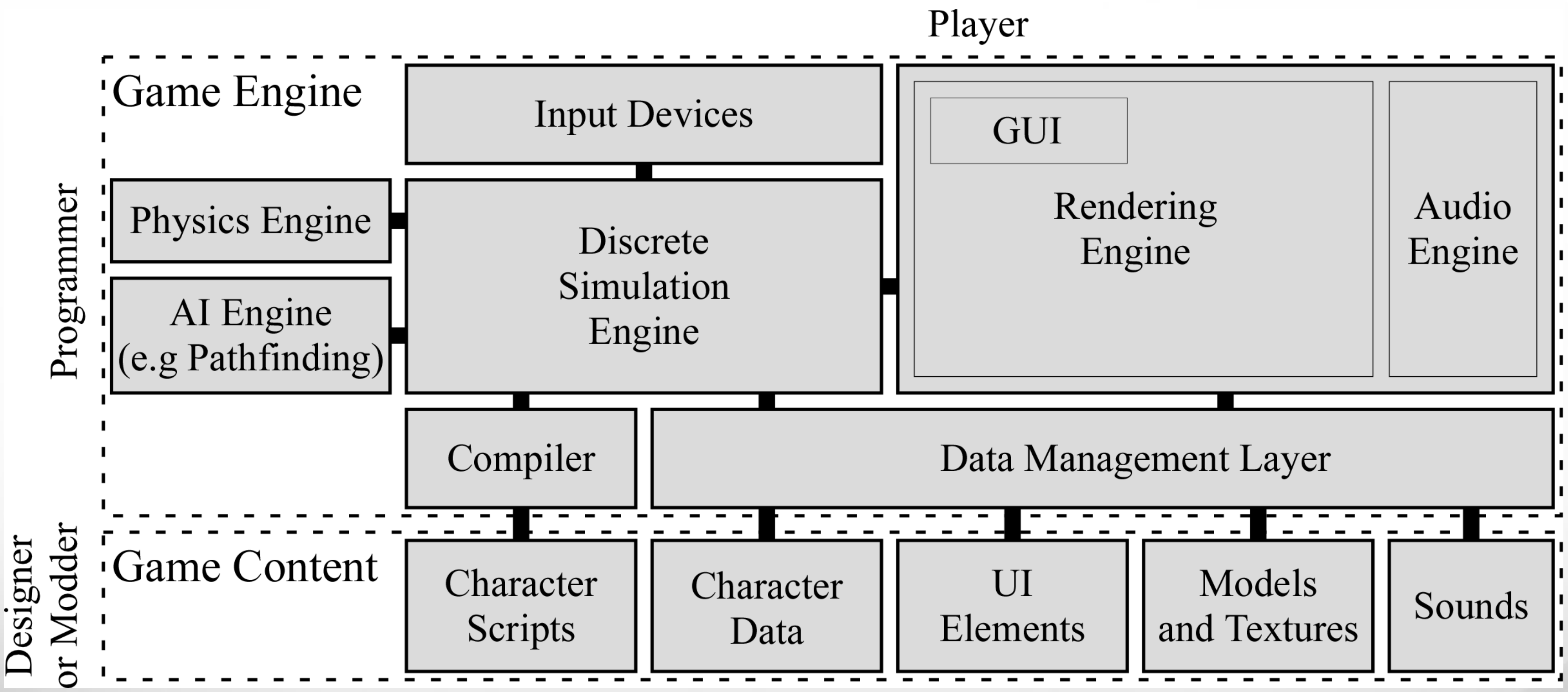
CIS 3110:

Architecture Design

Questions for Today’s Lecture

- How do you develop large-scale software?
  - How do you manage a large(ish) developer team?
  - How do you divide up responsibilities?
  - What happens when you change something?
- Are architecture & programming different?
  - Can you do one without the other?
- What tools can help with architecture?

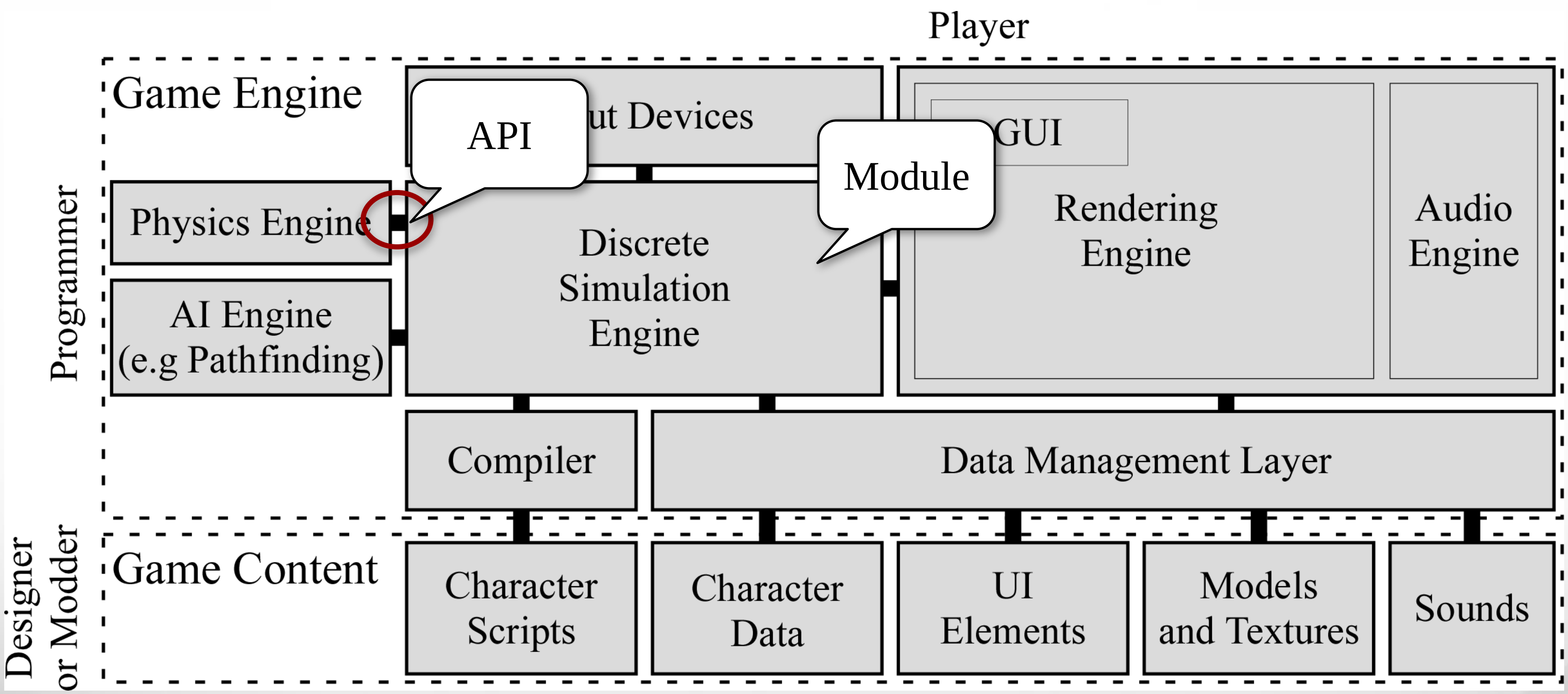
Architecture Diagram for a Computer Game



Modules (Subsystems)

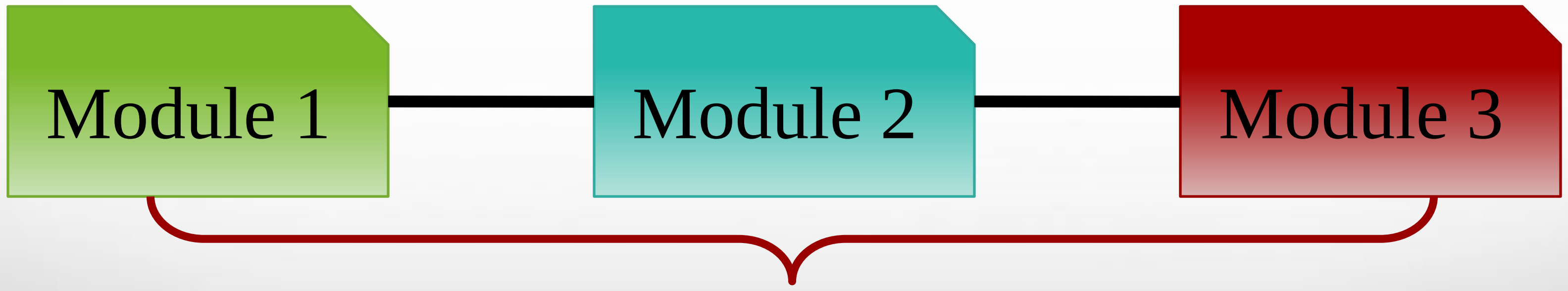
- **Module**: logical unit of functionality
  - Often reusable over applications
  - Implementation details hidden behind API
- API: **Application Programming Interface**
  - Collections of methods/functions
  - Results of calling them fully documented
  - But implementation details are hidden
- **Idea**: Split modules across programmers

Architecture Diagram for a Computer Game



Relationship Graph

- Shows when one module “depends” on another
  - Module A calls a method/function of Module B
  - OO: Module A creates/loads instance of Module B
- **General Rule**: Does *A* need the API of *B*?

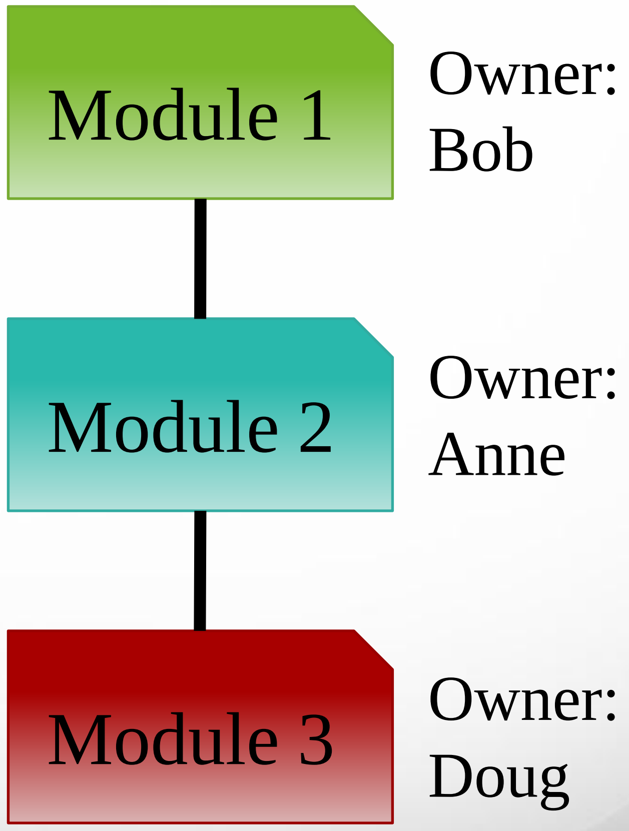


Module 1 does not “need” to know about Module 2

Dividing up Responsibilities

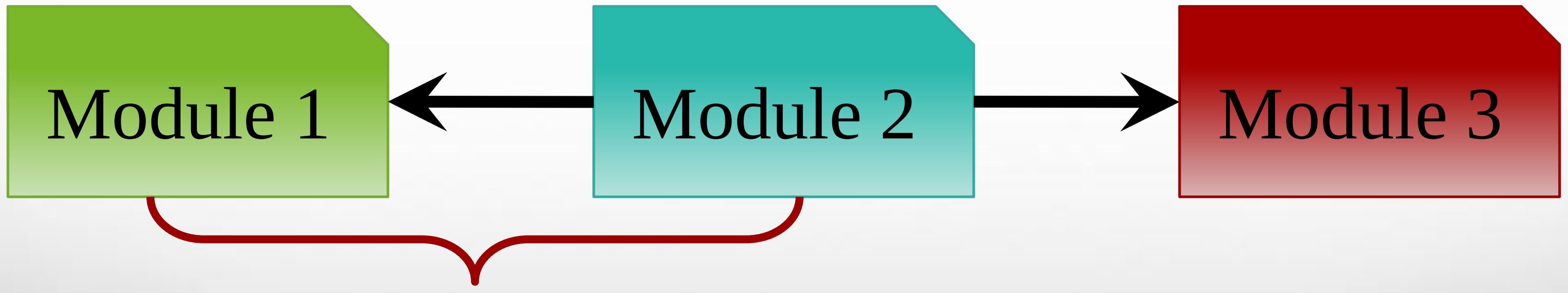
- Give each programmer a module
  - Programmer **owns** the module
  - Final word on implementation
- Owners collaborate w/ **neighbors**
  - Agree on API at graph edges
  - “Interface Parties”
- Works, but…

  must agree on modules and responsibilities ahead of time



Relationship Graph

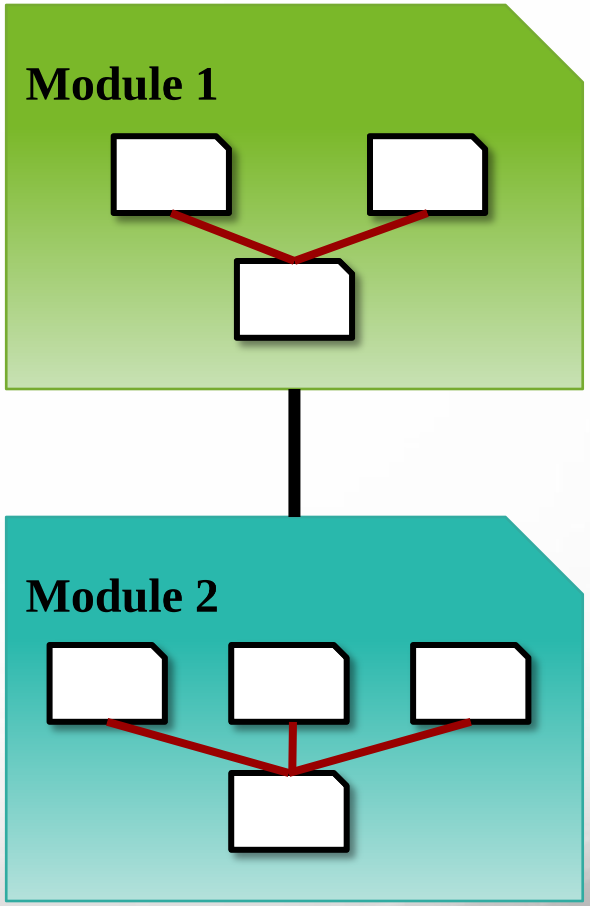
- Edges in relationship graph are often **directed**
  - If *A* calls a method of *B*, is *B* aware of it?
- But often undirected in architecture diagrams
  - Direction clear from other clues (e.g. layering)
  - Developers of both modules should still agree on API



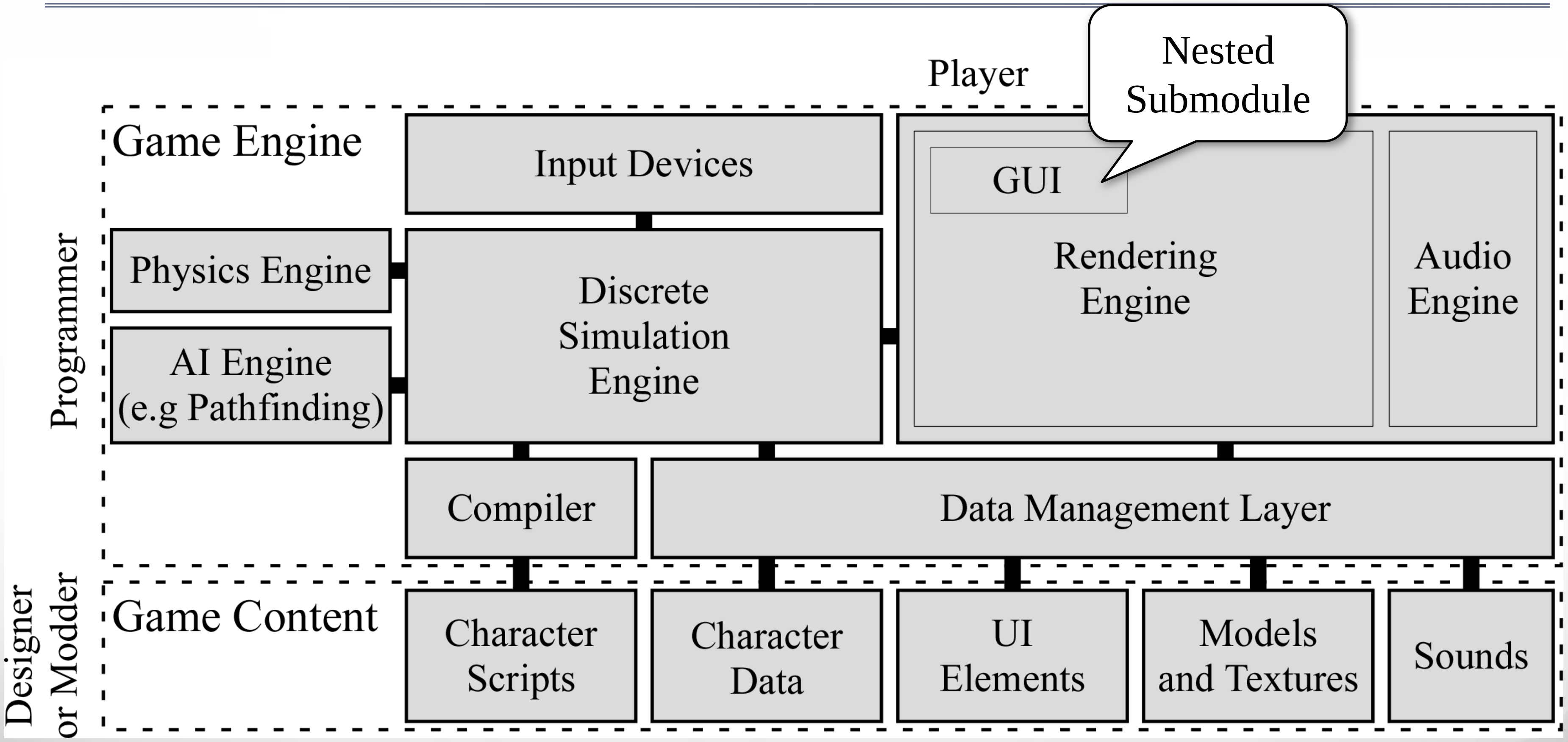
Does Module 1 need to know about Module 2?

Nested (Sub)modules

- Can do this **recursively**
  - Module is a piece of software
  - Can break it into (sub)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 Diagram for a Computer Game



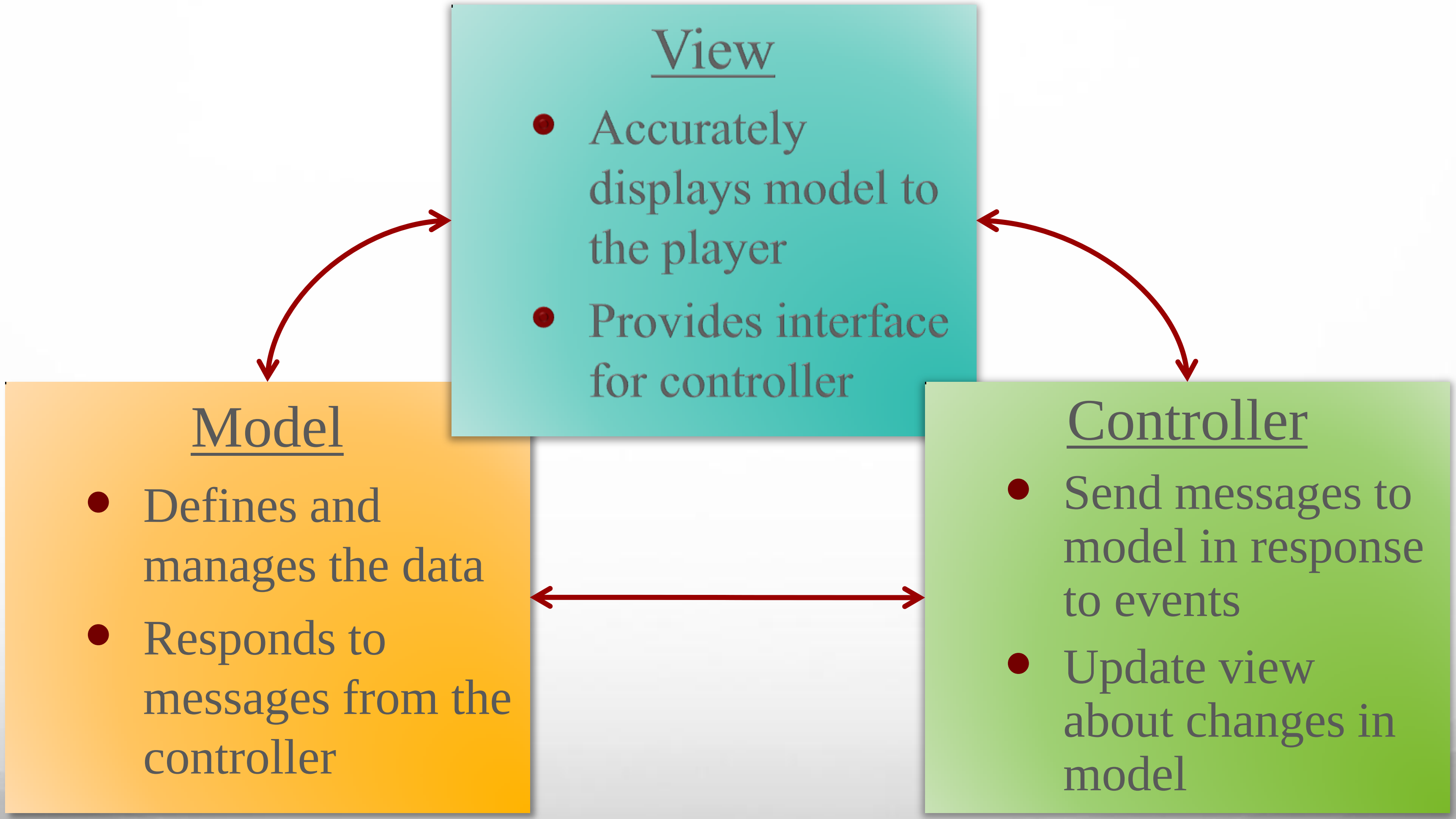
How Do We Get Started?

- 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
  - What their relationships are
- Responsibility of the lead architect

Architecture Patterns

- Essentially same idea as **software pattern**
  - Template showing how to organize code
  - But does not contain any code itself
  - Relationship graph + module guidelines
- Only difference is **scope**
  - **Software pattern**: simple functionality
  - **Architecture pattern**: complete program

Model-View-Controller Pattern



View

- Accurately displays model to the player
- Provides interface for controller

Model

- Defines and manages the data
- Responds to messages from the controller

Controller

- Send messages to model in response to events
- Update view about changes in model

Example: Temperature Converter

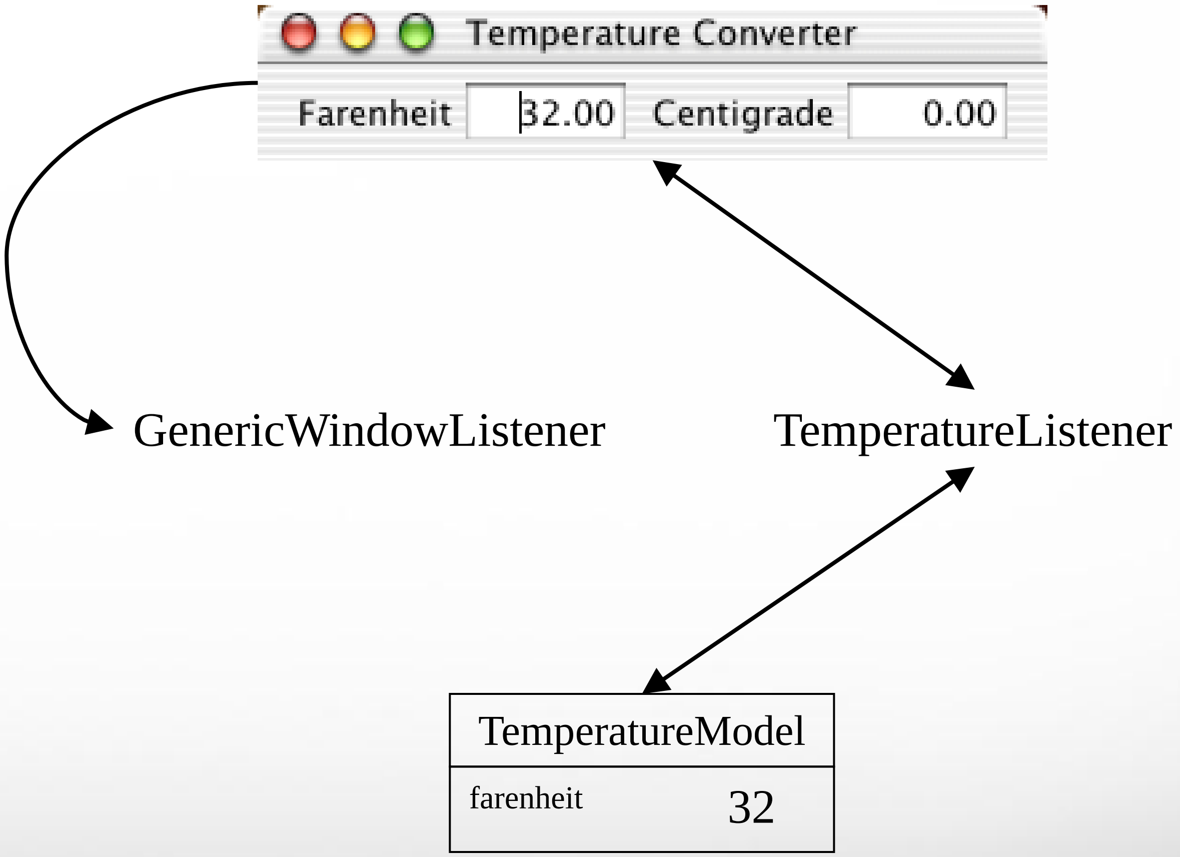
- Model: (TemperatureModel.java)
  - Stores one value: fahrenheit.
  - ADT abstraction presents two values.
- View: (TemperatureConverter.java)
  - Constructor creates objects and connects them.
  - Main method just calls constructor.
- Controller: Two Listeners
  - Respond to window events (GenericWindowListener.java)
  - Keep fields consistent (TemperatureListener.java)

MVC Illustrated

View

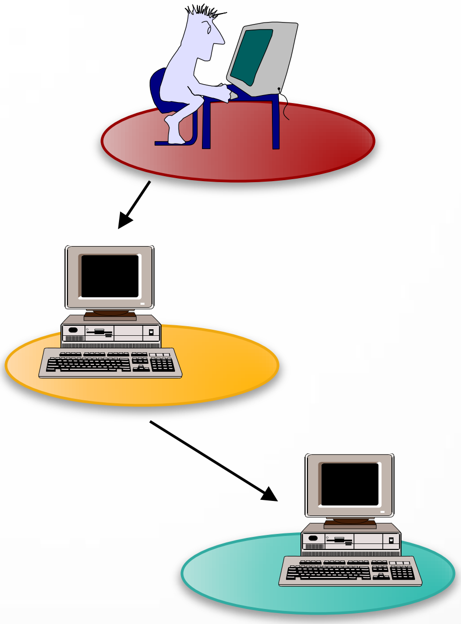
Controller

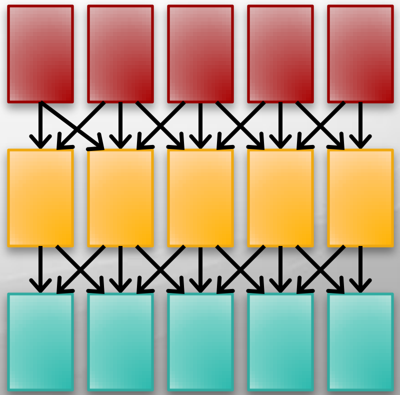
Model



Alternatives to MVC

- Model-View-Presenter
  - Presenter is lightweight controller
  - View handles controls for GUI
- Model-View-Viewmodel
  - Viewmodel translates model into new form
  - Useful for customizable UIs
- Three-tier Applications
  - Staple of web application development
- … and many others





Design: CRC Cards

- Class-Responsibility-Collaboration
  - **Class**: Represents your module (or *class* in OO)
  - **Responsibility**: What that module does
  - **Collaboration**: Other modules required
- Called “cards” because often on index card
- English description of your API
  - Responsibilities become methods/functions
  - Collaboration identifies dependencies

CRC Card Examples

Module Name

Controller **AI Controller**

Responsibility	**Collaboration**
Pathfinding: Avoiding obstacles	Game Object, Scene Model
Strategic AI: Planning future moves	Player Model, Action Model
Character AI: NPC personality	Game Object, Level Editor Script

Model **Scene Model**

Responsibility	**Collaboration**
Enumerates game objects in scene	Game Object
Adds/removes game objects to scene	Game Object
Selects object at mouse location	Mouse Event, Game Object

Creating Your Cards

- Architecture pattern is a start
  - Model-View-Controller
  - List responsibilities of each
  - May be all that you need (TemperatureConverter)
- Split a module if
  - Too much work for one person
  - API is too long for one module
- Don't need to nest (**yet**)
  - Perils of **ravioli code**

Module	
Responsibility	**Collaboration**
…	…
…	…
…	…

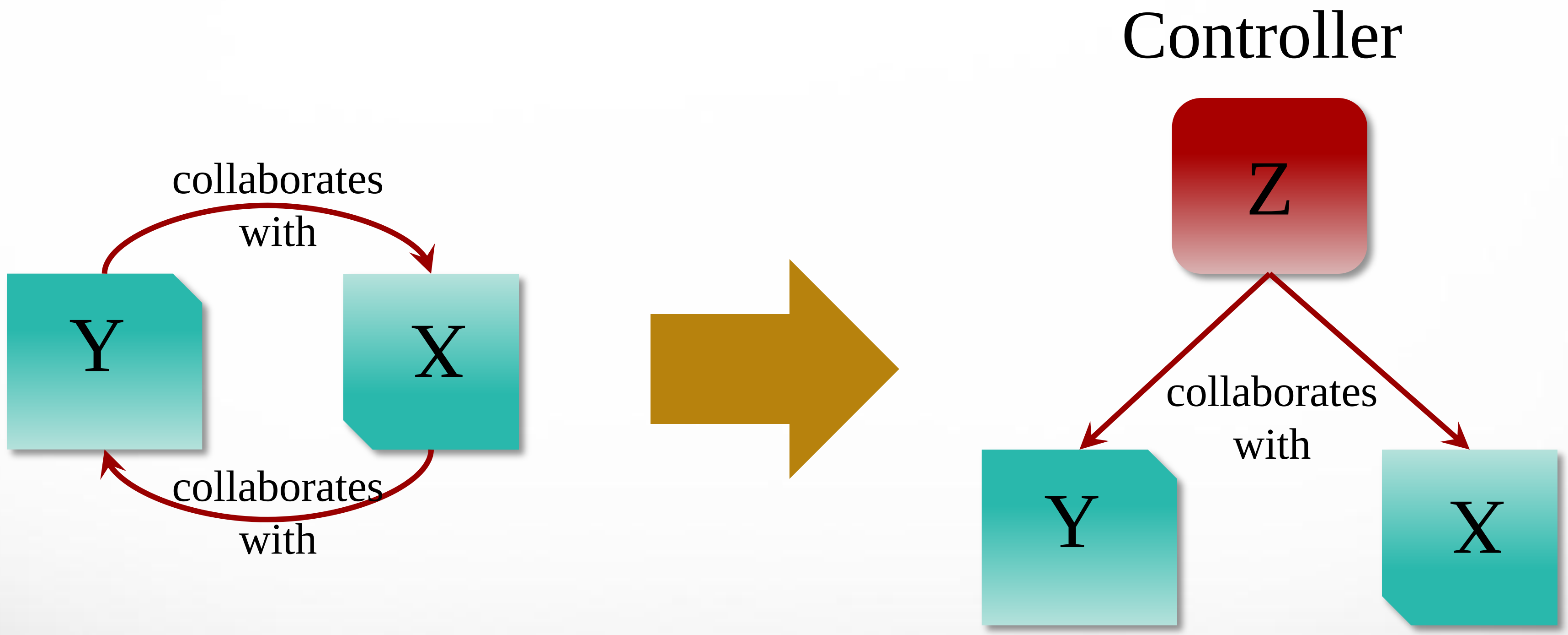
Creating Your Cards

- Architecture pattern is a start
  - Model-View-Controller
  - List responsibilities of each
  - May be all that you need (TemperatureConverter)
- Split a module if
  - Too much work for one person
  - API is too long for one module
- Don’t need to nest (**yet**)
  - Perils of **ravioli code**

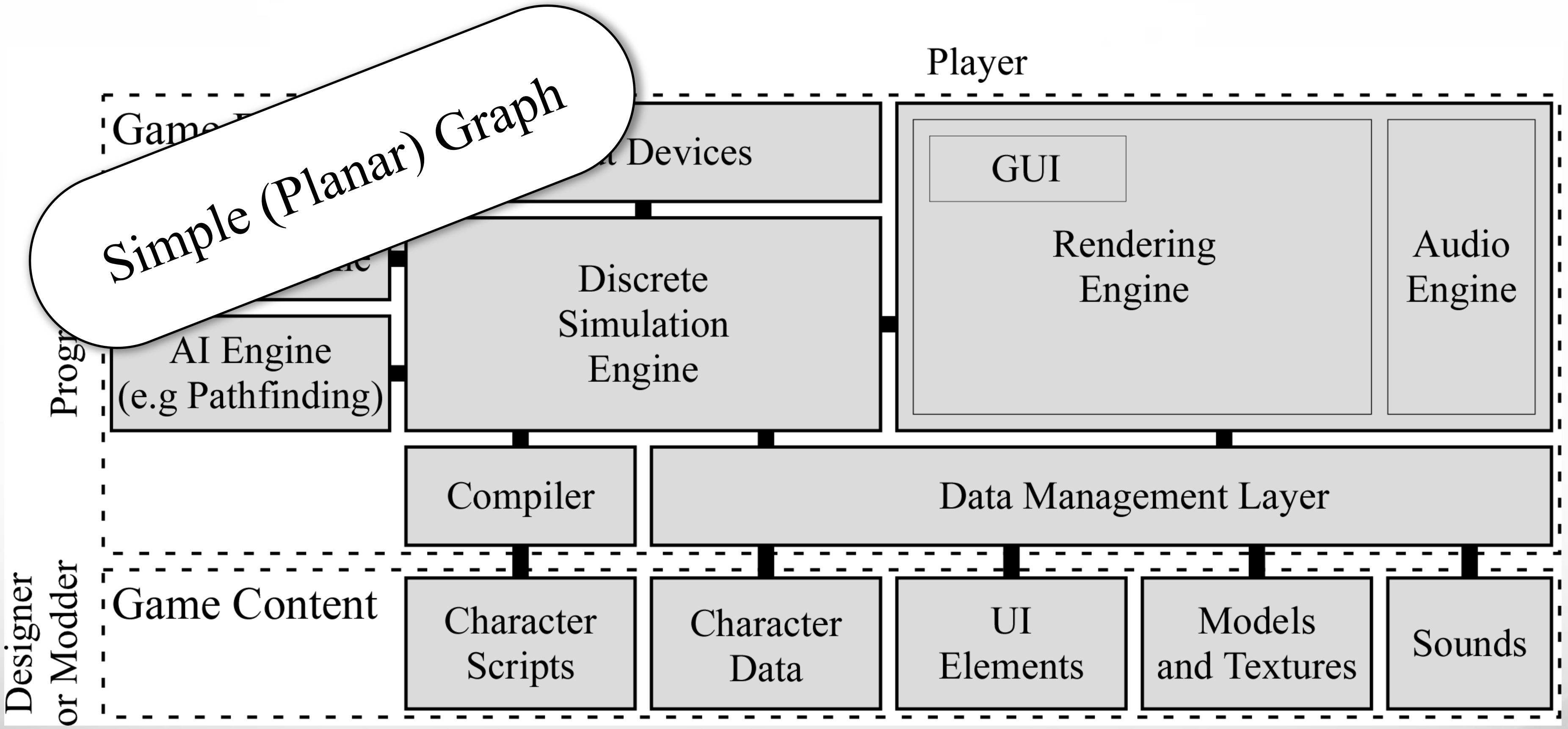
Module 1	
Responsibility	**Collaboration**
…	…
…	…
…	…

Module 2	
Responsibility	**Collaboration**
…	…
…	…
…	…

Avoid Cyclic Collaboration



Architecture Diagram for a Computer Game



CRC Index Card Exercise

Try to make collaborators adjacent

Module 1	
Responsibility	**Collaboration**
…	Module 2
…	Module 3
…	Module 4

Module 2	
Responsibility	**Collaboration**
…	…
…	…
…	…

Module 2	
Responsibility	**Collaboration**
…	…
…	…
…	…

Module 3	
Responsibility	**Collaboration**
…	…
…	…
…	…

If cannot do this, time to think about nesting!

Designing Module APIs

- Make CRC cards formal
- Turn responsibilities into methods/functions
- Turn collaboration into parameters

Scene Model	
Responsibility	**Method**
Enumerates game objects	`Iterator<GameObject> enumObjects()`
Adds game objects to scene	`void addObject(gameObject)`
Removes objects from scene	`void removeObject(gameObject)`
Selects object at mouse	`GameObject getObject(mouseEvent)`

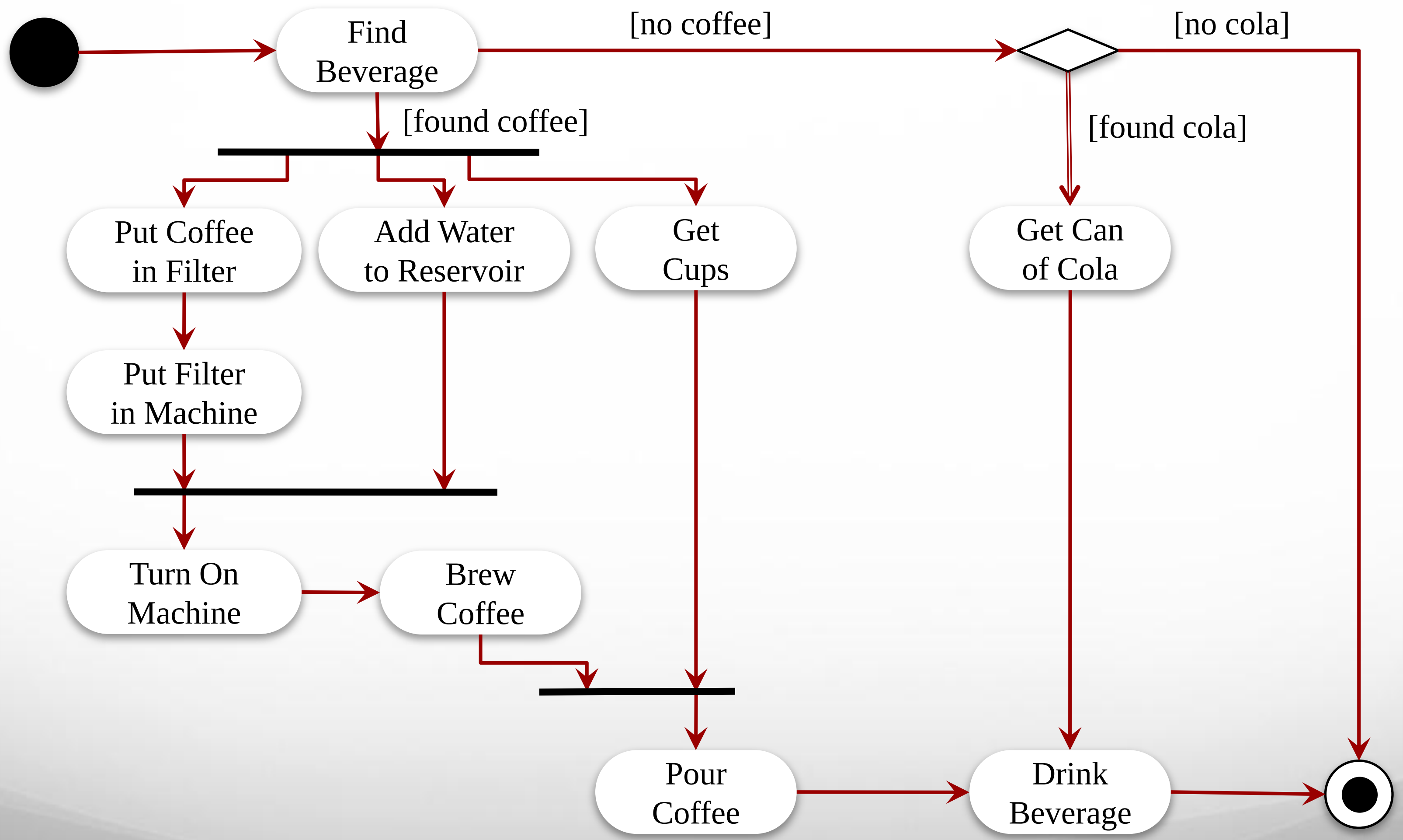
Taking This Idea Further

- UML: Unified Modeling Language
  - Allows you to specify class relationships
  - But models other things
  - Examples: data flow, human users
- How useful is it?
  - Using a language to program in another language
  - But many tools exist for working in UML
  - Use as little or as much as you find useful

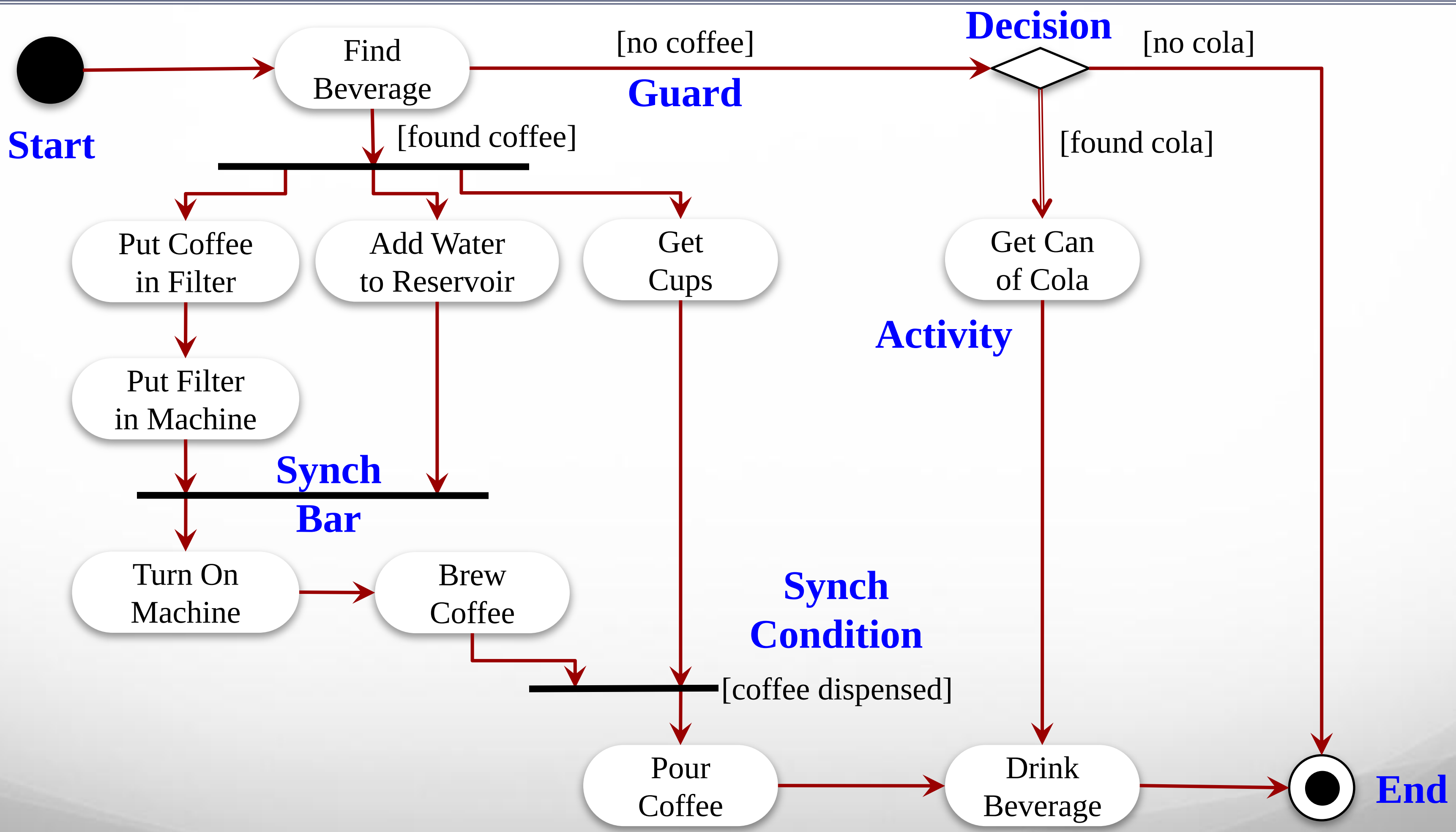
Activity Diagrams

- Define the **workflow** of your program
  - Very similar to a standard flowchart
  - Can follow simultaneous paths (threads)
- Are an component of UML
- Good way to identify modules
  - Each activity is a responsibility
  - Need extra responsibility; create it in CRC
  - Responsibility not there; remove from CRC

Activity Diagram Example



Activity Diagram Example



Summary

- Modules are important part of software design
  - Logical, self-contained unit of functionality
  - Elegant way to break up responsibilities in team
  - Use relationship graph to model dependencies
- Many tools to help with proper module design
  - Start with an architecture pattern
  - Use CRC cards to further break up modules
- UML is a popular tool for architecture design