Announcements

- Discuss your product backlog in person or via email by Wednesday, February 1
- Design document due Monday, February 6

Lecture 09

- Coupling
- Modeling interactions and behavior

Software design concepts

- What is a module?
  - “A lexically contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier”

Abstraction

- High: general terms from problem environment
- Low: stated in direct implementation terms

Information hiding

- Moving the details of the software implementation to as low a level as possible
  - Separates module definition from actual implementation
  - Allows implementation to change without affecting calling routines
Independence

- Module or class independence can be measured using two qualitative criteria: cohesion and coupling

Cohesion

- A measure of the relative functional strength of a module
- Highly cohesive systems have increased readability and reusability
- Complexity is well managed
- In object-oriented programming, classes are cohesive if the methods are similar in many aspects

Cohesion “categories”

- Functional – best
- Communicational/Informational – almost as good
- Procedural
- Temporal
- Logical
- Coincidental – worst

Coupling

- A measure of the relative interdependence among modules
- Changes in one place may require changes somewhere else
- Difficult to see how components work

Coupling categories

- Data – best
- Stamp
- Control
- External
- Common
- Content – worst

Content coupling

- Two modules are content coupled if one directly references the contents of the other
- Encapsulation is broken
- Examples
  - Module A modifies code in module B
  - Module A branches into the local space of module B
  - Module A uses data within the local space of module B
Common coupling
- Two components have and use write access privileges to the same global data
- Written by only one routine and read only by one or more routines is not common coupling
- Singleton pattern provides encapsulated global access to an object

External coupling
- Modules use or pass data and/or control signals to external systems or devices
- OS dependencies, shared libraries, hardware
- Examples
  - System calls
  - Mac tool box commands
  - Direct I/O routines

Control coupling
- Two components are control coupled if one passes an element of control to the other component
- Calling module must explicitly control the logic of the called module
- Example: integer "signal" passed to C switch statement. Module A must know internal structure of module B

Stamp coupling
- Components pass data structures (classes) as parameters
- Not all fields required by called module
- Examples
  - Linked lists

Data coupling
- Two components are data coupled if all parameters (data items) are used by the called routine
- No data items best; but, usually not possible
- Trade-off between data coupling and stamp coupling
- Increasing one often decreases the other

Coupling attributes

<table>
<thead>
<tr>
<th>Coupling</th>
<th>Independence</th>
<th>Error prone</th>
<th>Reusable</th>
<th>Extensible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Stamp</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Control</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>External</td>
<td>Low to Medium</td>
<td>High</td>
<td>Low to Medium</td>
<td>Low</td>
</tr>
<tr>
<td>Common</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Content</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>
Dynamic (behavioral) diagrams

- Model the dynamic aspects of a software system
- Sequence diagrams
- State machine diagrams
- Activity diagrams

Interactions and messages

- Sometimes built from a use case (user story) and a class diagram
- Show how a set of objects accomplish the required interactions with an actor
- Steps taken together is an interaction
- Can show several types of communication
  - Method calls, network messages, etc
  - All referred to as messages

Diagram elements

- Class instances
  - Boxes with class and object identifier underlined
- Actors
  - Stick-person symbol (use case diagrams)
- Messages
  - Arrows from actor to object or object to object

Creating

- Develop a class diagram and use case model before creating an interaction diagram

Sequence diagrams

- Shows the sequence of messages exchanged by a set of objects performing a specific task
  - Objects are arranged horizontally
  - Actor is often on the left
  - Vertical dimension represents time
  - Vertical line, lifeline, attached to each object or actor
  - Messages are arrows between sender and receiver
  - Can have argument list and return value
**State diagrams**
- Describe the behavior of a system, part of a system, or an individual object
- Directed graph
  - Nodes = states
  - Arcs = transitions
- System or object is always in some state
  - Will behave in a specific way in response to an event or input
  - Events cause the system to change state
  - Each state behaves a certain way

**States**
- System is always in a single state
- Event causes state to transition
- States are rounded rectangles
  - Contain the name
- Special states
  - Black circle = start state
  - Circle with a ring around it = end state
  - Can be multiple end states

**Transitions**
- Change of state in response to an event
- Label on each transition is the event
**Time-outs and conditions**

(a) [Diagram of traffic light with time-outs]

(b) [Diagram of traffic light with time-outs]

**Actions**

- Closed:
  - enter: stop motor
  - open: run motor forwards

- Opening:
  - enter: run motor in reverse
  - opening: complete

- Closing:
  - enter: run motor

**Activity diagrams**

- Similar to a state diagram except most transitions caused by internal events (e.g., completion of computation)
- Help understand object or component work flow
- Visualize interrelation and interaction between different use cases
- Most often associated with several classes
- Can represent concurrency

**Concurrency**

- Fork - one incoming transition, multiple outgoing transitions
- Join - multiple incoming transitions, one outgoing transition
- All incoming transitions must occur before proceeding
- Rendezvous - multiple incoming and multiple outgoing transitions
- All incoming transitions must occur first
Swimlanes
- Partition activities among classes

Questions?