High-level Design

- **Software Architecture**
  - What is it?
  - Examples of common architectures
- Parnas’ KWIK index example of information hiding
- Model view controller in high level layered design

---

What is software architecture?

“The architecture of a system is comprehensive framework that describes its form and structure -- its components and how they fit together” Jerrold Grochow, Pressman, Ch 10

- Describes overall shape & structure of system
  - How components are integrated into cohesive whole
  - Their externally visible properties
  - Their relationships
- **Goal:** choose architecture to reduce risks in SW construction & meet requirements
SW Architectural Styles

- Architecture composed of
  - Set of components
  - Set of connectors between them
    - Communication, co-ordination, co-operation
  - Constraints on integration
  - Semantic models for understanding the overall properties from analysis of component known properties

- Architecture patterns for common organizational structures

Pipe & Filter

- Examples:
  - UNIX pipes, compilers, signal processors

- Pros:
  - Filters oblivious of their neighbors, can be built in parallel
  - System behavior is compositional

- Cons:
  - Hard to handle errors
  - Often need encoding/decoding of input/output

Source: Adapted from Shaw & Garlan 1996, p21-2. See also van Vliet, 1999 Pp266-7 and p279
Event-based Architecture

- **Examples:**
  - GUIs, Breakpoint debuggers

- **Pros:**
  - Anonymous handlers of events
  - Supports re-use and evolution, new agents easy to add

- **Cons:**
  - Components have no control over order of execution

Source: Adapted from Shaw & Garlan 1996, p23-4. See also van Vliet, 1999 Pp264-5 and p278

---

Data-centric Architecture

- **Examples:**
  - Databases, programming environments

- **Pros:**
  - Promotes integrability (ease of changing/adding clients)
  - Reduces need to replicate complex data

- **Cons:**
  - Blackboard can become a bottleneck

Source: Adapted from Shaw & Garlan 1996, p26-7. See also van Vliet, 1999, p280
Process Control Architecture

- **Examples:**
  - Control systems (e.g., airplanes, spacecraft, industrial production lines, power stations)

- **Pros:**
  - Handles real-time, reactive systems
  - Separates control policy from controlled process

- **Cons:**
  - Hard to specify timing constraints or responses to disturbances

Source: Adapted from Shaw & Garlan 1996, p27-31.

---

OO Architecture

- **Examples:**
  - Abstract datatypes and object broker systems (e.g., CORBA)

- **Pros:**
  - Offers data hiding
  - Problems expressed as set of interacting agents

- **Cons:**
  - Objects need to know with whom they need to interact (more later on patterns)

Source: Adapted from Shaw & Garlan 1996, p22-3.
Layered Architecture

- Examples:
  - Operating systems, Communication protocols, E-commerce applications

- Pros:
  - Supports increasing levels of abstraction during design
  - Supports re-use and enhancement
  - Can have standard layer interfaces

Source: Adapted from Shaw & Garlan 1996, p25. See also van Vliet, 1999, p281.

Key Principle: Information Hiding

- Design modules to hide design decisions
  - To accommodate possible change
  - Module as a responsibility assignment, not a subprogram

- Problem
  - KWIC = Key Word In Context
  - Task is to build a contextualized index for the text; Input is a set of lines of text; Output is the set of all circular shifts of all lines, in alphabetical order

- Two Designs
  - Shared data model
  - Data abstraction model

D. Parnas, "On the criteria to be used in decomposing systems into modules", CACM 1971.
**KWIK Index - Shared Data**

- **Control-flow based design**
  - Every module “knows” internal representation of data and addressing mechanism!
  - Good for adding another functionality

- **Possible changes**
  1. Input format
  2. Decision to store index in-core
  3. Packing of chars in words
  4. Use indexed notation or write out shifts
KWIK Index - Information hiding

- Based on hiding design decisions from other modules
  - Easier to change
    - #4 only affects the circular shifter here

Design for Extension & Contraction

- Idea of tailoring SW for specific apps
  - “family of programs w common aspects

- Idea (like a VM):
  - Identify minimal subset of reqs needed to be useful
  - Use information hiding
  - Don’t think of components as processing steps
  - Think about uses (not invokes) relation
    - Pgm A uses pgm B if sometimes the correct functioning of pgm A requires availability of a correct implementation of pgm B

D. Parnas, "Designing software for ease of extension and contraction", IEEE Trans on SE, March 1979
Uses Relation

- Think about uses relation
  - Level 0 programs use no other
  - Level k programs use at least 1 program at level (k-1) but none at higher level
  - Then each level is a testable, usable subset of the system

- A should be allowed to use B if
  - A is simpler thereby
  - B is not more complex than A and doesn’t use A (want acyclic relation)
  - There is a useful subset of the system containing B but not A
  - There’s no useful subset of the system containing A but not B

Layered Architecture

- Logical architecture
  - Large-scale organization of SW classes into packages (i.e., namespaces), subsystems and layers

- Layer - coarse-grained grouping of classes having responsibility for a major aspect of system
  - Strict - layer only calls layer directly below it
  - Relaxed - layer can call any layer below it
  - Responsibilities of objects in a layer are strongly related to each other

- UML package diagrams show logical architecture (LAR Ch 13.1)
  - Coupling between packages shown by UML dependency line, A→→→→ B
Three Common Layers

- **User interface**
- **Application logic and domain objects**
  - E.g., Sale
  - Focus of OOA/D: Domain layer classes may represent domain model conceptual classes
- **Technical services**
  - E.g., interfacing with database, error logging
- **Layer can be partitioned into horizontal slices**
  - E.g., tech services: security & reporting

---

Example

<table>
<thead>
<tr>
<th>Payment</th>
<th>1 Pays-for 1</th>
<th>Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td></td>
<td>date</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Payment</th>
<th>1 pays-for 1</th>
<th>Sale</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount:Money</td>
<td></td>
<td>date: Date</td>
</tr>
<tr>
<td>getBalance(): Money</td>
<td></td>
<td>startDateTime: Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getTotal(): Money</td>
</tr>
</tbody>
</table>

Domain layer, Design model
Domain Layer in UP Model

- **Domain**
  - POS
  - Inventory
  - Tax

- **Tech services**
  - Persistence
  - Security

**High cohesion**
**Separation of concerns**

Model-View Separation Principle

1. Do not connect non-UI objects directly to UI objects
   - Sale object should not refer to a Java Swing JFrame object
2. Do not put application logic in UI object methods
   - UI objects initialize UI elements; receive UI events (e.g., mouse click); delegate requests for application logic to non-UI objects
3. Messages from UI to domain layer will be messages in corresponding sequence diagram
Example

```
Example cashier system
   makeNewSale()  enterItem(id, quant)
       description, total
    endSale()
```

UI

```
Swing
ProcessSale Frame
```

Domain

```
Register
  makenewSale()  enterItem()
```

```
High-level Design, CS431 F06, B G Ryder/A. Borgida/A Rountev
```

Object-Oriented Design

- Objects, with attributes and operations
- Classes define the blueprint for objects
  - Objects are instances of classes
- Messages between objects
  - Object x sends message to object y to activate one of m’s operations
- Operation in design -> method in code
- Message in design -> method call in code
“Flavor” of OO Design/Code

- Many small methods, many calls
- Distributed control: processing is split among many participants
- Distributed vs. centralized control
  - Centralized is easier to understand
    - In OO: “chasing around the objects, trying to find the program”
  - Distributed is more flexible
    - Reduces the impact of change

Example: Total of a Sale w/ Discount

- Centralized
  - Sale asks each SalesLineItem for its quantity and ProductSpecification
  - Sale asks each ProductSpecification for the price
  - Sale computes total:=sum(quantity*price)
  - Sale asks Customer object for discount info
  - Sale calculates discounted price using discount info
Example, cont.

- **Distributed**
  - Sale asks each SalesLineItem to compute its subtotal
  - Sale computes total := sum(subtotals)
  - Sale gives the total to the Customer object and asks it to compute the discounted price

- **More delegation of responsibility, less coupling**

Core Principles of OO Design

- **Identify** the responsibilities that are needed to satisfy the requirements
- **Assign** these responsibilities to objects
  - Add the appropriate operations (i.e., methods)
- **Design the interactions** among objects
  - Add the appropriate messages (i.e., calls)
### UP Artifacts

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Incep</th>
<th>Elab</th>
<th>Const</th>
<th>Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use-Case Model</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suppem. Spec</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domain Model</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Design Model</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Implem. Model</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Requirements analysis:** Use-Case Model + Supplementary Specification

**Domain analysis:** Domain Model

**Design:** Design Model

**Coding:** Implementation Model

### Relationships Among Models

```
Use Case Model

Domain Model

Design Model

Code
```
Design in the Unified Process

- At the end of elaboration
  - Almost all requirements are clarified
  - High-risk design aspects are stabilized

- Construction: iterative design and coding for the remaining requirements

- Interaction diagrams: sequence diagrams, communication diagrams

- Design class diagrams