System Design

What?
Why?
How?

Conquering Complexity is a Challenge

OMT Methodology

Object Analysis

Problem Statement
Object Model
Dynamic Model
Functional Model

Object Design

System Design

Object Design

Implementation

precise, concise, understandable

correct model of the world

GIGO

partitioning the system
System (Architectural) Design

- From problem definition to (high-level) solution definition

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\hline
\text{Problem} & \ldots & \text{n} \\
\hline
\end{array}
\]

Design Space

\[Design\ is\ essentially\ a\ decision-making\ process\]

Why?

An Architecture Analogy

[Mowbray & Zahavi]

Messy closet

Clean closet

- easy to use
- easy to maintain
- accommodate more items
- flexible

Anything else?
assume a sw system is composed of an arbitrary collection of modules, each with a series of versions

an interface is the link between
the server module that provides a service and
the client module that uses the service

* Recall: data flow  * Recall: operation/service

then, a **system model** is a complete and detailed description of the client/server relationships in a sw system at a given point

a sw system is "consistently composed" if for every client/server relationship, the client & the server agree on the interface between them

**What if algorithms change?**

for every client/server relationship, the system model must specify the version of (the interface, the server, the client)

names
Types of decisions

Partitioning the system into subsystems

- A subsystem is a package of classes, associations, operations, events, and constraints
- A subsystem has a reasonably well-identified interface
- A subsystem can in turn be decomposed into smaller subsystems
- Each lowest level subsystem is called “module” in OMT

Identifying concurrency inherent in the problem

- To achieve as much independence as possible
- The dynamic model can be the guide
  e.g., two objects receiving events at the same time

Allocating subsystems to processors and tasks

- Each concurrent subsystem is allocated to (a) hardware (or software)
  - hardware-software tradeoff
- The connectivity of subsystems needs to be determined

Architectural Styles

Consider patterns of interaction
(e.g., procedure call, external files, message passing, sockets, RPCs, MOMs, etc)

Consider system style, subsystem style, homogeneity, etc.
(e.g., 4+2 layer, first layer being pipe-and-filter, second layer OO, etc.)
A specification for a standard OO architecture for applications
- not a low-level design/implementation
- platform (OS, HW)-independence, PL-independence
- defined by the Object Management Group (OMG) since Nov 1990
  currently >500 members
- CORBA clients and servers do not need direct knowledge of each other
  the broker knows the locations and capabilities of the servers on the network
- A client request can be fulfilled by several (competing) servers
  the broker should know who can provide the service fastest and cheapest
- An Object Model requires abstraction, encapsulation, inheritance & polymorphism

"The ability to create simplifying abstractions is a key innate talent of the software architect. Few individuals practicing in the software industry have this ability - perhaps as few as one in five software designers." [Coplien, '94]
Object Design:

■ **Combine the 3 models**

When a person gets hired, s/he is given an office
s/he is recorded as an employee
s/he participates in projects

When the company changes her salary, a meeting is called for
Afterwards, the company informs the payroll of the change

... **assuming allocation of functions have been done**

When Sue gets hired, Sue.office <- FN2.106
instanceOf (Sue, Employee)
Sue.project <- {SuperBanking, MAN, GPS}

When the company changes her salary,

initiate -------> get constraints -------> request for
mtgProposal        schedule

Afterwards, the company informs the payroll of the change

---

obtain operations on classes

---

Lawrence Chung

Object Design:

■ **Design data structures and algorithms**

When a person gets hired, s/he is given an office
s/he is recorded as an employee
s/he participates in projects

When the company changes her salary, a meeting is called for
Afterwards, the company informs the payroll of the change

... **assuming allocation of functions have been done**

When Sue gets hired, Sue.office <- FN2.106
instanceOf (Sue, Employee)
Sue.project <- {SuperBanking, MAN, GPS}

When the company changes her salary,

initiate -------> get constraints -------> request for
mtgProposal        schedule

Afterwards, the company informs the payroll of the change

---

Searching
Sorting
Graph

---

Lawrence Chung
### Optimization

- **Adding redundant qualifier for efficient access**

\[
\text{Company} \xrightarrow{\text{Employs}} \text{Person} \xrightarrow{\text{Has-skill}} \text{Skill}
\]

Company::find-skill (Japanese) -> |Person|, |Person| x |Skill| iterations

\[
\text{Company} \xrightarrow{\text{language}} \text{Speaks language} \xrightarrow{\text{Person}}
\]

- **Saving derived attributes to avoid recomputation**

<table>
<thead>
<tr>
<th>Person</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>salary</td>
<td>salary</td>
</tr>
<tr>
<td>/salary-position</td>
<td>/salary-position</td>
</tr>
</tbody>
</table>

### Implementation of control

1. **State -> Location**

   - Start
   - Password entered
   - do: verify password
   - Password entered and verify password
   - until password OK

   repeat

   - read password
   - verify password
   - password OK

2. **Assume State Machine Engine Exists**

<table>
<thead>
<tr>
<th>Start</th>
<th>enter password</th>
<th>password OK</th>
<th>password NOK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password entered</td>
<td>verify password</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Password entered</td>
<td></td>
<td></td>
<td>Start</td>
</tr>
</tbody>
</table>

3. **Object -> Task (-> Concurrent Processes)**

   - High overhead
   - Major OOPLs do not support concurrency
   - -> no such thing as message passing
**Object Design:**

### Adjustment of inheritance

- **Generalize**
  - Person
  - Cat
  - Animal

- **Specialize**

- **Use delegation to share implementation**

<table>
<thead>
<tr>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
</tr>
<tr>
<td>remove</td>
</tr>
<tr>
<td>first</td>
</tr>
<tr>
<td>last</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
</tr>
<tr>
<td>pop</td>
</tr>
</tbody>
</table>

The body of each stack is a list

- `self.push(e) = self.add(e)` s.t. `self.first() = e`
- `self.pop() = self.remove()` s.t. `self.first() = ???`

### Design of associations

(in requirements, usually two-way; in design, efficiency)

- **One-way associations**

- **Two-way associations**

  1. use one-way: backward search expensive

  2. use one-way + set-valued attribute

  3. as distinct object

- **Link attributes**
  - Exercise!
Object Design:

- Packaging

  REVIEW Software Engineering

  - Information hiding
    private vs. public
  - A chunk of classes as a module
  - ??? cohesion
  - ??? coupling

Document decision decisions!!!