UML Diagrams
(Slides adapted from Michael Mateas)
Upcoming deadlines

- **Monday (Jan. 13):** Sprint 1 Plan due
  - Due by 11:59pm
  - See website for information and template
- **Monday (Jan. 13):** Sprint 1 begins
- **Friday (Jan. 17):** team status reporting
  - Due by midnight
  - Report on team activities this week
  - Be sure to use team status reporting template
    - courses.soe.ucsc.edu/courses/cmps171/Winter12/01/pages/teamstatus-template
  - Don’t worry about making up for last week
- **Tuesday (Jan. 21):** Technical design document due
  - Be prepared to discuss in team meetings with Prof. the following week
- **Friday, January 31:** End of Sprint 1
  - 18 days left in Sprint 1 (including a holiday, Jan 20)
To discuss

- **Artists**
  - Why are so few artists taking Art 118 class?

- **Lab lighting**
  - Controllers don’t appear to be working right – any insight?
  - Post-It Notes

- **Jim’s GitHub**
  - JimWhiteheadUCSC

- **Unity**
  - See Piazza for license code
  - Ends October 10, 2014
  - Unity 4.x, iOS, Android, Blackberry, Team 4.x
Meetings with Professor

- **Tuesday**
  - 9am - Orbion
  - 3pm - Apocalyptia
- **Wednesday**
  - 5pm – CARGO
  - 5:45pm - DJ
- **Friday**
  - 10am – 4WARD
  - 10:45am - Cocooned
  - 11:30am - Astral
  - 1pm – White Shark
  - 1:45pm - Launch
  - 2:30pm – Horror Stories
  - 3:30pm - Forsaken
  - 4:15pm - Intrigue
  - 5:00pm – LOD
  - 5:45pm - Immunogen
Meeting Preparation

- Presentation on your game design
- Art Direction
  - What are your goals for the emotional reaction you want the player to have?
  - What are 5-10 pieces of reference art that are consistent with this emotional reaction, and convey your current thinking on art direction
  - **Include your artists** in this process
  - Who will give the presentation, and have the reference art on their machine
Technical Design Document

- Details also on web
- Required elements
  - Title section
  - UML structure diagram for your design
    - Printout of 1 or more pages
  - UML sequence diagrams for your design
    - If creating your own game engine, must include sequences for:
      - Initialization, menu system, main game loop, player collision, enemy collision, and end of level
    - If using an existing game engine, must include sequences that show how your code is called from the game engine
      - Less clear to me which sequences are most important here, depends on the game engine. In general want to represent interesting and/or complex sequences
  - Also: need to be prepared to discuss this in weekly meetings with Professor week of Jan. 21
Lab cleanup

- Need teams to sign up for week-long duty to tidy the lab
  - 9 weeks left in Winter
  - We have 14 teams
  - Each team signs up for one week – some weeks have two teams
  - Will do signup on Piazza

- Every individual is responsible for cleaning up after themselves
  - Especially food containers, napkins, bottles, etc.

- But, sometimes people forget

- Team should make a task for cleaning, put it on scrum board, assign to a person

- Team on lab cleanup needs to:
  - Ensure overflowing trash cans are emptied to bin outside in 3rd floor courtyard (anytime during week)
  - By 5pm Monday and 5pm Friday (unless things get out of control, then more often):
    - Pick up food containers, bottles, etc.
    - Pick up stray craft materials, pens, etc. and return to drawers
    - Clean off tables in conference rooms and big circular table
    - Report any major soda/food spills to me, so we can call cleanup crews
    - Put controllers/game boxes/etc. away (tidy up game area)
    - Report any cleaning materials needed
Introduction to UML
Introduction to UML

- The Unified Modeling Language (UML) consists of a collection of diagrams for describing a software design.

- Creating a UML description forces a team to develop a software design before diving into the nitty-gritty of writing code.

Diagram:

- **Diagram**
  - **Structure Diagram**
    - Class Diagram
    - Component Diagram
    - Object Diagram
    - Composite Structure Diagram
    - Deployment Diagram
    - Package Diagram
  - **Behaviour Diagram**
    - Activity Diagram
    - Use Case Diagram
    - State Machine Diagram
    - Interaction Diagram
    - Interaction Overview Diagram
    - Communication Diagram
    - Timing Diagram

Class Diagram:

UC SANTA CRUZ
Class diagrams

- A class diagram describes static aspects of your object oriented design

- Classes are drawn as boxes.

- Members are listed inside the box. Fields appear in the top sub-box, methods in the bottom sub-box
  - Access indicated by + (public), - (private), # (protected) and ~ (package)

- Classes are connected together with lines indicating class relationships

```
<table>
<thead>
<tr>
<th>SequentialBehavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>- stepCounter: int</td>
</tr>
<tr>
<td>- child: Step</td>
</tr>
<tr>
<td>~ getNextStep(): Step</td>
</tr>
<tr>
<td>~ addChild(s: Step)</td>
</tr>
<tr>
<td>~ removeChild()</td>
</tr>
</tbody>
</table>
```
Generalization links

- Generalization links indicate subclass relationships
  - Parent/child relationships
- An open arrow points to the parent

```plaintext
Behavior
#parent: GoalStep
#priority: int
#signature: String
~failBehavior()
~succeedBehavior()

SequentialBehavior
-stepCounter: int
-child: Step
~getNextStep(): Step
~addChild(s: Step)
~removeChild()
```
Aggregation links

- Aggregation indicates that instances of one class will contain instances of another class
  - In aggregation, the lifespan of the enclosed instances is independent of the lifespan of the enclosing instance

- Container classes (lists, hashtables, etc.) will always have aggregation links to what they contain, though many classes will contain member instances of other classes

```
Behavior
#parent: GoalStep
#priority: int
#signature: String

~failBehavior()
~succeedBehavior()

SequentialBehavior
-stepCounter: int
-child: Step

~getNextStep(): Step
~addChild(s: Step)
~removeChild()
```

```
Step

GoalStep
```
Composition links

- Composition links indicate that one class contains instances of another class, but the contained class is created and destroyed with the instance class
  - The contained instances will be destroyed when the containing instance is destroyed

- In C++, this is the difference between a member variable of type MyClass* and MyClass

```plaintext
Behavior
#parent: GoalStep
#priority: int
#signature: String
-propertyTable: NamePropertySupport
~failBehavior()
~succeedBehavior()

NamedPropertySupport

Step

SequentialBehavior
-stepCounter: int
-child: Step
~getNextStep(): Step
~addChild(s: Step)
~removeChild()

GoalStep
```
Realization

- Realization links relates a class that implements (realizes) a behavior specified by another model element, to the model element that specifies this behavior.

- In Java, classes that implement an interface realize the interface.
- In C++, classes that are children of a pure abstract class realize behavior specified by the pure abstract class.
Dependency links

- Dependency links represent arbitrary relationship between classes, where a change made to one class may require a change to another class
  - The arrow points from the dependent towards the independent class

- You’ll want to use link labels for dependency links

- On the class diagram, only indicate important dependency relationships (ones that help communicate in the team)
There are many good, free UML tools

Some that seem interesting:

- Gliffy (http://www.gliffy.com/) - web based
- Creately (http://creately.com/) – web based
- UMLet (http://umlet.com/)
- Dia (https://live.gnome.org/Dia)
Introduction to UML

- The Unified Modeling Language (UML) consists of a collection of diagrams for describing a software design.

- Creating a UML description forces a team to develop a software design before diving into the nitty-gritty of writing code.
UML Sequence diagrams

- Sequence diagrams capture the temporal order of interactions between system objects (might be literal code objects or subsystems)

- You should capture a sequence diagram for each of the important chains of events that happens in your game
  - Collision detection
  - Controller event (player pressing a button)
  - Main loop
  - NPC action selection (if there are significant inter-object interactions)
  - Interface interactions
Lifelines

- Lifelines represent object instances or roles
  - The box at the top of the lifeline names the object instance or role
  - A dotted line under the box indicates how long the object lives

```
Object name  c: CollisionManager
             └── Class
```
Messages

- Messages indicate method invocations between objects
- Bars on the lifeline indicate the period of time during which execution/handling of the message takes place
- Dotted lines indicate return
  - Returns are optional, though it’s recommended to use them if you are returning a value

```
Main

1 : updateCollisions()
```

```
c : CollisionManager

2
```
Instances may create new instances

- If you need to show one instance creating another, use the <<create>> message stereotype
Self messages and call stacks

- Objects/roles can send messages to themselves
  - If working with object instances, this represents a method on an instance invoking other methods on the same instance
- Bars are nested to indicate the call stack
Indicating if-then semantics on individual messages

- Guards are used to indicate if-then semantics on individual messages
  - The message is sent only if the test in square braces is true
Combined fragments frame a subset of object interactions. They are used to show that a subsequence has alternatives (if then blocks) or loops. Here’s a loop example:

```
foreach character in CollisionManager
    char : Character
    hero : Character

5 : handleHeroCollisions()
6 : checkCollision()
7 : collision
8 [collision] : isAttacking()
9 : handleHeroObjectCollisions()
```
Alt fragments

- Alt fragments indicate if-then blocks
- Use interaction operands to indicate alternative sequences
Miscellaneous features

- All the message types used in the example are blocking message types (normal method invocations)
- Non-blocking messages (e.g. sending a request via an IPC mechanism like sockets) are indicated with open arrowhead
  - May need to use annotation if UML tool doesn’t support this
- Parallel processes can be indicated with a *par* combined fragment
Modeling Issues
Modeling issues

- A major UML modeling divergence is whether your game:
  - Creates its own game engine
    - I.e., using some game library like XNA that handles only a small part of game needs (scene graph, collision, pathing, NPC AI, etc.)
  - Uses an existing game engine

- Create own game engine
  - Relatively few connections to underlying game framework
  - Majority of code running the game is created by the team
  - Will have classes for gathering input, managing scenes (object lists), collision, AI, animation, scrolling, etc.
  - Team has large degree of control over the structure of software
Modeling issues (cont’d)

- Using existing game engine
  - Varies by engine
  - Typically takes the form of creating subclasses of existing classes of game engine framework
  - Can also involve creating code using scripting language inside game engine
  - UML diagram tends to look like a series of mostly disconnected classes (since connections occur inside game engine)
  - Important to show which game engine classes are subclasses off of
  - Need to understand clearly responsibilities handled by engine, and by code your team writes
Refresher: Strategy Pattern
Problem: Changing AI Behavior

Consider:
- AI behavior of an opponent often changes while the game is running
  - If it gets close to the player, or some other game event occurs

How can this be accomplished in code?
- Do not want to destroy opponent object, and create new one with changed AI behavior
  - I.e., creating a separate subtype of Opponent for each separate opponent behavior isn’t dynamic enough
- However, also do not want to encode all possible behaviors inside each type of opponent
  - Ideally want to re-use AI behaviors across many types of opponent
    - I.e., putting a big switch/case statement inside each Opponent type won’t work either
      - “Switch statement” and “duplicate code” bad code smells
Strategy Pattern

- Client creates instance of IStrategy subclass
  - `myStrategy = new IStrategySubclass();`
  - Or, can be given subclass instance in constructor
- Inside the client, write code that relates only to IStrategy
  - `myStrategy.Algorithm();`
  - Will call the Algorithm method on subclass currently assigned to myStrategy

*Show example of Strategy pattern in Visual C#*
Design Principles

- Two design principles at play here
  - Favor composition over inheritance
    - More flexible to compose IStrategy subclass with Client than to make lots of Client subclasses
  - Program to Interfaces, not implementations
    - If you program to an interface, are not tied to a specific class that implements the interface
    - Can easily create another implementation of the interface, and use that instead
      - If you program to an interface, substituting a new subclass of that interface is a small change