operation and sending it to another object.

♦ Which of the following models is better for implementing a stack?

```
+Add()  +Remove()
List

Stack
+Push()  +Pop()  +Top()
```

```
Stack
+Push()  +Pop()  +Top()
List

Remove()  Add()
```
Comparison: Delegation vs Implementation Inheritance

♦ Delegation
  ◆ Pro:
    ◆ Flexibility: Any object can be replaced at run time by another one (as long as it has the same type)
  ◆ Con:
    ◆ Inefficiency: Objects are encapsulated.

♦ Inheritance
  ◆ Pro:
    ◆ Straightforward to use
    ◆ Supported by many programming languages
    ◆ Easy to implement new functionality
  ◆ Con:
    ◆ Inheritance exposes a subclass to the details of its parent class
    ◆ Any change in the parent class implementation forces the subclass to change (which requires recompilation of both)
Many design patterns use a combination of inheritance and delegation
Component Selection

❖ Select existing
  ◆ off-the-shelf class libraries
  ◆ frameworks or
  ◆ components

❖ Adjust the class libraries, framework or components
  ◆ Change the API if you have the source code.
  ◆ Use the adapter or bridge pattern if you don’t have access

❖ Architecture Driven Design
Reuse...

Look for existing classes in class libraries
  - JSAPI, JTAPI, ....

Select data structures appropriate to the algorithms
  - Container classes
  - Arrays, lists, queues, stacks, sets, trees, ...

It might be necessary to define new internal classes and operations
  - Complex operations defined in terms of lower-level operations might need new classes and operations
Frameworks

- A framework is a reusable partial application that can be specialized to produce custom applications.
- Frameworks are targeted to particular technologies, such as data processing or cellular communications, or to application domains, such as user interfaces or real-time avionics.
- The key benefits of frameworks are reusability and extensibility.
  - Reusability leverages of the application domain knowledge and prior effort of experienced developers
  - Extensibility is provided by hook methods, which are overwritten by the application to extend the framework.
    - Hook methods systematically decouple the interfaces and behaviors of an application domain from the variations required by an application in a particular context.
Classification of Frameworks

♦ Frameworks can be classified by their position in the software development process.

♦ Frameworks can also be classified by the techniques used to extend them.
  ◆ Whitebox frameworks
  ◆ Blackbox frameworks
Frameworks in the Development Process

- Infrastructure frameworks aim to simplify the software development process
  - System infrastructure frameworks are used internally within a software project and are usually not delivered to a client.
- Middleware frameworks are used to integrate existing distributed applications and components.
  - Examples: MFC, DCOM, Java RMI, WebObjects, WebSphere, WebLogic Enterprise Application [BEA].
- Enterprise application frameworks are application specific and focus on domains
  - Example domains: telecommunications, avionics, environmental modeling, manufacturing, financial engineering, enterprise business activities.
White-box and Black-Box Frameworks

♦ Whitebox frameworks:
  ◆ Extensibility achieved through inheritance and dynamic binding.
  ◆ Existing functionality is extended by subclassing framework base classes and overriding predefined hook methods
  ◆ Often design patterns such as the template method pattern are used to override the hook methods.

♦ Blackbox frameworks
  ◆ Extensibility achieved by defining interfaces for components that can be plugged into the framework.
  ◆ Existing functionality is reused by defining components that conform to a particular interface
  ◆ These components are integrated with the framework via delegation.
Class libraries and Frameworks

♦ Class Libraries:
  ◆ Less domain specific
  ◆ Provide a smaller scope of reuse.
  ◆ Class libraries are passive; no constraint on control flow.

♦ Framework:
  ◆ Classes cooperate for a family of related applications.
  ◆ Frameworks are active; affect the flow of control.

♦ In practice, developers often use both:
  ◆ Frameworks often use class libraries internally to simplify the development of the framework.
  ◆ Framework event handlers use class libraries to perform basic tasks (e.g. string processing, file management, numerical analysis…. )
Components and Frameworks

♦ Components

◆ Self-contained instances of classes
◆ Plugged together to form complete applications.
◆ Blackbox that defines a cohesive set of operations,
◆ Can be used based on the syntax and semantics of the interface.
◆ Components can even be reused on the binary code level.
  † The advantage is that applications do not always have to be recompiled when components change.

♦ Frameworks:

◆ Often used to develop components
◆ Components are often plugged into blackbox frameworks.
Example: Framework for Building Web Applications
Documenting the Object Design: The Object Design Document (ODD)

♦ Object design document
  ◆ Same as the Requirements Analysis Document (RAD) plus...
  ◆ … additions to object, functional and dynamic models (from solution domain)
  ◆ … navigational map for object model
  ◆ … Javadoc documentation for all classes

♦ ODD Management issues
  ◆ Update the system models in the RAD?
  ◆ Should the ODD be a separate document?
  ◆ Who is the target audience for these documents (Customer, developer?)
  ◆ If time is short: Focus on the Navigational Map and Javadoc documentation?

♦ ODD Template:
  ◆ http://www.oose.org
Documenting Object Design: ODD Conventions

- Each subsystem in a system provides a service (see Chapters on System Design)
  - Describes the set of operations provided by the subsystem
- Specifying a service operation as
  - Signature: Name of operation, fully typed parameter list and return type
  - Abstract: Describes the operation
  - Pre: Precondition for calling the operation
  - Post: Postcondition describing important state after the execution of the operation
- Use JavaDoc for the specification of service operations.
JavaDoc

♦ Add documentation comments to the source code.
♦ A doc comment consists of characters between /** and */
♦ When JavaDoc parses a doc comment, leading * characters on each line are discarded. First, blanks and tabs preceding the initial * characters are also discarded.
♦ Doc comments may include HTML tags
♦ Example of a doc comment:

    /**
     * This is a <b> doc </b> comment
     */
More on JavaDoc

♦ Doc comments are only recognized when placed immediately before class, interface, constructor, method or field declarations.

♦ When you embed HTML tags within a doc comment, you should not use heading tags such as <h1> and <h2>, because JavaDoc creates an entire structured document and these structural tags interfere with the formatting of the generated document.

♦ Class and Interface Doc Tags

♦ Constructor and Method Doc Tags
Class and Interface Doc Tags

@author name-text
◆ Creates an “Author” entry.

@version version-text
◆ Creates a “Version” entry.

@see classname
◆ Creates a hyperlink “See Also classname”

@since since-text
◆ Adds a “Since” entry. Usually used to specify that a feature or change exists since the release number of the software specified in the “since-text”

@deprecated deprecated-text
◆ Adds a comment that this method can no longer be used. Convention is to describe method that serves as replacement
◆ Example: @deprecated Replaced by setBounds(int, int, int, int).
Constructor and Method Doc Tags

♦ Can contain @see tag, @since tag, @deprecated as well as:

@param parameter-name description
   Adds a parameter to the "Parameters" section. The description may be continued on the next line.

$return description
   Adds a "Returns" section, which contains the description of the return value.

(exception fully-qualified-class-name description
   Adds a "Throws" section, which contains the name of the exception that may be thrown by the method. The exception is linked to its class documentation.

@see classname
   Adds a hyperlink "See Also" entry to the method.
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 * Window win = new Window(parent);
 * win.show();
 * </pre>
 *
 * @author Sami Shaio
 * @version %I%, %G%
 * @see java.awt.BaseWindow
 * @see java.awt.Button
 */

class Window extends BaseWindow {
    ...
}

Example of a Method Doc Comment

/**
 * Returns the character at the specified index. An index
 * ranges from <code>0</code> to <code>length() - 1</code>.
 * 
 * @param     index  the index of the desired character.
 * @return    the desired character.
 * @exception StringIndexOutOfRangeException
 *              if the index is not in the range <code>0</code>
 *              to <code>length()-1</code>.
 * @see       java.lang.Character#charValue()
 */

public char charAt(int index) {
    ...
}


Example of a Field Doc Comment

♦ A field comment can contain only the @see, @since and @deprecated tags

```java
/**
 * The X-coordinate of the window.
 * @see window#1
 */
int x = 1263732;
```
Example: Specifying a Service in Java

/** Office is a physical structure in a building. It is possible to create an instance of a office; add an occupant; get the name and the number of occupants */

public class Office {
    /** Adds an occupant to the office */
    * @param  NAME  name is a nonempty string */
    public void AddOccupant(string name);

    /** @Return Returns the name of the office. Requires, that Office has been initialized with a name */
    public string GetName();

    ....
}

Package it all up

♦ Pack up design into discrete physical units that can be edited, compiled, linked, reused

♦ Construct physical modules
  ♦ Ideally use one package for each subsystem
  ♦ System decomposition might not be good for implementation.

♦ Two design principles for packaging
  ♦ **Minimize coupling:**
    ♦ Classes in client-supplier relationships are usually loosely coupled
    ♦ Large number of parameters in some methods mean strong coupling (> 4-5)
    ♦ Avoid global data
  ♦ **Maximize cohesion:**
    ♦ Classes closely connected by associations ⇒ same package
Packaging Heuristics

♦ Each subsystem service is made available by one or more interface objects within the package
♦ Start with one interface object for each subsystem service
  ◆ Try to limit the number of interface operations (7±2)
♦ If the subsystem service has too many operations, reconsider the number of interface objects
♦ If you have too many interface objects, reconsider the number of subsystems
♦ Difference between interface objects and Java interfaces
  ◆ Interface object: Used during requirements analysis, system design and object design. Denotes a service or API
  ◆ Java interface: Used during implementation in Java (A Java interface may or may not implement an interface object)