**System design**

- **1 Overview**
- **2 Subsystem decomposition**
- **3 Assessing O-O architectures**
- **4 Architectural styles**
- **5 Advanced issues**

---

**Simplicity**

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  
The Emperor’s Old Clothes  
1980 Turing Award lecture  
http://tinyurl.com/3yk3v2

Tony Hoare

---

**System design: scope**

- Bridge the gap between a problem and an existing system
- Divide and conquer: model new system as a set of subsystems

---

**Goals and tasks**

- Identify design goals
- Design initial subsystem decomposition
- Refine subsystem decomposition to address design goals

**Design goals**

- Qualities to be optimized

**Software architecture**

- Subsystem responsibilities
- Subsystems dependencies
- Subsystem mapping to hardware
- Policy decisions: control flow, access control, data storage...

---

**The “ilities” of software engineering**

- Correctness
- Performance
- Verifiability
- Robustness
- Scalability
- Understandability
- Reusability
- Reliability
- Evolvability
- Usability
- Portability
- Security
- Repairability
- Interoperability
**Typical design trade-offs**

- **Functionality** vs. **Usability**
- **Cost** vs. **Robustness**
- **Performance** vs. **Portability**
- **Rapid development** vs. **Functionality**
- **Cost** vs. **Reusability**
- **Backward Compatibility** vs. **Understandability**

**System design**

- 1 Overview
- 2 Subsystem decomposition
- 3 Assessing O-O architectures
- 4 Architectural styles
- 5 Advanced issues

**Why decompose a system?**

- **Management**
  - Partition effort
  - Clear assignment of requirements to modules
- **Modification**
  - Decouple parts so that changes to one don’t affect others
- **Understanding**
  - Allow understanding system one chunk at a time

**Subsystems**

- Collection of closely interrelated classes
- Deduced from natural groupings in analysis
- **Eiffel**: clusters
- **In UML**: packages
- Other programming languages: modules, packages (Java), or conventions, e.g. directories

**Services and Subsystem Interfaces**

**Traditional distinction:**

- **Service**: Set of related operations
  - Provided by one subsystem
  - Share a common purpose
  - Inputs, outputs & high-level behavior defined in system design
- **Subsystem interface**: Set of fully-typed operations
  - Specifies interaction and information flow from and to subsystem boundaries (not inside subsystem)
  - Refinement of services
  - Defined in detailed design

In object-oriented design, the distinction fades out

**Decomposition Example: Compiler**

- **Lexer**
  - **Service:** Scan input file and provide stream of tokens, Initialize symbol table, Report lexical errors
  - **Features**: next_token(File, ST)

- **Parser**
  - **Service:** Parse token stream and build abstract syntax tree, Enter symbol table information, Report syntax errors
  - **Features**: AST(File, ST)

- **Static Analyzer**
  - **Service:** Perform semantic analysis, Fill symbol table, Report type errors
  - **Features**: perform_analysis(AST, ST)

- **Code Generator**
  - **Service:** Generate target code from analyzed syntax tree
  - **Features**: generate_code(AST, ST)
Cohesion and coupling

- Cohesion: interdependence of elements of one module
- Coupling: interdependence between different modules
- Goal: high cohesion and low coupling

![Diagram showing high and low cohesion and coupling]

Modularity: increase cohesion, decrease coupling

Favored by architectural techniques tending to ensure decentralization of modules

Decomposability

Decompose complex systems into subsystems

COROLLARY: Division of labor:
- Example: Top-down design method (see next).
- Counter-example: General initialization module.

![Diagram showing decomposability]

Top-down functional design

![Diagram showing top-down functional design]

Composability

Build software elements so that they may be freely combined with others to produce new software.

![Diagram showing composability]

Direct mapping

Maintain a close connection between the structure of the design and the structure of the analysis model.
**Few interfaces principle**

Every module communicates with as few others as possible.

(A) (B) (C)

**Small interfaces principle**

If two modules communicate, they exchange as little information as possible.

**Explicit interfaces principle**

Whenever two modules $A$ and $B$ communicate, this is obvious from the text of $A$ or $B$ or both.

**Continuity**

Ensure that small changes in specifications yield small changes in architecture.

*Design method: Specification $\rightarrow$ Architecture*

Example: Principle of Uniform Access (see next)

Counter-example: Programs with patterns after the physical implementation of data structures.

**Uniform Access Principle**

A module's facilities are accessible to its clients in the same way whether implemented by computation or storage.

**Uniform Access: An example**

$$balance = list\_of\_deposits.total - list\_of\_withdrawals.total$$

(A1) $list\_of\_deposits$ $list\_of\_withdrawals$ $balance$

(A2) $list\_of\_deposits$ $list\_of\_withdrawals$

Net uniform access: $a.balance$  Uniform access: $a.balance(a)$
Uniform access principle

It doesn’t matter to the client whether you look up or compute

A call such as

`your_account.balance`

could use an attribute or a function

Information hiding (Parnas, 1972)

Underlying question: how does one “advertise” the capabilities of a module?

Every module should be known to the outside world through an official, “public” interface. The rest of the module’s properties comprises its “secrets”. It should be impossible to access the secrets from the outside.

Information Hiding Principle

The designer of every module must select a subset of the module’s properties as the official information about the module, to be made available to authors of client modules.

Information hiding

Justifications:

- Continuity
- Decomposability

System design

- 1 Overview
- 2 Subsystem decomposition
- 3 Assessing O-O architectures
- 4 Architectural styles
- 5 Advanced issues
Good architecture

- Result of a consistent set of principles and techniques, applied consistently through all phases of a project
- Resilient in the face of (inevitable) changes
- Source of guidance throughout the product lifetime
- Reuse of established engineering knowledge
  - Application of architectural styles
  - Analogous to design patterns in detailed design

The five secrets of good architecture

- Simplicity of design
- Consistency of design
- Ease of learning of the APIs
- Support for change
- Support for reuse

The contribution of object technology

- Single decomposition criterion: ADT
- Precision of specification: contracts
- Clear client-supplier separation, information hiding
- Organize abstractions in hierarchies: inheritance
- Polymorphism and dynamic binding
- Easily add new types
- Parameterize classes: generality
- Abstract behaviors into objects: agents, delegates
- Support for reuse, libraries
- Known, published collections of design patterns

O-O is for high cohesion and low coupling

- **Cohesion**
  - Features work on same data
  - Implement one ADT
- **Low coupling**
  - Small interfaces
  - Information hiding
  - No global data
  - Interactions are within subsystem rather than across subsystem boundaries

The key task in O-O

- Finding the right data abstractions

Judging good and bad architectures

- This is the basis of “refactoring”
- Never take a design for granted
- But: don’t delay good design (GIGO)
Judging good and bad architectures

- Examples:
  - Compiler
  - Math routines
  - Library design: lists
  - Observer pattern
  - Top-down vs O-O: the multi-display panel example
  - Visitor pattern

Cohesion and coupling in compiler example

- Cohesion
  - Each subsystem has a clear responsibility
  - Very high cohesion in compiler

- Coupling
  - Small interfaces between subsystems
  - But: All subsystems read and update the symbol table (global data)
  - Changes of symbol table structure have effect on all subsystems
  - Coupling can be further reduced

Architecture assessment: lists

Original API:
- `l.insert(i, x)`
- `l.remove(i)`
- `pos := l.search(x)`
- `l.insert_by_value(…)`
- `l.insert_by_position(…)`
- `l.search_by_position(…)`

New interface:
- Queries: `l.index`, `l.item`, `l.before`, `l.after`
- Commands: `l.start`, `l.forth`, `l.finish`, `l.back`

A list seen as an active data structure

Typical use:
- `j := l.search(x)`
- `l.insert(j + 1, y)`

**Features**
- Perfect
- Desirable
- ?

Compilers are the subject of the next page.
Architecture assessment: numerical library

Classical, non-O-O library style: NAG

nonlinear_ode

\[
\begin{align*}
\text{equation_count : in INTEGER} \\
\text{epsilon : in out DOUBLE} \\
\text{func : procedure} \\
\text{(eq_count : INTEGER, a : DOUBLE,} \\
\text{eps : DOUBLE b : ARRAY [DOUBLE],} \\
\text{cm : pointer Libtype,} \\
\text{left_count, coupled_count : INTEGER ...)}
\end{align*}
\]

(Altogether 19 arguments, including:
  * 4 in out values;
  * 3 arrays, used both as input and output;
  * 6 functions, each with 6 or 7 arguments, of which 2 or 3 arrays)

O-O equivalent (EiffelMath)

\[
\begin{align*}
\text{... Create e and set-up its values} \\
\text{(other than defaults) ...} \\
\text{e.solve} \\
\text{... Answer available in e.x and e.y ...}
\end{align*}
\]

The key to an O-O numerical library: abstractions such as
\[\text{EQUATION, PROBABILITY\_DISTRIBUTION, INTEGRATABLE\_FUNCTION, INTEGRATOR, RANDOM\_SEQUENCE...}\]

Numerical library example: lessons

Separate the auxiliary from the essential

Turn non-essential arguments into options

Each option is settable through its own command

"Option-operand separation principle"

But: this is a stateful solution (see next)

The statefulness issue

Subsystems can be
  * Stateless (e.g. HTTP)
    - Pro: simpler to program; no synchronization issue
    - Con: all information must be passed to every call, through arguments
  * Stateful (e.g. database, option-operand style, stateful firewalls)
    - Pro: simplicity of API; each call only passes new information
    - Con: client needs to have exclusive access, or supplier needs to maintain list of clients

Architecture assessment: Observer Pattern

PUBLISHER

* attach
  + detach

PUB_1

PUB_2

PUB_3

SUBSCRIBER

* update
  + subscribe
  - unsubscribe

SUB_1

SUB_2

SUB_3

SUB_4

\[\text{Deferred (abstract)}\]
\[\text{Effective (implemented)}\]

Observer pattern

To register itself, a subscriber executes:

\[
\begin{align*}
\text{subscribe (some_publisher)}
\end{align*}
\]

where subscribe is defined in SUBSCRIBER as:

\[
\begin{align*}
\text{subscribe (p: PUBLISHER) as:} \\
\text{-- Make current object observe p.} \\
\text{require} \\
\text{publisher_exists: p /= Void} \\
\text{do} \\
\text{p.attach (Current)} \\
\text{end}
\end{align*}
\]
Attaching an observer

In class `PUBLISHER`:

```java
feature
    attach (s : SUBSCRIBER)
    -- Register s as subscriber to current publisher.
    require
        subscriber_exists : s /= Void
    do
        subscribed.extend (s)
    end
```

Observer pattern

- Subscriber may subscribe to at most one publisher
- May subscribe at most one operation
- Not reusable — must be coded anew for each application

Analysis: this uses the wrong data abstractions

The Event library

Fundamental data abstraction: event type

Simple solution:
- One generic class: `EVENT_TYPE`
- Two features: `publish` and `subscribe`

A publisher:
- Statically, defines an event type
- Dynamically, uses `publish` to publish events

A subscriber:
- Subscribes an agent to an event type
- That's all!

Publish-subscribe example lessons

Initial solution:
- Direct coupling between publishers and subscribers
- Partly wrong abstractions: subscriber (observer)

Revised solution: relies on single, directly adapted abstraction (event type); no direct coupling between publishers and subscribers

Architecture assessment: panel-driven system

Flight sought from: Santa Barbara To: Zurich
Depart no earlier than: 18 Mar 2006 No later than: 18 Mar 2006

ERROR: Choose a date in the future

Choose next action:
- 0 – Exit
- 1 – Help
- 2 – Further enquiry
- 3 – Reserve a seat

The transition diagram
Top-down system architecture

execute_session
  -- Execute full session
  local current_state, choice : INTEGER
  do
    current_state := initial
    repeat
      choice := execute_state (current_state)
      current_state := transition (current_state, choice)
    until is_final (current_state)
  end
end

Actions in a state

execute_state (current_state : INTEGER): INTEGER
  -- Execute actions for current_state; return user's exit choice.
  local answer : ANSWER
  good : BOOLEAN
  choice : INTEGER
  do
    repeat
      display (current_state)
      [answer, choice] := read (current_state)
      good := correct (current_state, answer)
      if not good then message (current_state, answer) end
    until good
  end
  process (current_state, answer)
  return choice
end

Criticism

How amenable is this solution to change and adaptation?
  ¾ New transition?
  ¾ New state?
  ¾ New application?

Routine signatures:

execute_state (state : INTEGER): INTEGER
display (state : INTEGER)
read (state : INTEGER): [ANSWER, INTEGER]
correct (state : INTEGER; a: ANSWER): BOOLEAN
message (state : INTEGER; a: ANSWER)
process (state : INTEGER; a: ANSWER)
is_final (state : INTEGER)

Data transmission

All routines share the state as input argument. They must discriminate on it, e.g.:

display (current_state: INTEGER) is
  do
    inspect current_state
    when state1 then ...
    when state2 then ...
    when state3 then ...
  end
end

Consequences:
  ¾ Long and complicated routines.
  ¾ Must know about one possibly complex application.
  ¾ To change one transition, or add a state, need to change all.

The visible architecture

execute_session
  -- Execute full session
  local current_state, choice : INTEGER
  do
    current_state := initial
    repeat
      choice := execute_state (current_state)
      current_state := transition (current_state, choice)
    until is_final (current_state)
  end
end

Level 3
  execute_session

Level 2
  initial transition execute_state is_final
  display read correct message process

Level 1
  display read correct message process
The real story

Grouping by data abstractions

Class STATE

1. Use STATE as the basic abstract data type (and class).
2. Among features of every state:
   a. The routines of level 1 (deferred in class STATE)
   b. execute_state, as above but without the argument current_state
Class structure

STATE

- execute_state
- display
- read
- correct
- message
- process

- INITIAL
- LIGHT_QUEUE
- RESERVATION

Grouping by data abstractions

APPLICATION

- Level 3
  - execute_session

- Level 2
  - initial
  - transition
  - execute_state
  - is_final

- Level 1
  - display
  - read
  - correct
  - message
  - process

To build an application

Necessary states — instances of STATE — should be available.

Initialize application:

create a.make (state_count, choice_count)

Assign a number to every relevant state s:

a.put_state (s, n)

Choose initial state n0:

a.choose_initial (n0)

Enter transitions:

a.put_transition (sou, tar, lab)

May now run:

a.execute_session

Architecture assessment: panel-driven system

- Analyze data transmission
  - Data elements transmitted too far into the structure are usually the sign of an unrecognized abstraction
- Key to openness of last solution: architecture based on types of the problem’s objects (state, transition graph, application)
  - Ignore “the function” of the system. Usually a superficial property, subject to change. Systems usually don’t have a functional “top”
- Keep system open for evolution
- Key is search for data abstraction

Architecture assessment: use of contracts

Describing active structures properly: can after also be before?

Symmetry:

<table>
<thead>
<tr>
<th>start</th>
<th>finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>forth</td>
<td>back</td>
</tr>
<tr>
<td>after</td>
<td>before</td>
</tr>
</tbody>
</table>

For symmetry and consistency, it is desirable to have the invariant properties:

A after = (index = count + 1)
before = (index = 0)
Designing for consistency

Typical iteration:

```
from start until after loop some_action(item) forth
```

Conventions for an empty structure?

- after must be true for the iteration.
- For symmetry: before should be true too.

But this does not work for an empty structure (count = 0; see invariant A): should index be 0 or 1?

To obtain a consistent convention we may transform the invariant into:

```
after = (is_empty or (index = count + 1))
before = (is_empty or (index = 0))
```

-- Hence: is_empty = (before and after)

Symmetric but unpleasant. Leads to frequent tests

```
if after and not is_empty then ...
```

instead of just

```
if after then ...
```

Introducing sentinel items

Invariant (partial):

```
0 <= index <= count + 1
index <= count + 1
before = (index = 0)
```

```
A
after = (index = count + 1)
not (after and before)
```

Valid cursor positions

The case of an empty structure

```
0 before not after
```

```
1 (i.e. count + 1) after not before
```

Valid cursor positions

List structure example: lessons

General principles:

- Consistency
  - A posteriori: "How do I make this design decision compatible with the previous ones?"
  - A priori: "How do I take this design decision so that it will be easy or at least possible to make future ones compatible with it?"
- Use assertions, especially invariants, to clarify the issues
- Importance of symmetry concerns (cf. physics and mathematics)
- Importance of limit cases (empty or full structures)

System design

- 1 Overview
- 2 Subsystem decomposition
- 3 Assessing O-O architectures
- 4 Architectural styles
- 5 Advanced issues
Elements of a Software Architecture

- **Subsystems (components)**
  - Computational units with specified interface
  - Examples: filters, databases, layers, objects

- **Connectors**
  - Interactions between components
  - Examples: routine calls, pipes, event broadcasts, shared data


Architectural Styles

- **Data flow systems**
  - Batch sequential, pipe-and-filter

- **Call-and-return system**
  - Main program and subroutine

- **Independent components**
  - Interacting processes, event system

- **Data-centered systems (repositories)**
  - Databases, blackboards

- **Hierarchical systems**
  - Layers
  - Interpreters, rule-based systems

- **Client-server systems**

- **Peer-to-peer systems**

Data Flow Systems

- The availability of data controls the computation
- The structure is determined by the orderly motion of data from component to component
- Data flow is the only form of communication between components

- **Variations**
  - How control is exerted (e.g., push versus pull)
  - Degree of concurrency between processes
  - Topology

Data Flow Systems

- **Components: data flow components**
  - Interfaces are input and output ports
  - Input ports read data; output ports write data
  - Computational model: read data from input ports, compute, write data to output ports

- **Connectors: data streams**
  - Uni-directional
  - Usually asynchronous, buffered
  - Computational model: transport data from writer to reader

Batch Sequential Style

- **Components are independent programs**
- **Connectors are some type of media**
- **Each step runs to completion before next step begins**

Batch Sequential Style: Properties

- **History**: Mainframes and magnetic tape
- **Applications**: Business data processing
  - Discrete transactions of predetermined type and occurring at periodic intervals
  - Creation of periodic reports based on periodic data updates

- **Examples**
  - Payroll computations
  - Tax reports
Pipe-and-Filter Style

- Components (Filters)
  - Read streams of input data
  - Locally transform input data
  - Produce streams of output data

- Connectors (Pipes)
  - Streams, e.g., first-in-first-out buffer

Pipe-and-Filter Style: Properties

- Data is processed incrementally as it arrives
- Output usually begins before input is consumed
- Filters must be independent, no shared state
- Filters don’t know upstream or downstream filters

Examples
- lex/yacc-based compiler (scan, parse, generate code, …)
- Unix pipes
- Image / signal processing

Push Pipeline with Active Source

- Source of each pipe pushes data downstream
- Example: Unix pipes: grep pattern * | wc

Pull Pipeline with Active Sink

- Sink of each pipe pulls data upstream
- Example: Compiler: t := lexer.next_token

Mixed Pipeline With Passive Source and Sink

- If more than one filter is pushing / pulling, synchronization is needed

Pipe-and-Filter Style: Discussion

- Strengths
  - Reuse: any two filters can be connected if they agree on that data format that is transmitted
  - Ease of maintenance: filters can be added or replaced
  - Potential for parallelism: filters implemented as separate tasks, consuming and producing data incrementally

- Weaknesses
  - Sharing global data is expensive or limiting
  - Can be difficult to design incremental filters
  - Not appropriate for interactive applications
  - Error handling is Achilles heel, e.g., some intermediate filter crashes
  - Often lowest common denominator on data transmission, e.g., ASCII in Unix pipes
Call-and-Return Style (Explicit Invocation)

- **Components:** Objects
- **Connections:** Messages (routine invocations)
- **Key aspects**
  - Object preserves integrity of representation (encapsulation)
  - Representation is hidden from client objects
- **Variations**
  - Objects as concurrent tasks

Call-and-Return Style: Discussion

- **Strengths**
  - Change implementation without affecting clients
  - Can break problems into interacting agents (distributed across multiple machines / networks)
- **Weaknesses**
  - Objects must know their interaction partners (in contrast to Pipe-and-Filter)
  - When partner changes, objects that explicitly invoke it must change
  - Side effects: if A uses B and C uses B, then C’s effects on B can be unexpected to A

Event-Based Style (Implicit Invocation)

- **Characterized by the style of communication between components**
  - Component announces (broadcasts) one or more events
- **Generalized Observer Design Pattern**
- **Components**
  - May announce events
  - May register for events of other components with a callback
- **Connectors**
  - Bindings between event announcements and routine calls (callbacks)

Event-Based Style: Properties

- Announcers of events do not know which components will be affected by those events
- Components cannot make assumptions about ordering of processing, or what processing will occur as a result of their events
- **Examples**
  - Programming environment tool integration
  - User interfaces (Model-View-Controller)
  - Syntax-directed editors to support incremental semantic checking

Event-Based Style: Example

- Integrating tools in a shared environment
- Editor announces it has finished editing a module
  - Compiler registers for such announcements and automatically re-compiles module
  - Editor shows syntax errors reported by compiler
- Debugger announces it has reached a breakpoint
  - Editor registers for such announcements and automatically scrolls to relevant source line

Event-Based Style: Discussion

- **Strengths**
  - Strong support for reuse: plug in new components by registering it for events
  - Maintenance: add and replace components with minimum effect on other components in the system
- **Weaknesses**
  - Loss of control
    - What components will respond to an event?
    - In which order will components be invoked?
    - Are invoked components finished?
  - Ensuring correctness is difficult because it depends on context in which invoked
- **In practice, call-and-return style and event-based style are combined**
Data-Centered Style (Repository Style)

- **Components**
  - Central data store component represents system state
  - Independent components operate on the data store

Special Case: Blackboard Architectures

- **Interactions among knowledge sources solely through repository**
- **Knowledge sources make changes to the shared data that lead incrementally to solution**
- **Control is driven entirely by the state of the blackboard**

  - Example
    - Repository: modern compilers act on shared data: symbol table, abstract syntax tree
    - Blackboard: signal and speech processing

Data-Centered Style: Discussion

- **Strengths**
  - Efficient way to share large amounts of data
  - Data integrity localized to repository module

- **Weaknesses**
  - Subsystems must agree (i.e., compromise) on a repository data model
  - Schema evolution is difficult and expensive
  - Distribution can be a problem

Hierarchical Style (Layered Style)

- **Components**
  - Group of subtasks which implement an abstraction at some layer in the hierarchy

- **Connectors**
  - Protocols that define how the layers interact

  - Example
    - Communication protocols
    - Operating systems

Hierarchical Style: Properties

- Each layer provides service to the layer above it and acts as a client of the layer below
- Each layer collects services at a particular level of abstraction
- A layer depends only on lower layers
  - Has no knowledge of higher layers

  - Example
    - Communication protocols
    - Operating systems

Hierarchical Style: Example

- **THE operating system (Dijkstra)**
- **The OSI Networking Model**
  - Each level supports communication at a level of abstraction
  - Protocol specifies behavior at each level of abstraction
  - Each layer deals with specific level of communication and uses services of the next lower level

  - Layers can be exchanged
    - Example: Token Ring for Ethernet on Data Link Layer
**OSI Model Layers and Their Responsibilities**

- Application
- Presentation
- Session
- Transport
- Network
- Data Link
- Physical

- The system you are designing
- Performs data transformation services, such as byte swapping and encryption
- Initializes a connection, including authentication
- Reliably transmits messages
- Transmits and routes data within the network
- Sends and receives frames without error
- Sends and receives bits over a channel

**Hierarchical Style: Example (cont’d)**

**Hierarchical Style: Discussion**

- **Strengths**
  - Increasing levels of abstraction as we move up through layers: partitions complex problems
  - Maintenance: in theory, a layer only interacts with layer below (low coupling)
  - Reuse: different implementations of the same level can be interchanged

- **Weaknesses**
  - Performance: communicating down through layers and back up, hence bypassing may occur for efficiency reasons

**Interpreters**

- Architecture is based on a virtual machine produced in software
- Special kind of a layered architecture where a layer is implemented as a true language interpreter
- Components
  - "Program" being executed and its data
  - Interpretation engine and its state
- Example: Java Virtual Machine
  - Java code translated to platform independent bytecode
  - JVM is platform specific and interprets the bytecode

**Client Server Style**

- **Components**
  - Subsystems are independent processes
  - Servers provide specific services such as printing, etc.
  - Clients use these services
- **Connectors**
  - Data streams, typically over a communication network

**Client Server Style Example: Databases**

- **Front-end: User application (client)**
  - Customized user interface
  - Front-end processing of data
  - Initiation of server remote procedure calls
  - Access to database server across the network
- **Back-end: Database access and manipulation (server)**
  - Centralized data management
  - Data integrity and database consistency
  - Database security
  - Concurrent operations (multiple user access)
  - Centralized processing (for example archiving)
Client Server Style: Variants

- **Thick / fat client**
  - Does as much processing as possible
  - Passes only data required for communications and archival storage to the server
  - Advantages: less network bandwidth, fewer server requirements

- **Thin client**
  - Has little or no application logic
  - Depends primarily on the server for processing activities
  - Advantages: lower IT admin costs, easier to secure, lower hardware costs

Client Server Style: Discussion

- **Strengths**
  - Makes effective use of networked systems
  - May allow for cheaper hardware
  - Easy to add new servers or upgrade existing servers
  - Availability (redundancy) may be straightforward

- **Weaknesses**
  - Data interchange can be hampered by different data layouts
  - Communication may be expensive
  - Data integrity functionality must be implemented for each server
  - Single point of failure

Peer-to-Peer Style

- **Similar to client-server style, but each component is both client and server**
- **Pure peer-to-peer style**
  - No central server, no central router
- **Hybrid peer-to-peer style**
  - Central server keeps information on peers and responds to requests for that information

- **Examples**
  - File sharing applications, e.g., Napster, Gnutella, Kazaa
  - Communication and collaboration, e.g., Skype

Peer-to-Peer: Discussion

- **Strengths**
  - Efficiency
    - All clients provide resources
  - Scalability
    - System capacity grows with number of clients
  - Robustness
    - Data is replicated over peers
    - No single point of failure in the system (in pure peer-to-peer style)

- **Weaknesses**
  - Architectural complexity
  - Resources are distributed and not always available
  - More demanding of peers (compared to client-server)
  - New technology not fully understood

Assessing architectures

- **General style can be discussed ahead of time**
- **Know pros and cons**
- **Architectural styles → Patterns → Components**

System design

- **1 Overview**
- **2 Subsystem decomposition**
- **3 Assessing O-O architectures**
- **4 Architectural styles**
- **5 Advanced issues**
Areas of System Design: Specific Issues

- Identify design goals
- Design initial subsystem decomposition
- Refine subsystem decomposition to address design goals

Concurrency
- Execution threads are sequences of atomic actions during a program execution
- Concurrent programs can have more than one thread
- Execution of threads can be parallel (on several processors) or virtually parallel (on one processor)
- Design goal: response time, performance

Concurrency Questions
- Which objects of the object model are independent?
  - Candidates for separate threads
- Does the system support multiple users?
  - Example: Client-server architecture with several clients
- Can a single request to the system be decomposed into multiple requests? Can these requests be handled in parallel?
  - Search in a distributed database
  - Image recognition by decomposing the image into stripes

Hardware / Software Mapping
- This activity addresses two questions:
  - How shall we realize the subsystems: with hardware or with software?
  - How do we map the object model on the chosen hardware and software?
- Much of the difficulty of designing a system comes from meeting externally-imposed hardware and software constraints

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?

Mapping the Objects (cont'd)
- Example: stock trading
  - Usually steady rate of stock orders per day
  - Extreme peaks for important IPOs
- Bank is liable for loss of orders
  - System must be able to handle peak load
Mapping the Associations

- 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?

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?
- General system performance question
  - What is the desired response time?

Example: ATM Machine and Host System

- Server software runs on workstations; one per region
- Backend software runs on mainframe; one for the whole country
- Client software runs on common PC; one PC per ATM
- Connected via backbone
- Connected via leased line (low latency)

Data Management

- Some objects in the models need to be persistent
- Persistency is achieved by files and databases
- 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?
  - Is 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?
- When should you choose a database?
  - Does the data require access 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 (e.g., indexing, transactions)?

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
Object-Oriented Databases

- An object-oriented database supports all the fundamental object modeling concepts
  - Classes, Attributes, Routines, Associations, Inheritance
- Mapping an object model to an OO-database
  - Determine which objects are persistent
  - Perfrom normal requirement analysis and detailed design
  - Do the mapping specific to commercially available product
- Suitable for medium-sized data set, irregular associations among objects

Relational Databases

- Data is presented as two-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
  - Foreign key: Reference to a primary key in another table
- SQL is the standard language for defining and manipulating tables
- Suitable for large data set, complex queries over attributes

Mapping an Object Model to a Relational DB

- UML object models can be mapped to relational databases
- UML mappings
  - Each class is mapped to a table
  - Each class attribute is mapped onto a column in the table
  - An instance of a class represents a row in the table
  - A one-to-many association is implemented as foreign key
  - A many-to-many association is mapped into its own table
- Methods are not mapped

Mapping 1:n and n:1 Associations

- Buried Foreign Keys

```
<table>
<thead>
<tr>
<th>Transaction</th>
<th>Portfolio</th>
</tr>
</thead>
<tbody>
<tr>
<td>transactionID</td>
<td>portfolioID</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>TransactionTable</th>
<th>PortfolioTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>transactionID</td>
<td>portfolioID</td>
</tr>
<tr>
<td>Primary Key</td>
<td>Primary Key</td>
</tr>
</tbody>
</table>
```

Mapping Many-to-Many Associations

- Separate table for association

```
<table>
<thead>
<tr>
<th>City</th>
<th>Airport</th>
</tr>
</thead>
<tbody>
<tr>
<td>cityName</td>
<td>airportCode</td>
</tr>
<tr>
<td>Houston</td>
<td>IAH</td>
</tr>
<tr>
<td>Albany</td>
<td>HOU</td>
</tr>
<tr>
<td>Munich</td>
<td>MUC</td>
</tr>
<tr>
<td>Hamburg</td>
<td>HAM</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>CityTable</th>
<th>AirportTable</th>
<th>ServesTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>cityName</td>
<td>airportCode</td>
<td>airportName</td>
</tr>
<tr>
<td>Houston</td>
<td>IAH</td>
<td>Intercontinental</td>
</tr>
<tr>
<td>Albany</td>
<td>HOU</td>
<td>Hobby</td>
</tr>
<tr>
<td>Munich</td>
<td>MUC</td>
<td>Munich Airport</td>
</tr>
<tr>
<td>Hamburg</td>
<td>HAM</td>
<td>Hamburg Airport</td>
</tr>
</tbody>
</table>
```

Mapping Inheritance

- Option 1: separate table

```
<table>
<thead>
<tr>
<th>Person</th>
<th>Student</th>
<th>Assistant</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>legi</td>
<td>office</td>
</tr>
<tr>
<td>56</td>
<td>123456</td>
<td>RZ F02</td>
</tr>
<tr>
<td>56</td>
<td>Urs</td>
<td>Sile</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>StudentTable</th>
<th>AssistantTable</th>
<th>PersonTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>name</td>
<td>legi</td>
</tr>
<tr>
<td>id</td>
<td>name</td>
<td>legi</td>
</tr>
<tr>
<td>56</td>
<td>Urs</td>
<td>123456</td>
</tr>
<tr>
<td>79</td>
<td>Sile</td>
<td>123456</td>
</tr>
</tbody>
</table>
```
Mapping Inheritance (cont'd)

- Option 2: duplicating columns

<table>
<thead>
<tr>
<th>Person</th>
<th>Student</th>
<th>Assistant</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
<td>name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StudentTable</th>
<th>AssistantTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>name</td>
</tr>
<tr>
<td>123</td>
<td>Urs</td>
</tr>
</tbody>
</table>

Separate Tables vs. Duplicated Columns

- Trade-off between modifiability and response time
  - How likely is a change of the superclass?
  - What are the performance requirements for queries?
- Separate table mapping
  - Pro: Adding attributes to the superclass is easy (adding a column to the superclass table)
  - Con: Searching for the attributes of an object requires a join operation
- Duplicated columns
  - Con: Modifying the database schema is more complex and error-prone
  - Pro: Individual objects are not fragmented across a number of tables (faster queries)

Data Management Questions

- Should the data be distributed?
- Should the database be extensible?
- How often is the database accessed?
- What is the expected request rate? In the worst case?
- What is the size of typical and worst case requests?
- Does 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?

Boundary Conditions

- Most of the system design effort is concerned with the steady-state behavior described in the analysis phase
- Additional administration use cases describe:
  - Initialization ("startup use cases")
  - Termination ("termination use cases")
    - What resources are cleaned up and which systems are notified upon termination
  - Failure ("failure use cases")
    - Many possible causes: Bugs, errors, external problems
    - Good system design foresees fatal failures

Boundary Condition Questions

- Initialization
  - How does the system start up?
  - What data needs 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?
- Termination
  - Are single subsystems allowed to terminate?
  - Are other subsystems notified if a single subsystem terminates?
  - How are local updates communicated to the database?
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

Influences from Requirements Analysis

- Nonfunctional Requirements
- Definition of Design Goals
- Functional model
- Subsystem Decomposition
- Object model
- Hardware/software Mapping, Data Management
- Dynamic model
- Identification of Concurrency
- Finally: The subsystem decomposition influences boundary conditions

Summary: System Design

- Design goals definition
  - Describes and prioritizes the qualities that are important for the system
- 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