CS189A - Capstone

Chandra Krintz
Department of Computer Science
UC Santa Barbara
http://www.cs.ucsb.edu/~ckrintz/
Next Up: Design & Prototyping

- Detailed design specification (Design Doc)
  - **Augment your PRD v1 to produce v2**
  - Break down stories into tasks and components associated with design
    - Prioritize stories
    - Assign timings to stories
    - Specify acceptance/test that can be used to verify a story is finished
  - Prototype tasks (primary implementation before demo)
    - Prioritize tasks
    - Assign timings to tasks
    - Specify what test(s) are to be used as evidence of task completion/acceptance
    - Each member/developer chooses task, implements, and tests task
    - Another member does code review/test and accepts the pull request
      - Test is the one specified above (Acceptance)
    - When a Story is complete, some member performs story test/acceptance

• Authors, Team, Project Title
• Intro – including problem, innovation, science, core technical advance
• Glossary of Terms
• System architecture overview
  – SW architecture with significant detail; components; APIs; dependencies
• Requirements (functional and non-functional)
  – Update original use cases or user stories as needed - reprioritize
  – An additional 20 use cases or user stories -- prioritized
• Prototyping code and test cases (Github URL)
• System models (design)
  – Contexts, interactions, structural, behavioral (UML)
  – Use cases, sequencing, event response, system state, classes/objects
• Appendices -- Technologies employed
Review

• Use Cases
• User Stories
Use Cases

• A **use case** describes the possible **sequences of interactions** among the system and one or more actors in response to some initial stimulus by one of the actors
  – Each way of using the system is called a use case
  – A use case is not a single scenario but rather a description of a set of scenarios
  – For example: *Creating an account*
  – Individual use cases are shown as named ovals in use case diagrams

  *Creating an account*

• A use case involves a sequence of interactions between the initiator and the system, possibly involving other actors.
• In a use case, the system is considered a black-box.

*We are only interested in describing externally visible behavior*
*Have preconditions and postconditions*
Use Cases

• To define a use case, group all transactions that are similar in nature

• A typical use case might include a main case, with alternatives taken in various combinations and including all possible exceptions that can arise in handling them

• Description of a use case should include events exchanged between objects and the operations performed by the system that are visible to actors

• Have preconditions and postconditions
  – Precondition states all assumptions about state/environment of system that impacts the actor(s) in this use case
  – Postcondition is an acceptance test (how to know when implementation is complete) and describes externally visible state/environmental changes
**Use case:** Update Benefits

**Actors:** Employee, Employee Account Database, Healthcare Plan System, Insurance Plan System

**Precondition:** Employee has logged on to the system and selected “update benefits” option

**Flow of Events:**

**Basic Path:**
1. System retrieves employee account from Employee Account Database
2. System asks employee to select medical plan type; **uses** Update Medical Plan
3. System asks employee to select dental plan type; **uses** Update Dental Plan
...

**Alternative Paths:**
If health plan is not available in the Employee’s area the employee is informed and asked to select another plan  *(exceptional cases that must be handled)*

Employee selects cancel, logs out, or leaves page at any point prior to confirming the update

**Postcondition:** Employee account plan type has been updated

*Note that code tests can be written for pre/post conditions*
Writing Good User Stories

• It's typically difficult to get started writing good user stories
  – Here are 4 steps to make it easier

1. As a [role], I can [feature] so that [reason]
2. Use index cards and a sharpie
3. Make it testable with acceptance stories
4. Connect the dots

From: http://codesqueeze.com/the-easy-way-to-writing-good-user-stories/
As a [role], I can [feature] so that [reason]

- Role – a person; feature – something your project does; reason – a solution to a problem the person has
  - This is a pattern that is commonly used for stories

  As a account owner, I can check my balance online so that I can keep a daily balance 24 hours a day.

- Variations
  - As a [role], I want [feature] because [reason]
  - As a [role], I can [feature]
  - As a [role], I can [feature] so that [reason]
Make it testable with acceptance stories

• If they are short and sweet and without detail, how do we know when they are “done”?

• Include an acceptance test:
  
  **Scenario 1: Title**
  
  Given [context]
  
  And [some more context]…
  
  When [event]
  
  Then [outcome]
  
  And [another outcome]…

  **Example**

  Scenario 1: Account balance is negative
  
  Given the account’s balance is below 0
  
  And there is not a scheduled direct deposit that day
  
  When the account owner attempts to withdraw money
  
  Then the bank will deny it
  
  And send the account owner a nasty letter.

• All tests should fit on back of story card (in sharpie)
  - If they don’t break up the story into two
  - You should be able to **code** them in a few lines of code
Software Design

• Two primary phases:
  – Architectural Design
    • Divide the system into a set of modules
    • Determine the interfaces of the modules
    • Figure out the interactions among different modules
  – Detailed Design
    • Detailed design for individual modules
    • Write the pre and post-conditions for the operations in each module
      – The conditions that must be true before (pre) and after (post) each
    • Draw pictures
    • Use code/documentation to explain individual modules key functionality
      – Automatic documentation generation (e.g. sphynx generator)
      – Alternatively, you can write unit tests for each and turn in your code as part of the design doc (git repo)
Modularity: Reducing Design Complexity

• Modularity principle suggests dividing a complex system into simpler pieces, called modules
  – Possible: Module = function or Module = functions or Module = functions+data

• When we have a set of modules, we can use separation of concerns and work on each module separately
  – to improve
    • maintainability
    • reusability
    • productivity

• Modularity can also help us to create an abstraction of a module’s environment using interfaces of other modules
Two Modularization Strategies

• Both attempt to generate modules with **low coupling** and **high cohesion**
  – **Coupling** is a measure of a module’s independence
    • The degree of dependency among modules (lower is better)
    • Minimize and localize change to one module v/s those that depend on it
  – **Cohesion** is a measure of the degree to which all elements of a module are directed toward a single task (how self contained are they?)
    • The internal glue that holds a module together (higher is better)

• Modularization techniques
  – Functional decomposition
  – Parnas’ modularization

  “On the Criteria to be Used in Decomposing Systems into Modules”, Parnas, 1972
• Functional decomposition
  – Divide and conquer approach – modules are steps in the computation
  – Use stepwise refinement
    1. Clearly state the intended function
    2. Divide the function to a set of sub-functions and re-express the intended function as an equivalent structure of properly connected sub-functions, each solving part of the problem
    3. Divide each sub-function far enough until the complexity of each sub-function is manageable
Functional Decomposition

• One way of achieving functional decomposition: Make each step in the computation a separate module
  – Draw a flowchart showing the steps of the computation and convert steps of the computation to modules
  – **Shortcoming**: Does not specify the granularity of each step

• Another way of achieving functional decomposition is to look at the **data flow** in the system
  – Represent the system as a set of processes that modify data. Each process takes some data as input and produces some data as output.
  – Each process becomes a module

• **Shortcoming**: Both of these approaches produce a network of modules, not a hierarchy
What about Data Structures?

- **Fred Brooks**: “Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won’t usually need your code; it’ll be obvious.”
  - Author of The Mythical Man Month and No Silver Bullet (IBMer, Turing Award Winner)

- **Eric Stevens Raymond**: “Smart data structures and dumb code works a lot better than the other way around.”
  - Open source evangelist and author of The Cathedral and the Bazaar and The New Hacker’s Dictionary

- Functional decomposition focuses on operations performed on data
- According to Brooks and Raymond data structures should come first
- Parnas’ modularization approach (from 1972!) focuses on data
Parnas’ Modularization

- Define your set of data structures
- Foreach data_structure
  - Define the set of possible operations on it (as functions)
    - Encapsulate code and data
    - Make public the set of functions that other modules or users employ to interact with the data structure
      - Make everything else (code and data) private
- Make each data structure reusable and extensible (inheritance)
  - And customizable (polymorphism)
The Unified Modeling Language (UML)

• A tool for all phases of software development
  – Requirements specification, architectural design, detailed design & impl

• Many books on UML, some goode ones are:
  – “UML Distilled,” Martin Fowler
  – “Using UML,” Perdita Stevens
  – “UML Explained,” Kendall Scott

• The Object Management Group (OMG, a computer industry consortium) defines the UML standard
  – The current UML language specification is available at:
    http://www.uml.org/

• Tools: http://www.visual-paradigm.com/solution/freeumlDesignTool/
  – http://yuml.me (online tool)
The Unified Modeling Language (UML)

• A tool for all phases of software development
  – Requirements specification, architectural design, detailed design & impl

• **Class diagrams**
  – Visual representation of the static structure, interrelationships, and composition of a particular system
  – Most used UML diagram type

  – Help simplify how objects in a system interact
  – **Facilitate translating a designed system into code prototypes**
Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>operation()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circle</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>itsRadius:double</td>
</tr>
<tr>
<td></td>
<td>itsCenter:Point</td>
</tr>
<tr>
<td></td>
<td>Area():double</td>
</tr>
<tr>
<td></td>
<td>Circumference():double</td>
</tr>
<tr>
<td></td>
<td>SetCenter(Point)</td>
</tr>
<tr>
<td></td>
<td>SetRadius(double)</td>
</tr>
</tbody>
</table>
## Access Modifiers

<table>
<thead>
<tr>
<th>Circle</th>
<th>Public</th>
<th>Protected</th>
<th>Private</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ itsRadius: Double</td>
<td>+</td>
<td>#</td>
<td>-</td>
<td>~</td>
</tr>
<tr>
<td>+ itsCenter: Point = (0, 0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Area(): Double</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Circumference(): Double</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ SetCenter(Point)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ SetRadius(double)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Class Interrelationships: Logical Connections

Association

Directed Association

Reflexive Association

Multiplicity

Aggregation

Composition

Inheritance

Realization

From: http://creately.com/blog/diagrams/class-diagram-relationships/
UML Diagram Relationships

UC Santa Barbara

From: http://creately.com/blog/diagrams/class-diagram-relationships/
Annotations

• For any relationship (edge between classifiers), we can annotate:
  – The name of the relationship (may be directional – indicated with a solid arrowhead in the direction the relationship holds)
  – The role of target instance in the source
  – Cardinality constraints (1:N, N:M, etc.) at either end
  – Possible ordering at either end

<table>
<thead>
<tr>
<th></th>
<th>Exactly one</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Many (any number)</td>
</tr>
<tr>
<td>0..1</td>
<td>Optional (zero or one)</td>
</tr>
<tr>
<td>m..n</td>
<td>Specified range</td>
</tr>
<tr>
<td>{ordered}*</td>
<td>Ordered</td>
</tr>
</tbody>
</table>
Composition

- Parent class contains 1+ instances of child class in some way
  - The containment is **complete** - not seen elsewhere
  - A type of **has-a** relationship, e.g. Child belongs-to Parent
- Represented with a filled-in diamond
- Aggregation + **lifetime** implications: When the parent object is deleted, the child object is also deleted.

Multiplicity can/should be used for any relationship
Aggregation

- Weaker version of composition
- Does not speak to object lifetimes
- Represented with an open diamond
- Not commonly used
  - No clear semantics in UML (underdefined)
  - Association is used instead...

Diamond-end class is built as a collection of instances of non-diamond-end class

Relationship with 1 to many Books instances (multiplicity)

Multiplicity can/should be used for any relationship
Association

- A relationship between two classifiers (classes) – very abstract
  - With a description of what the relation is (any logical relation)
  - Not a composition or a whole/part relationship (more general)
- Represented with a line without a diamond

Edges can be directional or bidirectional (no arrow).
Association

- A relationship between two classifiers (classes)
  - With a description of what the relation is
  - Not a composition or a whole/part relationship (more general)
- Represented with a line without a diamond

Edge can be labeled with a description of the association (name, not shown here) and with the **role** of the instance. Example: bidirectional association w/ multiplicity at each end and roles

A flight is associated with 0 or 1 planes (the assignedPlane). A plane also knows about its associated flights (there are 0 or more of them and they are referred to via the role assignedFlights)
Association vs Aggregation

• Association is a relationship where all objects have their own lifecycle and there is no owner. Example: Teacher and Student objects. Multiple students can associate with a single teacher and single student can associate with multiple teachers, but there is no ownership between the objects and both have their own lifecycle. Both can be created and deleted independently.

• Aggregation is a special form of Association where all objects have their own lifecycle but there is ownership and the child object cannot belong to another parent object. Example: Department and Teacher objects. A single teacher cannot belong to multiple departments, but if we delete the department, the teacher object will not be destroyed. Aggregation implements a “has-a” relationship.
Abstract Class

- Defines one or more abstract methods
- Intended to serve as a base class
  - Class name is *italicized*
- Either it contains data or it contains at least one non-abstract method (ow it is an Interface)
- Inheritance denoted with an open arrowhead
Interfaces

- A special kind of abstract class
- Contain no data, and all methods are abstract
- Connected via dotted arrow: realization/implementation relation

Diamond = aggregation; no diamond = simple assoc
If diamond was filled in then composition (lifetime dependency)
Arrow means directional association
Dotted line means weak dependency (used in argument) or realization (implements)
Realization or Dependency

- Can have an open arrow (aggregation) or filled arrow (composition=lifetimes depend upon each other)
- Used to show that one class is dependent upon another in some way
  - Dependency: client (tail of arrow) depends on supplier (arrow head)
    - Labeled with <<uses>> for data-access dependencies
    - Labeled with <<creates>> for classes that create instances of another (however create may instead be a method in the tail class)
  - Realization: supplier is interface; client is implementation of some/part of that interface
Interfaces

- Specify interfaces provided/used in the class
- Or as “lollipops”
• See lecture notes for a bunch of links and examples
UML Diagram Relationships

From: http://creately.com/blog/diagrams/class-diagram-relationships/
Nontrivial Example
Small solid triangle placed in front of association name to show order of the association.
Sequence Diagrams: Basic Idea

- Illustrate interactions between objects over time
- Show behavior as opposed to static design
Classes

- Dashed lines are called Object Lifetimes
Interactions

• In pure OOP, all interactions are through method calls
• Showing how one class communicates with another
• Synchronous messages represented with a solid horizontal arrow with an enclosed head
• Message name and optionally parameters specified
Synchronous Messages

doSomething(5)
Return Values

- Represented with a dashed arrow with a stick head
- Returned to caller, can be annotated with kind/type of info returned
Asynchronous Messages

- Represented with solid arrows with stick heads
- Call may never return
- Semantics may be of parallel execution, but hard to represent
Long Message

- Something that takes awhile to process
- Usually used for I/O bound activities
  - Disk access, network access
- Used to indicate a potentially noticeable delay
- Represented with a slanted arrow with an enclosed head

```java
doSomethingLong( 5 )
```
Object Creation

- For objects that did not exist at the beginning
- Shown further down the diagram

```
new(i)
getSomething(x)
something
```
Object Destruction

- Ending the lifetime of an object
- Represented with an “X”
Common Stereotypes

- Non-objects that are intrinsic to the interactions
- Can include users, databases, etc.
Example 1

- ShoppingCart
- Order

* addItem(i) -> new(i) -> o1: OrderLine

- remove(o1)

- the ShoppingCart and Order objects already exist when the interaction starts
- an OrderLine object is created
- the lifeline indicates its lifetime
- the OrderLine object is destroyed
Example 2
Example 3

1. User opens the iBooklet
   1.1: Displays Log In Page
2. Selects Sign Up
   2.1: Displays Sign Up Page
   3: Enters Credentials
   5.1: Sends Credentials Info
     5.1.1: Checks If Valid
     5.1.2: Creates New User
     5.1.3: Sends Creation Confirmation
   3.1.3: Sends Creation Confirmation
   4: Enters Log In Credentials
   4.1: Sends Log In Info
   4.1.1: Checks If Valid
   4.1.2: Sends Login Confirmation
4.1.2.1: Redirects To Book Selection Screen

• Authors, Team, Project Title
• Intro – including problem, innovation, science, core technical advance
• Glossary of Terms
• System architecture overview
  – SW architecture with significant detail; components; APIs; dependencies
• Requirements (functional and non-functional)
  – Update original use cases or user stories as needed - reprioritize
  – An additional 20 use cases or user stories -- prioritized
• Prototyping code and test cases (Github URL)
• System models (design)
  – Contexts, interactions, structural, behavioral (UML)
  – Use cases, sequencing, event response, system state, classes/objects
• Appendices -- Technologies employed
Summary of PRD Draft 2 Milestone

• Revise spec to add detail to the functional specification to match your design
• Add user stories and break up the stories you have into finer grained stories
  – Provide UML, sequence diagrams, dataflow diagrams
  – Goal: a CS senior should be able to take your doc and implement the project
• For each fine-grained story, provide a description and acceptance test
  – Provide time estimates (1 person-hours) for each story implementation
    • Ensure you can finish the implementation in the time you have (this/next quarter)
  – Prioritize tasks to have a complete prototype by the end of this quarter
    • Focus on the externally facing interfaces, mock out what you cannot get to
  – Write unit tests to implement tasks for mandatory tasks
    • Document these tasks (autogen the documentation/usage)
• Prototype designed mandatory tasks
  – Github must have unit tests, documentation (for anything without unit tests), and prototyping implementations
• If you have a user interface
  – Provide mockups that are tied to the functionality described in 1+ components