Lecture 12

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 MVC?
  • 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

Game Engine
- Input Devices
- Discrete Simulation Engine
- Compiler

Player
- GUI
- Rendering Engine
- Audio Engine

Designer or Modder
- Game Content
  - Character Scripts
  - Character Data
  - UI Elements
  - Models and Textures
  - Sounds

Programmer
- Physics Engine
- AI Engine (e.g. Pathfinding)
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*
Architecture: The Big Picture

Architecture Design
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?

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

Game Engine
- Input Devices
- Discrete Simulation Engine
- Compiler
- Data Management Layer

Game Content
- Character Scripts
- Character Data
- UI Elements
- Models and Textures
- Sounds

Physics Engine
- AI Engine (e.g. Pathfinding)

Player
- GUI
- Rendering Engine
- Audio Engine

Nested Module

Architectures

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

## AI Controller

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaboration</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>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>
## CRC Card Examples

### Controller

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaboration</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>

### 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>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibility</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<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*

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

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

Ownership

Collaboration

Model

Model

Model

Subcontroller

Subcontroller

View

Architectural Revisited
Application Structure

- **Ownership**
  - Instantiated the object
  - Superset of collaboration

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

collaborates with

collaborates with

Controller

collaborates with

collaborates with
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**
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

Game Content
Designer or Modder

Game Engine

AI Engine (e.g., Pathfinding)

Player

Output Devices

GUI

Rendering Engine

Audio Engine

Discrete Simulation Engine

Compiler

Data Management Layer

Character Scripts

Character Data

UI Elements

Models and Textures

Sounds
CRC Index Card Exercise

Try to make collaborators adjacent

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

If cannot do this, time to think about nesting!

<table>
<thead>
<tr>
<th>Class 2</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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class 3</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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class 4</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>
</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 defacto-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 [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.
 * <p>
 * @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

Add Water to Reservoir

Get Cups

Get Can of Cola

[found cola]

Put Filter in Machine

Turn On Machine

Brew Coffee

[coffee dispensed]

Pour Coffee

[no coffee]

[no cola]

Drink Beverage

[found coffee]

[no coffee]
Activity Diagram Example

Start

Find Beverage

Put Coffee in Filter

Put Filter in Machine

Turn On Machine

Add Water to Reservoir

Brew Coffee

[no coffee]

[no cola]

Decision

Guard

[found coffee]

[found cola]

Activity

Get Can of Cola

Get Cups

[coffee dispensed]

Pour Coffee

Drink Beverage

End

synch bar

architecture design
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
*[for each selected]

Determine Goal

Measure to Goal
*[for each object]

[all objects checked]

Move Object
*[for each object]

Draw
*[for each object]

[all objects checked]

[new goal]

[path found]

* Find Path

Reset Pathfinder
Asynchronous Pathfinding

Get Input

*[for each object]*

Measure to Goal

*[for each object]*

Determine Goal

[all objects checked]

*[for each object]*

Move Object

*[for each selected]*

*Get Input*

*[for each selected]*

Reset Pathfinder

[all objects checked]

*[for each object]*

Move Object

*[for each object]*

Draw

*[new goal]*

[all objects checked]

*[for each object]*

Move Object

*[path found]*

[all objects checked]

*[for each object]*

Move Object

*[path found]*

[all objects checked]

*[for each object]*

Move Object

Find Path

Buffer

Task Separator

Architecture Design
Asynchronous Pathfinding

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

Task Separator

Get Input

Iteration

* [for each selected]

Determine Goal

* [for each object]

Measure to Goal

[new goal]

[all objects checked]

Move Object

* [for each object]

Draw

Reset Pathfinder

Architecture Design
Expanding Level of Detail

Get Input

*[for each object]*

Measure to Goal

*[for each selected]*

Determine Goal

*[for each object]*

[all objects checked]

*[for each object]*

Move Object

*[for each object]*

Draw

[all objects checked]

Draw Background

Draw Objects

Draw HUD

Architecture Design

Reset Pathfinder

Find Path
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
• 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