Lecture 11

Architecture Design
Take Away for Today

- What should the lead programmer do?
- How do CRC cards aid software design?
  - What goes on each card?
  - How do you lay them out?
  - What properties should they have?
- How do activity diagrams aid design?
  - How do they relate to CRC cards?
- Difference between design & documentation
Role of Lead Programmer

- Make high-level **architecture decisions**
  - How are you splitting up the classes?
  - What is your computation model?
  - What is stored in the data files?
  - What third party libraries are you using?

- **Divide** the work among the **programmers**
  - Who works on what parts of the game?
  - What do they need to coordinate?
Architecture: The Big Picture
Identify Modules (Subsystems)

- **Modules**: logical unit of functionality
  - Often reusable over multiple games
  - Implementation details are hidden
  - API describes interaction with rest of system

- Natural way to break down work
  - Each programmer decides implementation
  - But entire team must agree on the API
  - Specification first, then programming
Example: Physics Engines

- API to manipulate objects
  - Put physics objects in “container”
  - Specify their connections (e.g. joints)
  - Specify forces, velocity
- Everything else hidden from user
  - Collisions detected by module
  - Movement corrected by module
Relationship Graph

- Shows when one module “depends” on another
  - Module A calls a method/function of Module B
  - Module A creates/loads instance of Module B

- **General Rule:** Does $A$ need the API of $B$?
  - How would we know this?

![Diagram showing Module 1, Module 2, and Module 3 connected with arrows.]

Module 1 does not “need” to know about Module 3
Relationship Graph

- Edges in relationship graph are often **directed**
  - If \( A \) calls a method of \( B \), is \( B \) aware of it?

- But often undirected in architecture diagrams
  - Direction clear from other clues (e.g. layering)
  - Developers of both modules should still agree on API

Does Module 1 need to know about Module 2?
Dividing up Responsibilities

- Each programmer has a module
  - Programmer owns the module
  - Final word on implementation

- Owners collaborate w/ neighbors
  - Agree on API at graph edges
  - Call meetings “Interface Parties”

- Works, but…
  must agree on modules and responsibilities ahead of time
Nested (Sub)modules

- Can do this **recursively**
  - Module is a piece of software
  - Can break into more modules

- Nested APIs are **internal**
  - Only needed by module owner
  - Parent APIs may be different!

- Critical for very **large groups**
  - Each small team gets a module
  - Inside the team, break up further
  - Even deeper hierarchies possible
Architecture: The Big Picture

Diagram of game architecture showing:
- Game Engine
  - Input Devices
  - Discrete Simulation Engine
  - Compiler
  - Data Management Layer
- Physics Engine
- AI Engine (e.g. Pathfinding)
- Game Content
  - Character Scripts
  - Character Data
  - UI Elements
  - Models and Textures
  - Sounds
- Player
  - GUI
  - Rendering Engine
  - Audio Engine
  - Nested Module

Designer or Modder

Architecture Design
How Do We Get Started?

- Remember the design caveat:
  - Must agree on module responsibilities first
  - Otherwise, code is duplicated or even missing

- Requires a high-level architecture plan
  - Enumeration of all the modules
  - What their responsibilities are
  - Their relationships with each other

- Responsibility of the lead architect
Design: CRC Cards

- **Class-Responsibility-Collaboration**
  - **Class**: Important class in subsystem
  - **Responsibility**: What that class does
  - **Collaboration**: Other classes required
    - May be part of another subsystem

- **English description of your API**
  - Responsibilities become **methods**
  - Collaboration identifies **dependencies**
# CRC Card Examples

<table>
<thead>
<tr>
<th><strong>Responsibility</strong></th>
<th><strong>Collaboration</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pathfinding</strong>: Avoiding obstacles</td>
<td>Game Object, Scene Model</td>
</tr>
<tr>
<td><strong>Strategic AI</strong>: Planning future moves</td>
<td>Player Model, Action Model</td>
</tr>
<tr>
<td><strong>Character AI</strong>: NPC personality</td>
<td>Game Object, Level Editor Script</td>
</tr>
</tbody>
</table>

## Scene Model

<table>
<thead>
<tr>
<th><strong>Responsibility</strong></th>
<th><strong>Collaboration</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumerates game objects in scene</td>
<td>Game Object</td>
</tr>
<tr>
<td>Adds/removes game objects to scene</td>
<td>Game Object</td>
</tr>
<tr>
<td>Selects object at mouse location</td>
<td>Mouse Event, Game Object</td>
</tr>
</tbody>
</table>
## CRC Card Examples

### Controller vs. AI Controller

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pathfinding: Avoiding obstacles</td>
<td>Game Object, Scene Model</td>
</tr>
<tr>
<td>Strategic AI: Planning moves</td>
<td>Player Model, Action Model</td>
</tr>
<tr>
<td>Character AI: NPC personality</td>
<td>Game Object, Level Editor Script</td>
</tr>
</tbody>
</table>

### Model vs. Scene Model

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumerates game objects in scene</td>
<td>Game Object</td>
</tr>
<tr>
<td>Adds/removes game objects to scene</td>
<td>Game Object</td>
</tr>
<tr>
<td>Selects object at mouse location</td>
<td>Mouse Event, Game Object</td>
</tr>
</tbody>
</table>
Creating Your Cards

- Start with MVC Pattern
  - Gives 3 basic subsystems
  - List responsibilities of each
  - May be all that you need (TemperatureConverter)

- Split up a module if
  - Too much for one person
  - API for module too long

- Don’t need to nest (yet)
  - Perils of ravioli code

<table>
<thead>
<tr>
<th>Module</th>
<th>Responsibility</th>
<th>Collaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Creating Your Cards

- Start with MVC Pattern
  - Gives 3 basic subsystems
  - List responsibilities of each
  - May be all that you need
    (TemperatureConverter)
- Split up a module if
  - Too much for one person
  - API for module too long
- Don’t need to nest (yet)
  - Perils of ravioli code

### Module 1
<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Module 2
<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Application Structure

Ownership

Collaboration

Architecture Design
**Application Structure**

- **Collaboration**
  - Must import class/interface
  - Instantiates an object **OR**
  - Calls the objects methods

- **Ownership**
  - Instantiated the object
  - Subset of collaboration
Avoid Cyclic Collaboration

collaborates with
Y

X

collaborates with
Z

collaborates with
Controller

Y

X

Architecture Design
Avoid Cyclic Collaboration

- **Example**: Lab 3
  - Ship fires projectiles
  - Must add to game state

- Originally all in model
  - Ship referenced game state
  - And game state stored ship
  - **Cyclic Reference**

- We added a new controller
  - It references game state
  - Only it adds to game state
  - **Cycle broken**
Avoid Cyclic Collaboration

- **Example**: Lab 3
  - Ship fires projectiles
  - Must add to game state

- Originally all in model
  - Ship referenced game state
  - And game state stored ship
  - **Cyclic Reference**

- We added a new controller
  - It references game state
  - Only it adds to game state
  - **Cycle broken**

![Diagram showing relationships between GameplayController.java, Game State, and Ship.java]
**Alternative: Interfaces**

- **Relationships are for APIs**
  - Implementation not relevant
  - Can be class or interface

- **Interfaces can break cycles**
  - Start with single class
  - Break into many interfaces
  - Refer to interface, not class

- **Needed if actions in model**
  - Abstracts game state
  - Hides all but relevant data
Architecture: The Big Picture

Simple (Planar) Graph

Architecture Design
CRC Index Card Exercise

### Class 1
<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>Class 2</td>
</tr>
<tr>
<td>...</td>
<td>Class 3</td>
</tr>
<tr>
<td>...</td>
<td>Class 4</td>
</tr>
</tbody>
</table>

Try to make collaborators adjacent

If cannot do this, time to think about nesting!

### Class 2
<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Class 3
<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Class 4
<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Designing Class APIs

- Make classes formal
- Turn responsibilities into methods
- Turn collaboration into parameters

### Scene Model

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enumerates game objects</td>
<td><code>Iterator&lt;GameObject&gt; enumObjects()</code></td>
</tr>
<tr>
<td>Adds game objects to scene</td>
<td><code>void addObject(GameObject)</code></td>
</tr>
<tr>
<td>Removes objects from scene</td>
<td><code>void removeObject(GameObject)</code></td>
</tr>
<tr>
<td>Selects object at mouse</td>
<td><code>GameObject getObject(MouseEvent)</code></td>
</tr>
</tbody>
</table>
Documenting APIs

- Use a formal **documentation style**
  - What **parameters** the method takes
  - What values the method **returns**
  - What the method does (**side effects**)  
  - How method responds to errors (**exceptions**)  

- Make use of **documentation comments**
  - **Example**: JavaDoc in Java
  - Has become de-facto-standard (even used in C++)
/**
 * Returns an Image object that can then be painted on the screen.
 * <p>
 * The url argument must specify an absolute [@link URL]. The name argument is a specifier that
 * is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the image exists. When this applet
 * attempts to draw the image on the screen, the data will be loaded. The graphics primitives that
 * draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */

public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) { return null; }
}
Taking This Idea Further

- **UML**: Unified Modeling Language
  - Often used to specify class relationships
  - But expanded to model other things
  - **Examples**: data flow, human users

- How useful is it?
  - Extremely useful for documentation
  - Less useful for design (e.g. before implementation)
  - A language to program in another language
Activity Diagrams

• Define the **workflow** of your program
  • Very similar to a standard flowchart
  • Can follow simultaneous paths (threads)

• Are an **component** of UML
  • But did not originate with UML
  • Mostly derived from **Petri Nets**
  • One of most useful UML **design** tools

• Activity diagrams are only UML we use
Activity Diagram Example

- Find Beverage
- [found coffee]
- Put Coffee in Filter
- Put Filter in Machine
- Turn On Machine
- Brew Coffee
- [coffee dispensed]
- Pour Coffee
- [no coffee]
- Add Water to Reservoir
- Get Cups
- [found cola]
- Get Can of Cola
- [no cola]
- Drink Beverage
Activity Diagram Example

Start

Find Beverage

Put Coffee in Filter

Put Filter in Machine

Turn On Machine

Add Water to Reservoir

Brew Coffee

Get Coffee in Filter

[found coffee]

[no coffee]

[found cola]

[no cola]

Guard

Decision

Activity

Synch Bar

[coffee dispensed]

Pour Coffee

Drink Beverage

End
Activity Diagram Components

- **Synchronization Bars**
  - **In**: Wait until have happened
  - **Out**: Actions “simultaneous”
  - … or order does not matter

- **Decisions**
  - **In**: Only needs one input
  - **Out**: Only needs one output

- **Guards**
  - When we can follow edge
  - * is iteration over container
Asynchronous Pathfinding

Get Input

Determine Goal

Measure to Goal

Move Object

Draw

Reset Pathfinder

Find Path

[for each selected]

[new goal]

[path found]

[all objects checked]

*[for each object]

*[for each selected]

[all objects checked]

*[for each object]
Asynchronous Pathfinding

**Iteration**

Get Input

* [for each selected]

Measure to Goal

* [for each object]

[all objects checked]

Determine Goal

[for each object]

Move Object

Draw

[for each selected]

[all objects checked]

Reset Pathfinder

Find Path

Buffer

Task Separator
Asynchronous Pathfinding

- Get Input
- **Iteration**
  - *[for each selected]*
  - Measure to Goal
  - *for each object*
  - Determine Goal
  - [new goal]
  - [path found]
- *[for each object]*
- Move Object
- Draw

**Synchronization + Guard**
Think of as multiple outgoing edges (with guard) from bar

**Task Separator**
Expanding Level of Detail

1. Get Input
2. Determine Goal
3. Measure to Goal
4. [for each object]
5. [all objects checked]
6. [for each object]
7. Move Object
8. Draw

- Draw Background
- Draw Objects
- Draw HUD

Reset Pathfinder
Find Path

Architecture Design
Using Activity Diagrams

• Good way to identify major subsystems
  • Each action is a responsibility
  • Need extra responsibility; create it in CRC
  • Responsibility not there; remove from CRC

• Do activity diagram first?
  • Another iterative process
  • Keep level of detail simple
  • Want outline, not software program
Architecture Design

- Identify major subsystems in CRC cards
  - List responsibilities
  - List collaborating subsystems

- Draw **activity diagram**
  - Make sure agrees with CRC cards
  - Revise CRC cards if not

- Create **class API** from CRC cards
  - Recall intro CS courses: *specifications first!*
  - But **not** actually part of specification document
Programming Contract

- Once create API, it is a **contract**
  - Promise to team that “works this way”
  - Can change **implementation**, but not **interface**

- If change the interface, must **refactor**
  - Restructure architecture to support interface
  - May change the CRCs and activity diagram
  - Need to change any written code
Summary

- Architecture design starts at a high level
  - **Class-responsibilities-collaboration**
  - Layout as cards to visualize dependencies

- **Activity diagrams** useful for update loop
  - Outline general flow of activity
  - Identifies *dependencies* in the process

- Must formalize **class APIs**
  - No different from standard Java documentation
  - Creates a *contract* for team members
Where to From Here?

- Later lectures fill in architecture details
  - **Data-Driven Design**: Data Management
  - **2D Graphics**: Drawing
  - **Physics Engines**: Collisions, Forces
  - **Character AI**: Sense-Think-Act cycle
  - **Strategic AI**: Asynchronous AI
  - **Networking** (at end of course)

- But there is more design coming too