CS189A
Software Engineering: Concepts and Practices

Jianwen Su
Department of Computer Science
UC Santa Barbara

https://capstone.cs.ucsb.edu/cs189a/cs189a_sched.html
Today’s Lecture

- Progress checking and upcoming deadlines

- Updates from teams:
  - 1-slide team report
    - 2nd sprint goal & plan
    - PRDv1 progress update

- Design techniques and tools
Overall Course Plan @Week 5

- Four 2-week sprints:
  - Oct 15-29 (PRD v1 – tools, technologies, design, terminology);
  - Oct 29-Nov 12 (use cases/user studies, prototyping, PRD v1);
  - Nov 12-26 (design, prototyping, testing, PRD v2);
  - Nov 26-Dec 10 (prototype demo/pres prep, prototyping and testing)

- Specify what the product will do
  - Vision statement
  - Product Requirements Document (PRD) (due Nov 6 and Dec 5)
  - Design tools, brainstorming, coding (tests and implementation)

- Build and test an initial prototype
  - Typically teams iterate on these activities until they converge to a working prototype!
This Week’s Plan

Team activities
- Scrum: Sprint 2 & PRD v1
- Each member: add 1 use case or user story (feature) to PRD v1; add to Product backlog & break down into <1/2 day tasks w/ timings

Section: Scrum, TA meetings:
- Sprint 2 planning (share and show TA Sprint 2 trello board and burndown)
  Sprint 2 continues

Upcoming deadlines:
- Nov 6: Product Requirement Document (release 1)
- Nov 12: Sprint 2 ends (Sprint 3 starts)
  - Product Requirements Document v2
Team Updates

- Each lead overviews their project
  - Spring 2 plan
  - PRDv1 progress

<table>
<thead>
<tr>
<th>Company</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teladoc Health (John)</td>
<td>A²LIST</td>
</tr>
<tr>
<td>Alcon</td>
<td>ALPRO</td>
</tr>
<tr>
<td>Novacoast</td>
<td>BINARY BROS</td>
</tr>
<tr>
<td>Well Health</td>
<td>LOG</td>
</tr>
<tr>
<td>Appfolio</td>
<td>O(MG)</td>
</tr>
<tr>
<td>PowWow Energy</td>
<td>POWWOW++</td>
</tr>
<tr>
<td>LogMeIn</td>
<td>RUNTIME TERROR</td>
</tr>
<tr>
<td>Invoca</td>
<td>STORKEAI</td>
</tr>
<tr>
<td>QAD</td>
<td>TRANSFORM</td>
</tr>
<tr>
<td>Teladoc Health (Ole)</td>
<td>#STUB</td>
</tr>
</tbody>
</table>
Design Techniques and Tools

- Last week:
  Requirements engineering → PRDv1

- Design techniques and tools → PRDv2
The official statement of what is required of the system developers
Includes a specification of both user and system requirements
Defines **WHAT** the system should do, not **HOW** it should do it
  - Design comes later

Agile and extreme SWE processes express requirements as
  - **Use cases** – how a system will act
  - Or as scenarios called **user stories** (describe result/benefit of it)

  - **Both** document how the system responds from an **external** perspective (when viewed from the **outside**) – like a black box…

  **So - we are only interested in describing externally visible behavior**
**Use Case (Example)**

**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 (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/
Next Up: Design & Prototyping

- Detailed design specification (Design Doc)
  - Augment your PRD v1 to produce v2
    - Update/turnin vision statement again if changed, with PRD v2

- Authors, Team, Project Title
- Intro – including problem, innovation, science, core technical advance
- Glossary of Terms
- System architecture overview
  - SW architecture with detailed design; components; APIs; dependencies; UI mockups
- 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 for commit(s))
- System models (design)
  - Contexts, interactions, structural, behavioral (UML)
  - Use cases, sequencing, event response, system state, classes/objects
- Appendices -- Technologies employed
  
Due ~1mo (see schedule for date) via email as PDF
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)
Your Project Design: PRDv2

- **Architecture (hardware/software)**
  - Evolve your overview picture from PRDv1 to provide significantly more detail and any updates or changes

- **Detailed design**
  - UML diagrams of 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
    - Synchronous and asynchronous for key interactions between classes
      - At least 3 different interactions
    - User interactions with the system
      - At least 3 different interactions
      - 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
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
    o maintainability
    o reusability
    o 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 and manage dependencies: **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

- 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
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

- For each 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 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:
    http://www.uml.org/

- Tools: http://www.visual-paradigm.com/solution/freeumldesigntool/
  - http://yuml.me (online tool)
UML Diagram Types

- Use case diagrams: interactions between a system and its external entities (actors) in terms of use cases
- **Class diagrams**: classes used in a system
- State machine diagrams: ways in which an object changes state; different states affect behaviors
- Activity diagrams: workflow or actions (or sequence of events) during program execution
- **Communication (collaboration) diagrams**: interactions among objects in a system, with an emphasis on what interactions occur
- **Sequence diagrams**: interactions among the objects in a system, but emphasize when interactions occur
UML Class Diagram

- 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>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>itsRadius:double</td>
</tr>
<tr>
<td>operation()</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/
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>1</td>
<td></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>
Class Diagram for ATM System

- Keypad
- DepositSlot
- Screen
- CashDispenser
- ATM
- BankDatabase
- Account
- Withdraw

relations:
- ATM executes Withdraw
- BankDatabase contains Account
- Keypad
- DepositSlot
- Screen

properties:
- 0..1: executes
- 0..1: contains
- 1: authenticates user against
- 0..1: accesses/modifies
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 (otherwise it is an Interface)
- Inheritance denoted with an open arrowhead
Interfaces

- Defines abstract methods
- Inheritance denoted with an open arrowhead

Interface (all methods are abstract)

```
<<interface>>

DrawingContext

SetPoint(int,int,bool)
ClearScreen()
GetVerticalSize():int
GetHorizontalSize():int
```

Abstract Class
Some methods are abstract and some are implemented

```
Shape
  Draw()
  Erase()

Circle

Square
```
The Strategy Pattern

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable
- Strategy lets the algorithm to vary independently from clients that use it

General form:

```
Context
  - Strategy myStrategy

Strategy
  algorithm()

ConcreteStrategy1
  + algorithm()

ConcreteStrategy2
  + algorithm()
```
UML Diagrams

- UML provides several types of diagrams to model collaboration behavior of a system
- **Communication diagram:** which objects participate in collaborations
- **Sequence diagram:** when messages are sent between objects over time
UML Communication Diagrams

- **Objects**: rectangles containing names in the form `object Name : ClassName`
  - Only one object of each type: Omit object name, i.e., `: ClassName`

- **Communication**: solid lines message name appears next to the line, usually operation name (i.e., method name in Java) of the receiving object (direction indicated)

- **Message**: solid filled arrow
  - Synchronous call in the UML, method call in Java: Control passed from the sender to the receiver

- **Asynchronous messages**:
Examples

- Single message:

- Also has sequence and nested:

```plaintext
1: getAvailableBalance ( accountNumber )
2: getTotalBalance( accountNumber )
```

```plaintext
1.1: getAvailableBalance ()
2.1: getTotalBalance()
```
Sequence Diagrams

- Model the timing of collaborations
- Dotted line extending down: object’s lifeline, with progression of time
- Solid arrow: a message between two objects
  - Activation on the receiving object’s lifeline
- Thin vertical rectangle: an object is executing
- Return message: the activated object returning control to the original object
Example Sequence Diagram

: Withdraw

: Screen
- displayMessage(message)
- getInput()

: Keypad
- getAvailableBalance(accountNumber)

: Account
- isSufficientCashAvailable(amount)
- debit(accountNumber, amount)

: BankDatabase
- getAvailableBalance()

: CashDispenser
- debit(amount)
- dispenseCash(amount)
- displayMessage(message)
More on Sequence Diagrams

: Class1

new()

activationSynchronousMessage()

activationAsynchronousMessage()

confirmation()

inquiry()

delete()

: Class2
PRDv2: Your Living Requirements Document: A Shared Google Doc

- Authors, Team, Project Title
- Intro: problem, innovation, science, core technical advance (3+ pages)
  - Define project specifics, team goals/objectives, background, and assumptions
- System architecture overview
  - High level diagram (1 page)
  - User interaction and design (1+ pages) – ie detailed design
- Requirements (functional and non-functional)
  - User stories or use cases (links) → 20+ for PRDv2 prioritized w/acc. tests
  - Prototyping code, tests, metrics (10+ user stories): github commits/issues
- System models (1+ pages)
  - Contexts, interactions, structural, behavioral (UML)
  - Use cases, sequencing, event response, system state, classes/objects
- Appendices - Technologies employed
Your Project Design: PRDv2

- **Architecture (hardware/software)**
  - Evolve your overview picture from PRDv1 to provide significantly more detail and any updates or changes

- **Detailed design**
  - UML diagrams of **primary data structures** that comprise the system architecture connected via their associations (if any)
    - Ensure that each "class" is balanced in terms of cohesion & coupling
    - Annotate with pre/post conditions when appropriate
  - **Sequence diagrams**
    - Synchronous and asynchronous for key interactions between classes
      - At least 3 different interactions
    - User interactions with the system
      - At least 3 different interactions
      - 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**
PRDv2 User Stories / Use Cases

- 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)
  - Add trello/pivotal task links (titles must match) to PRDv2 for each story
- Prototype designed mandatory tasks; add github commit ID/link to PRDv2
  - Github must have unit tests, documentation (for anything without unit tests), and prototyping implementations for each story in Sprint
- If you have a user interface
  - Provide mockups that are tied to the functionality described in 1+ components