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

Architecture Design

Game Engine

- Input Devices
- Discrete Simulation Engine
- Compiler

Game Content

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

Player

- GUI
- Rendering Engine
- Audio Engine

Programmer

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

Designer or Modder
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 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 with neighbors
  - Agree on API at graph edges
  - Call meetings “Interface Parties”

- Works, but…
  
  must agree on modules and responsibilities ahead of time

Architecture Design
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 modules
  - 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

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

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

Designer or Modder

Player
- GUI
- Rendering Engine
- Audio Engine

Nested Module
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

### Controller vs. 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>

### 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*

<table>
<thead>
<tr>
<th>Module 1</th>
<th>Module 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibility</td>
<td>Collaboration</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Avoid Cyclic Collaboration

collaborates with

collaborates with

Controller

collaborates with

Architecture Design
### 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>

- **Class 2**
  - Responsibility
  - Collaboration
  - ...
  - ...
  - ...

- **Class 3**
  - Responsibility
  - Collaboration
  - ...
  - ...
  - ...

- **Class 4**
  - Responsibility
  - Collaboration
  - ...
  - ...
  - ...

**If cannot do this, time to think about nesting!**
Designing Class APIs

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

<table>
<thead>
<tr>
<th>Scene Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Responsibility</strong></td>
</tr>
<tr>
<td>Enumerates game objects</td>
</tr>
<tr>
<td>Adds game objects to scene</td>
</tr>
<tr>
<td>Removes objects from scene</td>
</tr>
<tr>
<td>Selects object at mouse</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
  - C# uses an XML style for comments
/**
 * Returns an Image object that can then be painted on the screen. 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; }
}
/// <summary>
/// Returns an Image object that can then be painted on the screen. The url
/// argument must specify an absolute <see cref="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.
/// </summary>
/// <param name="url"">an absolute URL giving the base location of the image</param>
/// <param name="name"">the location of image, relative to the url argument</param>
/// <returns>the image at the specified URL</returns>
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

[coffee dispensed]

Turn On Machine
Brew Coffee

Pour Coffee
Drink Beverage

Get Can of Cola

[found cola]

Get Cups

Add Water to Reservoir

[no cola]

[no coffee]
Activity Diagram Example

Start

Find Beverage

Guard

[found coffee]

Put Coffee in Filter

Put Filter in Machine

Turn On Machine

Activity

Add Water to Reservoir

Brew Coffee

Synch Bar

Get Cups

Get Can of Cola

Synch Condition

[coffee dispensed]

Pour Coffee

[no coffee]

[no cola]

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

*[for each selected]

Determine Goal

Measure to Goal

*[for each object]

[all objects checked]

*[for each object]

Move Object

*[for each object]

Draw

[new goal]

*[path found]

Reset Pathfinder

Find Path
Asynchronous Pathfinding

Get Input

Iteration

*Determine Goal

*Measure to Goal

*Get Input

*Move Object

*Draw

[all objects checked]

*[for each object]

*[for each selected]

Reset Pathfinder

Buffer

Task Separator

[all objects checked]

*[for each object]

[new goal]

[all objects checked]

*[for each object]

*[for each selected]

[all objects checked]

*[for each object]

[all objects checked]

*[for each object]

[all objects checked]

*[for each object]
Asynchronous Pathfinding

**Iteration**

- Get Input
- *[for each selected]*
- Determine Goal
- Measure to Goal
- *[for each object]*
- [all objects checked]*
- *[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. Measure to Goal
   - [for each object]
   - [all objects checked]
3. Determine Goal
   - [for each selected]
4. Draw
   - Draw Background
   - Draw Objects
   - Draw HUD
5. Move Object
6. Draw
7. Find Path
8. Reset Pathfinder
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 Specification

- 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!*
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**
  - Same detail as Java documentation
  - Creates a *contract* for team members