Lecture 22: Moving into Design

- Analysis vs. Design
  - Why the distinction?
- Design Processes
  - Logical vs. Physical Design
  - System vs. Detailed Design
- Architectures
  - System Architecture
  - Software Architecture
  - Architectural Patterns (next lecture)
- Useful Notation
  - UML Packages and Dependencies

Refresher: Lifecycle models

Waterfall model
- Perceived need
  - Requirements
  - Architecture
  - High level design
  - Code
  - Integration
  - Unit test
  - Integrate
  - Maintain

V model
- System requirements
- Software requirements
- Preliminary design
- Detailed design
- Code and design
- Component test
- Integration test
- System test
- Acceptance test
- Test and integrate
- Time
- Evolutionary development
  (each version incorporates new requirements)
- Spiral model
  - Evaluate alternatives and risks
  - Plan
  - Develop and test
  - Risk analysis
  - Integration and test plan
  - Implementation
  - Post-implementation
  - Lessons learnt
- Version 1
  - Requirements
  - Design
  - Code
  - Test
  - Integration
  - O&M
- Version 2
  - Requirements
  - Design
  - Code
  - Test
  - Integration
  - O&M

© Easterbrook 2004
Analysis vs. Design

- **Analysis**
  - Asks "what is the problem?"
    - what happens in the current system?
    - what is required in the new system?
  - Results in a detailed understanding of:
    - Requirements
    - Domain Properties
  - Focuses on the way human activities are conducted

- **Design**
  - Investigates "how to build a solution"
    - How will the new system work?
    - How can we solve the problem that the analysis identified?
  - Results in a solution to the problem
    - A working system that satisfies the requirements
    - Hardware + Software + Peopleware
  - Focuses on building technical solutions

- Separate activities, but not necessarily sequential

---

Refresher: different worlds

Analysis is all about studying this world

Design is all about building this world

Application Domain

- Domain properties
- R - requirements

Machine Domain

- Specification
- C - computer
- P - program

But who builds the bridge?
Four design philosophies

Decomposition & Synthesis
Drivers: Managing complexity, Reuse
Example: Design a car by designing separately the chassis, engine, drivetrain, etc. Use existing components where possible

Search
Drivers: Transformation, Heuristic Evaluation
Example: Design a car by transforming an initial rough design to get closer and closer to what is desired

Negotiation
Drivers: Stakeholder Conflicts, Dialogue Process
Example: Design a car by getting each stakeholder to suggest (partial) designs, and then compare and discuss them

Situated Design
Drivers: Errors in existing designs, Evolutionary Change
Example: Design a car by observing what's wrong with existing cars as they are used, and identifying improvements

Logical vs. Physical Design

Logical Design concerns:
- Anything that is platform-independent:
  - Interactions between objects
  - Layouts of user interfaces
  - Nature of commands/data passed between subsystems
- Logical designs are usually portable to different platforms

Physical Design concerns:
- Anything that depends on the choice of platform:
  - Distribution of objects/services over networked nodes
  - Choice of programming language and development environment
  - Use of specialized device drivers
  - Choice of database and server technology
  - Services provided by middleware
System Design vs. Detailed Design

**System Design**
- Choose a System Architecture
  - Networking infrastructure
  - Major computing platforms
  - Roles of each node (e.g. client-server; clients-broker-servers; peer-to-peer, ...)
- Choose a Software Architecture
  - (see next lecture for details)
- Identify the subsystems
- Identify the components and connectors between them
  - Design for modularity to maximize testability and evolveability
  - E.g. Aim for low coupling and high cohesion

**Detailed Design**
- Decide on the formats for data storage
  - E.g. design a data management layer
- Design the control functions for each component
  - E.g. design an application logic layer
- Design the user interfaces
  - E.g. design a presentation layer

Global System Architecture

**Choices:**
- Allocates users and other external systems to each node
- Identify appropriate network topology and technologies
- Identify appropriate computing platform for each node

**Example:**
- See next slide...
System Architecture Questions

Key questions for choosing platforms:

- What hardware resources are needed?
  - CPU, memory size, memory bandwidth, I/O, disk space, etc.
- What software/OS resources are needed?
  - application availability, OS scalability
- What networking resources are needed?
  - network bandwidth, latency, remote access.
- What human resources are needed?
  - OS expertise, hardware expertise,
  - system administration requirements,
  - user training/help desk requirements.
- What other needs are there?
  - security, reliability, disaster recovery, uptime requirements.

Key questions constraining the choice:

- What funding is available?
- What resources are already available?
  - Existing hardware, software, networking
  - Existing staff and their expertise
  - Existing relationships with vendors, resellers, etc.
Data Management Questions

How is data entry performed?

E.g. Keyless Data entry
- bar codes; Optical Character Recognition (OCR)
- E.g. Import from other systems
- Electronic Data Interchange (EDI), Data interchange languages,

What kinds of data persistence is needed?

Is the operating system’s basic file management sufficient?
- Is object persistence important?
- Can we isolate persistence mechanisms from the applications?

Is a Database Management System (DBMS) needed?

Is data accessed at a fine level of detail
- E.g. do users need a query language?
- Is sophisticated indexing required?
- Is there a need to move complex data across multiple platforms?
- Will a data interchange language suffice?
- E.g. HTML, SGML, XML
- Is there a need to access the data from multiple platforms?

Software Architecture

A software architecture defines:

- the components of the software system
- how the components use each other’s functionality and data
- How control is managed between the components

An example: client-server

- Servers provide some kind of service; clients request and use services
- Applications are located with clients
- E.g. running on PCs and workstations;
- Data storage is treated as a server
  - E.g. using a DBMS such as DB2, Ingres, Sybase or Oracle
  - Consistency checking is located with the server
- Advantages:
  - Breaks the system into manageable components
  - Makes the control and data persistence mechanisms clearer
- Variants:
  - Thick clients have their own services, thin ones get everything from servers
- Note: This is a SOFTWARE architecture
  - Clients and server could be on the same machine or different machines...
## Coupling

Given two units (e.g. methods, classes, modules, ...), A and B:

<table>
<thead>
<tr>
<th>Form</th>
<th>Features</th>
<th>Desirability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data coupling</strong></td>
<td>A &amp; B communicate by simple data only</td>
<td><strong>High</strong> (use parameter passing &amp; only pass necessary info)</td>
</tr>
<tr>
<td><strong>Stamp coupling</strong></td>
<td>A &amp; B use a common type of data</td>
<td><strong>Okay</strong> (but should they be grouped in a data abstraction?)</td>
</tr>
<tr>
<td><strong>Control coupling</strong></td>
<td>A transfers control to B by procedure call</td>
<td><strong>Necessary</strong></td>
</tr>
<tr>
<td>(activating)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Control coupling</strong></td>
<td>A passes a flag to B to tell it how to behave</td>
<td><strong>Undesirable</strong> (why should A interfere like this?)</td>
</tr>
<tr>
<td>(switching)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Common environment</strong></td>
<td>A &amp; B make use of a shared data area (global variables)</td>
<td><strong>Undesirable</strong> (if you change the shared data, you have to change both A and B)</td>
</tr>
<tr>
<td><strong>coupling</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Content coupling</strong></td>
<td>A changes B's data, or passes control to the middle of B</td>
<td><strong>Extremely Foolish</strong> (almost impossible to debug)</td>
</tr>
</tbody>
</table>

## Cohesion

How well do the contents of an object (module, package,...) go together?

<table>
<thead>
<tr>
<th>Form</th>
<th>Features</th>
<th>Desirability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data cohesion</strong></td>
<td>all part of a well defined data abstraction</td>
<td><strong>Very High</strong></td>
</tr>
<tr>
<td><strong>Functional cohesion</strong></td>
<td>all part of a single problem solving task</td>
<td><strong>High</strong></td>
</tr>
<tr>
<td><strong>Sequential cohesion</strong></td>
<td>outputs of one part form inputs to the next</td>
<td><strong>Okay</strong></td>
</tr>
<tr>
<td><strong>Communicational cohesion</strong></td>
<td>operations that use the same input or output data</td>
<td><strong>Moderate</strong></td>
</tr>
<tr>
<td><strong>Procedural cohesion</strong></td>
<td>a set of operations that must be executed in a particular order</td>
<td><strong>Low</strong></td>
</tr>
<tr>
<td><strong>Temporal cohesion</strong></td>
<td>elements must be active around the same time (e.g. at startup)</td>
<td><strong>Low</strong></td>
</tr>
<tr>
<td><strong>Logical cohesion</strong></td>
<td>elements perform logically similar operations (e.g. printing things)</td>
<td><strong>No way!!</strong></td>
</tr>
<tr>
<td><strong>Coincidental cohesion</strong></td>
<td>elements have no conceptual link other than repeated code</td>
<td><strong>No way!!</strong></td>
</tr>
</tbody>
</table>
UML Packages

We need to represent our architectures
- UML elements can be grouped together in packages
- Elements of a package may be:
  - other packages (representing subsystems or modules);
  - classes;
  - models (e.g., use case models, interaction diagrams, statechart diagrams, etc)
- Each element of a UML model is owned by a single package
- Packages need not correspond to elements of the analysis or the design
  - they are a convenient way of grouping other elements together

Criteria for decomposing a system into packages:
- Ownership
  - who is responsible for working on which diagrams
- Application
  - each problem has its own obvious partitions;
- Clusters of classes with strong cohesion
  - e.g., course, course description, instructor, student,...
- Or use an architectural pattern to help find a suitable decomposition

Package notation

2 alternatives for showing package containment:
Package Diagrams

- **Dependencies:**
  - Similar to compilation dependencies
  - Captures a high-level view of coupling between packages:
    - If you change a class in one package, you may have to change something in packages that depend on it

- **A good architecture minimizes dependencies**
  - Fewer dependencies means lower coupling
  - Dependency cycles are especially undesirable

---

...Dependency Cycles

Sub-system A

Sub-system B

Sub-system C

Sub-system D

The server sub-system does not depend on the client sub-system and is not affected by changes to the client’s interface.

Each peer sub-system depends on the other and each is affected by changes in the other’s interface.
Architectural Patterns

E.g. 3 layer architecture:

- Presentation Layer Package
  - Java AWT
  - Application Windows

- Application Logic Layer Package
  - Control Objects
  - Business Objects

- Storage Layer Package
  - JDBC
  - Object to Relational
  - Java SQL