CS48 – CS Project

Chandra Krintz
Department of Computer Science
UC Santa Barbara
http://www.cs.ucsb.edu/~ckrintz/
CS48 Sprints

• Sprints: **M-F daily standup/scrum w/ burndown**
  • Sprint 1: April 15 – 26, planning in class/section Apr 10-12
    – Plan sprint 2 Apr 26 w/ tutor: w/ at least 3 new stories/use cases
    – Retrospective; start sprint 2 **in class** Apr 29
  • Sprint 2: April 29 – May 10, plan sprint 3 May 10 w/ tutor
    – Retrospective; start sprint 3 in class May 13
  • Sprint 3: May 13 – May 24, retro/demo prep w/ tutor May 24
Sprint Planning (in Friday’s Discussion Section)
This slide is linked on the schedule under Friday’s date

- Product backlog should have 8+ stories/use cases prioritized (some completed in prior sprints)
  - **Story:** Name; As a [role], I can [feature] so that [reason]; **Acceptance test** = 1+: Name, Given [context], when [event], then [outcome]
  - **Use case:** Name, Actors, Precondition(s), Event flows (normal/exceptional); **Acceptance test** = Postcondition(s)
    - Contact Chandra/Tas for help with these if needed!

- Break up top two stories unfinished into tasks of 1/2 day or 1 day long
  - Repeat until you have at least 10 days worth of work for each team member

- Go around the team, having each pick a task to perform
  - Until each team member has 10 days worth of work
  - Give each task 2 tests: normal case, exceptional case

- Document story, acceptance test, task breakdown, task tests, and task assignment (team member)
  - Draft project or Trello cards
  - If Trello: then put Name and link to trello card in the Draft project
  - Share with ckrintz@ucsb.edu, jcai00@ucsb.edu, nazmus@ucsb.edu, + mentor

- **Note** that you are required to have a component in C++/Java (analysis, event, service)
- In class review of travis-ci, github, and AWS lambda and cloud services: next week
Draft Project – Due May 8th

• Authors, Team, Project Title
• Intro – **Vision Statement** (revised): 2-3 pages
  – Define project specifics, team goals/objectives, background, and assumptions
  – Links to github, travis-ci, slack
• System architecture overview (drawings) – **Project Design**
  – **High level diagram** (1 page)
  – **Interfaces/interactions between components** (UML, sequence diag.)
• Requirements (functional and non-functional)
  – **User stories or use cases** → 5+ for first turn-in, 10+ for final turn-in
    -- each prioritized with **acceptance tests**
  • Links to github issues, commits, tests/demos for all subtasks
  • You can use Trello or Pivotal for free (online stories/use cases) – if you use this, then make sure TAs/instructor are invited and give links in draft project
• **Appendix**
  – List of technologies employed
Use Case (Example)

Use case: Update Benefits


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 (an end-early path)

Postcondition: Employee account plan type has been updated in the Employee Account Database or nothing has changed (end-early paths)

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

• As a [role], I can [feature] so that [reason]
  – Use index cards and a sharpie

• Make it testable with acceptance criteria or demo plan
  – Acc. Test: Title: Given [context], when [event], then [outcome]
  – Should be easily coded (commit tests as part of pull request)

• Connect the dots
  – Lay the stories out, determine which ones are dependent on others, prioritize them in order to provide a working system/product each sprint

From: http://codesqueeze.com/the-easy-way-to-writing-good-user-stories/
Self Augment your Draft Project (shared google doc) as you go
  – Break down stories into tasks and components associated with design
    • Prioritize stories
    • Assign timings to stories/use cases
    • Specify acceptance test / demo to verify a story is finished
      – Each task must have an acceptance test as well – typically 2+ code tests
        (normal case, exceptional case)

• Next step: Design
  – Consider "real" use cases
    • Sharpen focus to actual technology, specific user interfaces, particular other systems, …
  – Package coherent subsystems together
    • And organize the packages into overall system architecture
  – Model the interactions between objects
    • Including interactions between packages
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 – focus = operations performed on data
  – 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 (IBM, 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 and design
  – **Use it for designing your modules and interfaces**

• Many books on UML, some good 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:

• **Tools:**
  – [http://yuml.me](http://yuml.me) (online tool)
The Unified Modeling Language (UML)

• **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>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

**Association**
- Airplane to Passengers
- Passengers to Airplane

**Directed Association**
- A container/contained directional flow
- Airline Staff to 0..* Passenger

**Reflexive Association**
- Multiple functions + relationship between 2 instances of same class
- Relationship with 0-to-many Passenger instances

**Multiplicity**
- Built as a collection
- Library to Books

**Aggregation**
- A container/contained directional flow
- Aggregation w/ lifecycle dependency
- Parent to Child

**Composition**
- A child/parent directional flow
- Parent to Child

**Inheritance**
- A child/parent directional flow
- Parent to Child

**Realization**
- Child defines the functionality of the Parent

From: [http://creately.com/blog/diagrams/class-diagram-relationships/](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>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Exactly one</td>
</tr>
<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

![Diagram](image)

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)
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

Abstract Class
Some methods are abstract and some are implemented
Interfaces

- Defines abstract methods
- Inheritance denoted with an open arrowhead

Interface (all methods are abstract)

<<interface>>

Abstract Class
Some methods are abstract and some are implemented
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)
• Specify interfaces provided/used in the class

Stereotype (domain specific classifier)

```
<component>
  Order
  «provided interfaces»
    OrderEntry
    AccountPayable
  «required interfaces»
    Person
```

Written out

Or graphical
Interfaces

- Specify interfaces provided/used in the class
- Or as “lollipops”

Stereotype (domain specific classifier)

Written out
Or graphical
Sequence Diagrams: Basic Idea

- Illustrate interactions between objects over time
- Show behavior as opposed to static design
• Dashed lines are called Object Lifetimes
• Messages between objects are method/function calls
  • Can be asynchronous or synchronous
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

:Foo

:Bar

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

```plaintext
doSomething( 5 )
getSomething( x )
something
```

- 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

```
:Foo

doSomethingLong( 5 )

:Bar
```

```

something
```
Object Creation

- For objects that did not exist at the beginning
- Shown further down the diagram
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.
• See examples of UML diagrams and sequence diagrams linked to today’s date in the CS48 Schedule
Your Project Design: Draft Project

• Architecture (hardware/software)
  – Create an abstract/overview picture
  – The detailed design will provide specifics of 1+ components of it

• Detailed design
  – UML diagrams of 4+ primary data structures that comprise the system architecture connected via their associations (if any) – story/use case
    • Ensure that each "class" is balanced in terms of cohesion & coupling
    • Annotate with pre/post conditions when appropriate
  – Sequence diagrams – **Required** in your draft project
    • 1 synchronous and asynchronous for key interactions between classes
    • 1 User interactions with the system
      – Can be a human user or a machine user (API) interaction
        » Event response, updated application state
      – If you have a user interface: **Provide mockups for primary UIs**
Draft Project – Due May 8th

- Authors, Team, Project Title
- Intro – Vision Statement (revised): 2-3 pages
  - Define project specifics, team goals/objectives, background, and assumptions
  - Links to github, travis-ci, slack
- System architecture overview (drawings) – Project Design
  - High level diagram (1 page): Architecture Design Interfaces/interactions between components (UML, sequence diag.)
    - Detailed Design
  - Requirements (functional and non-functional)
    - User stories or use cases → 5+ for first turn-in, 10+ for final turn-in
      -- each prioritized with acceptance tests
    - Links to github issues, commits, tests/demos for all subtasks
    - Use Trello or Pivotal and give links in draft project
- Appendix
  - List of technologies employed
Today

• Scrum/Standup
• Start working on your architecture diagram
  – Add detail to it via UML classes
• Continue working on Sprint 1 tasks