the gamedesigninitiative at cornell university

Lecture 11

Architecture Design

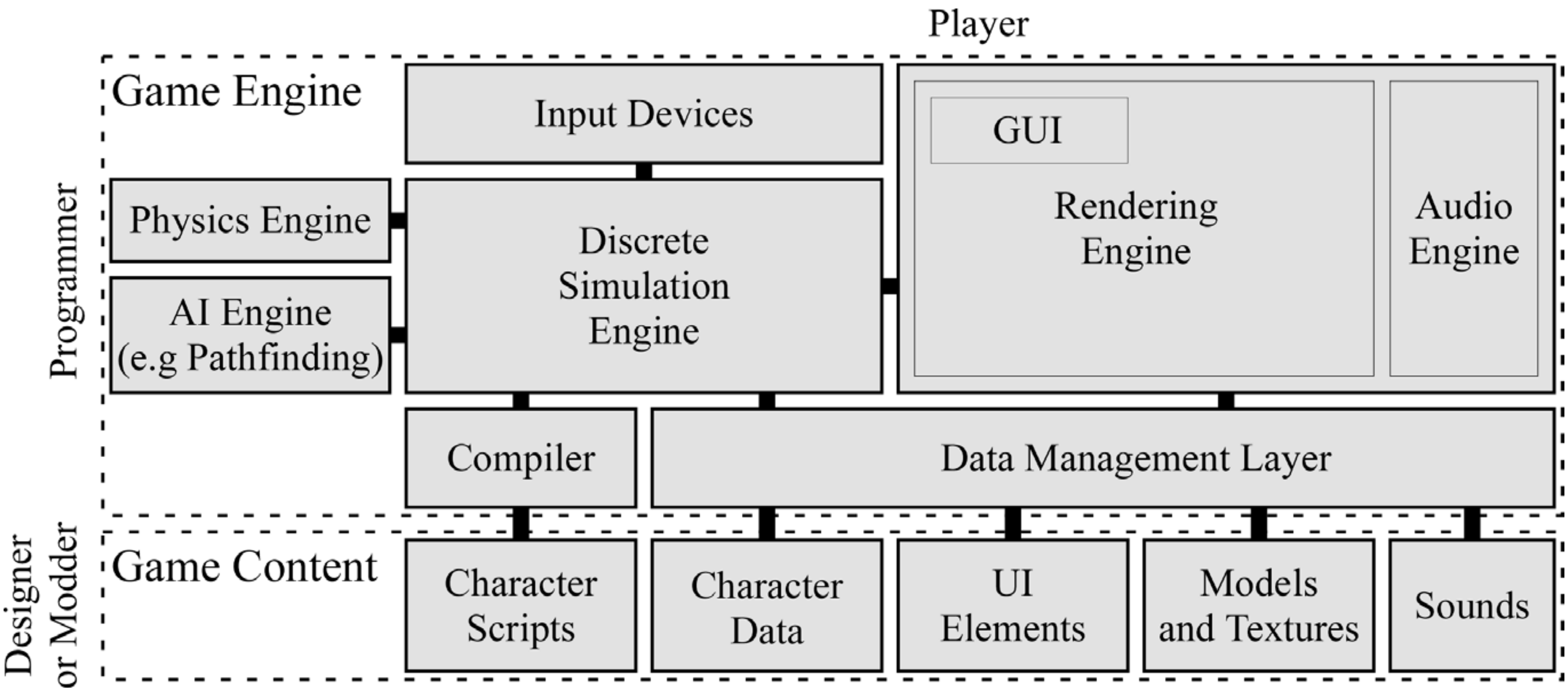
Take Away for Today

- 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?

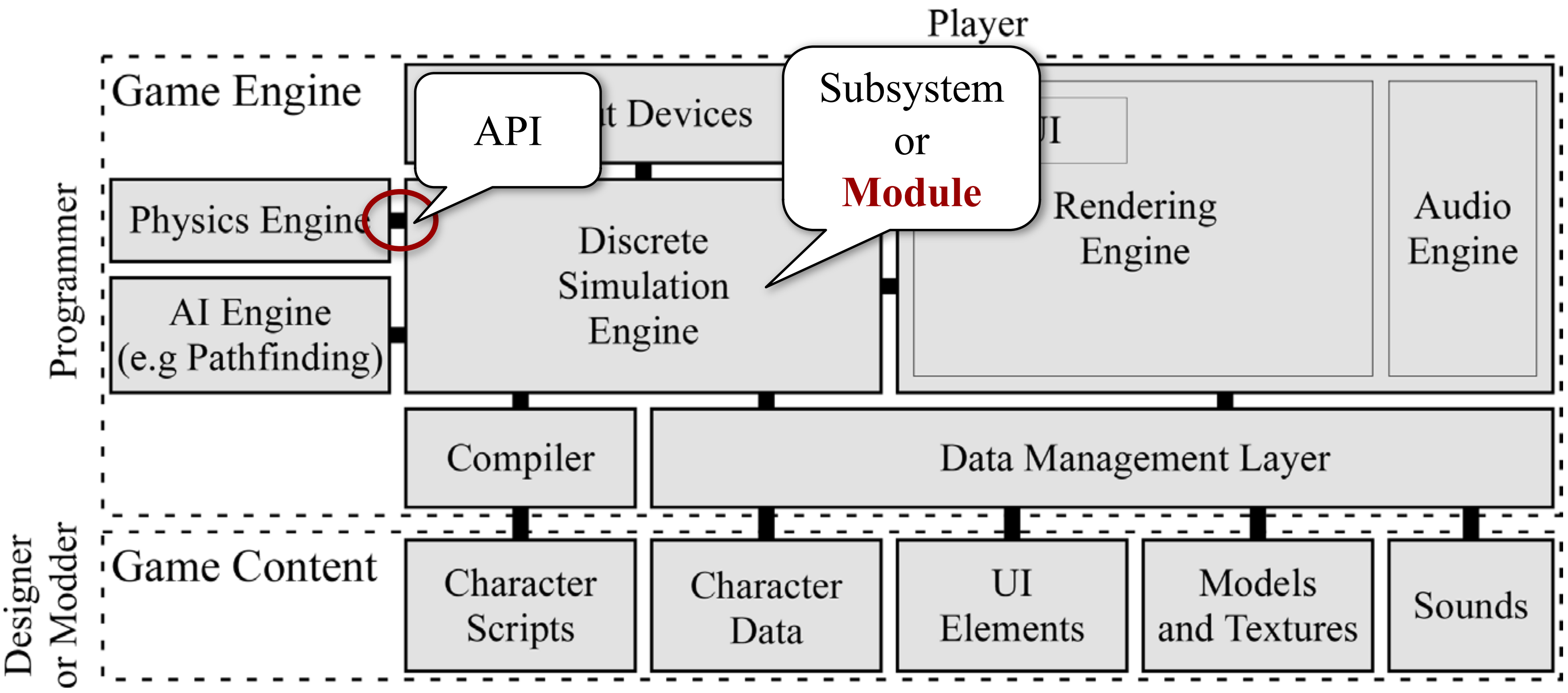
Architecture: The Big Picture



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**

Architecture: The Big Picture

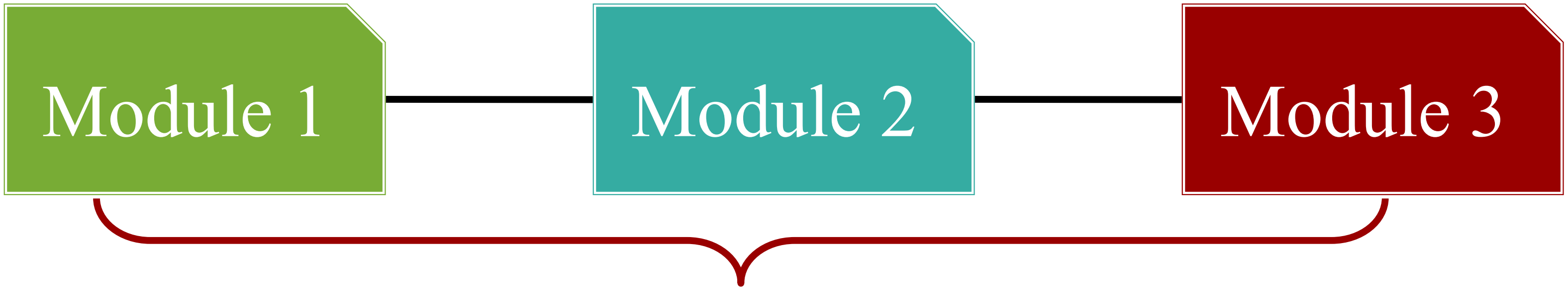


Example: Physics Engines

- 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

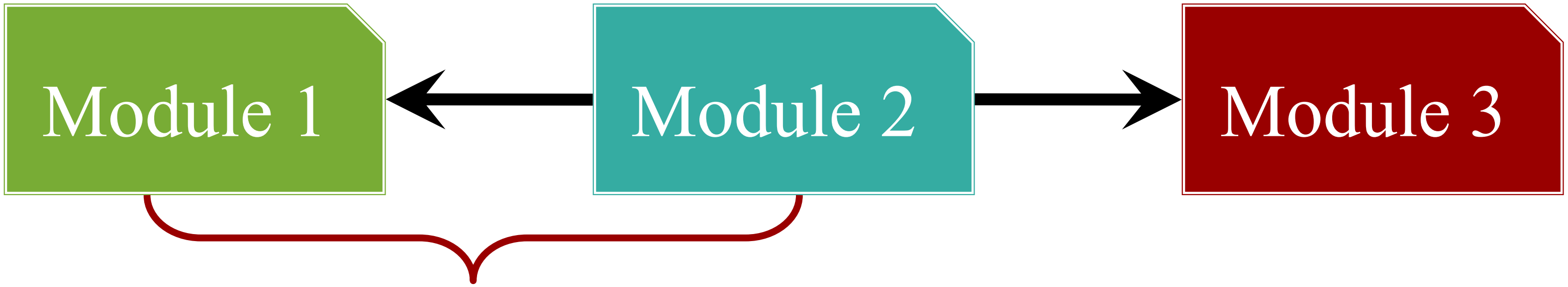
- Shows when one module “depends” on another
  - Module A calls a method/function of Module B
  - Module A creates/loads instance of Module B
- **General Rule**: Does *A* need the API of *B*?
  - How would we know this?



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

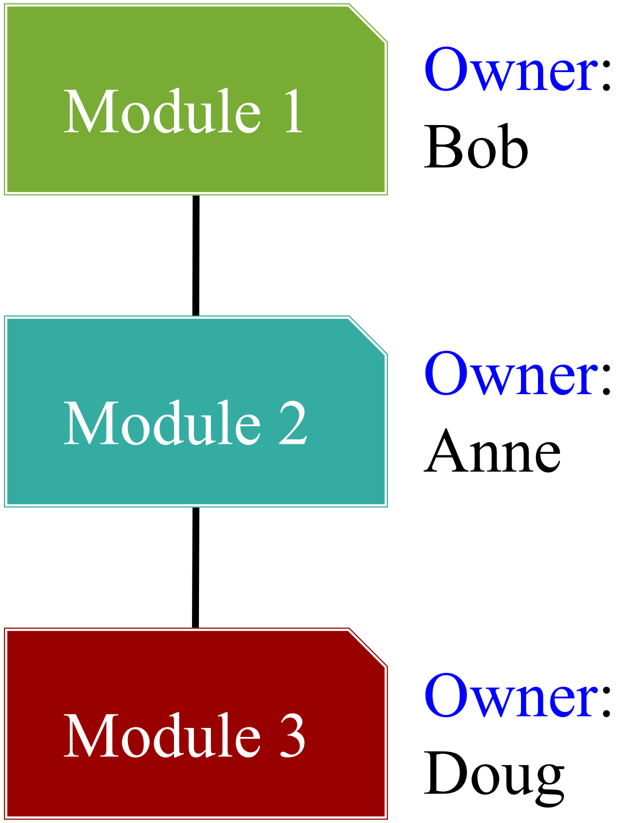


Does Module 1 need to know about Module 2?

Dividing up Responsibilities

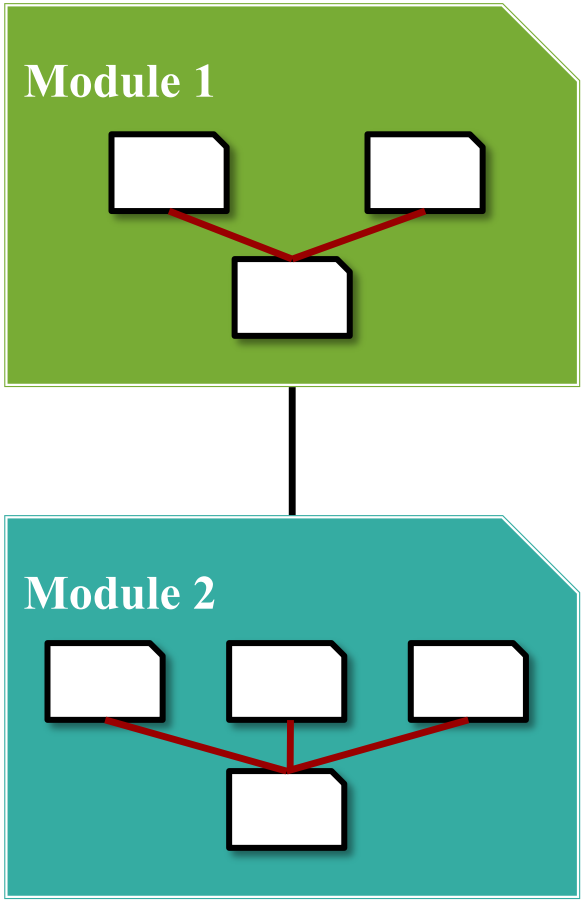
- Each programmer has a module
  - Programmer **owns** the module
  - Final word on implementation
- Owners collaborate w/ **neighbors**
  - Agree on API at graph edges
  - Call meetings "Interface Parties"
- Works, but…

  must agree on modules and responsibilities ahead of time

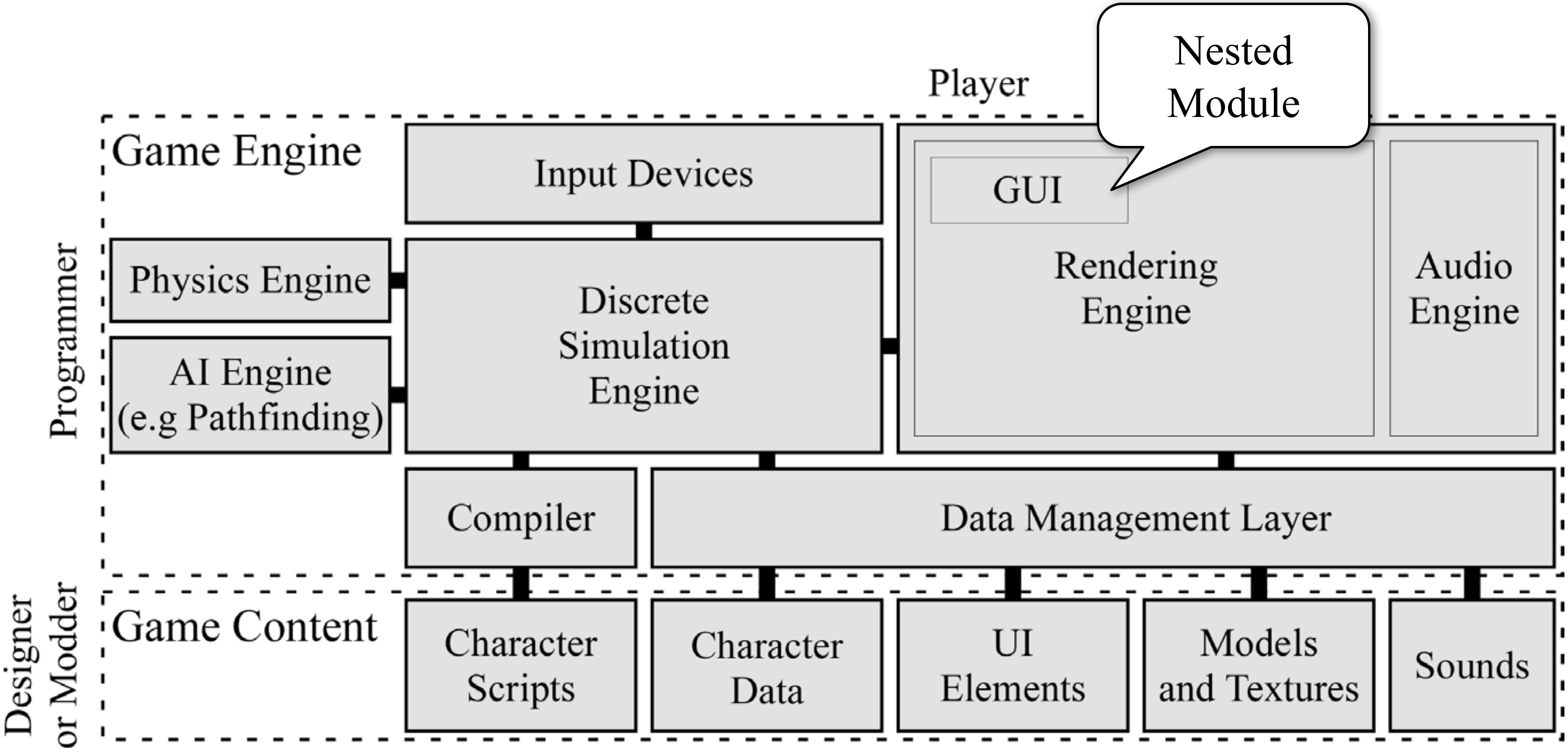


Nested (Sub)modules

- 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 modules
  - Inside the team, break up further
  - Even deeper hierarchies possible



Architecture: The Big Picture



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

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

CRC Card Examples

Controller **AI Controller**

Class Name

Responsibility	**Collaboration**
Pathfinding: Avoiding obstacles	Game Object, Scene Model
Strategic AI: Planning future moves	Player Model, Action Model
Character AI: Driving 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

- 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	
Responsibility	**Collaboration**
…	…
…	…
…	…
…	…
…	…
…	…
…	…
…	…

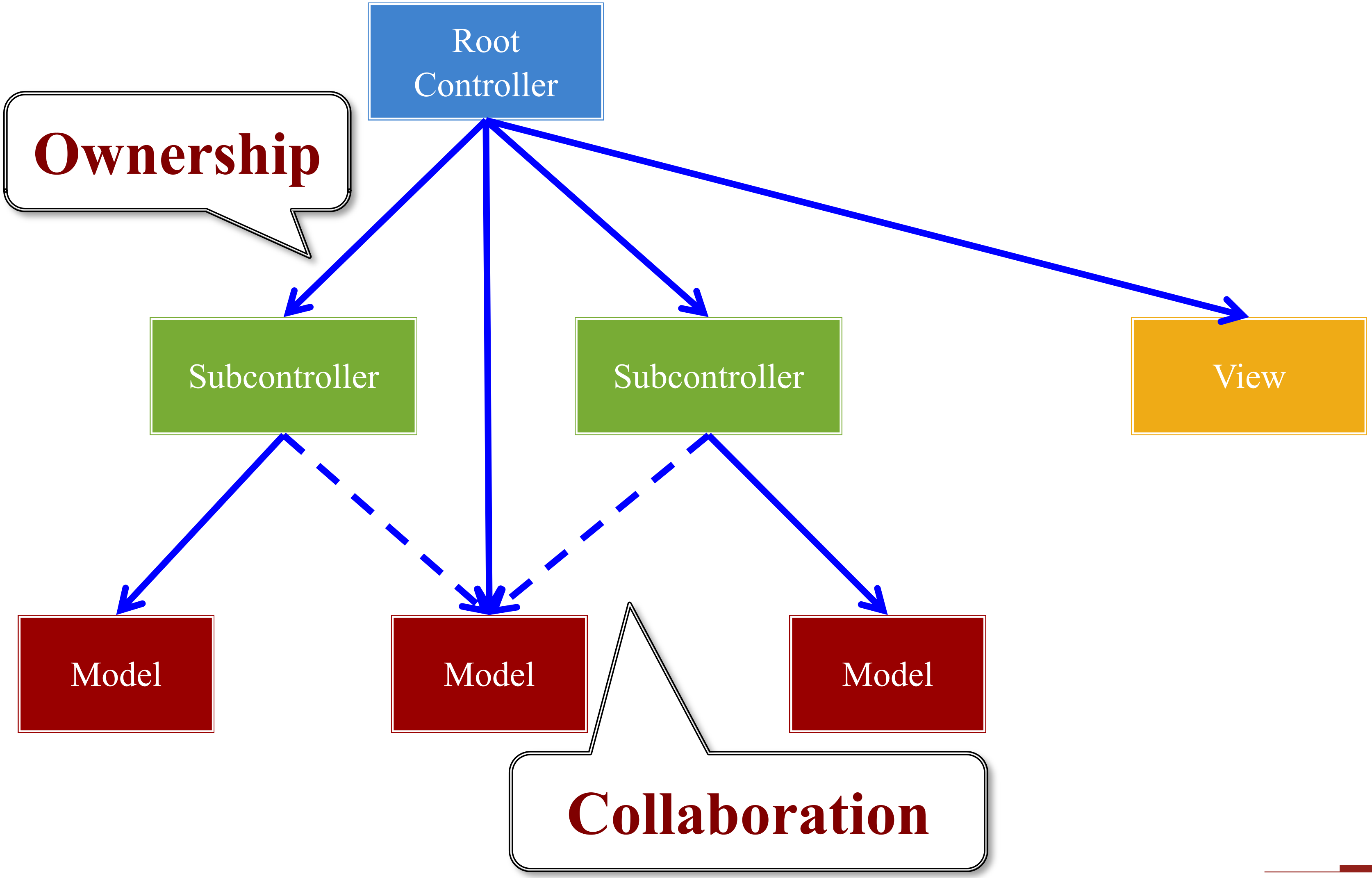
Creating Your Cards

- 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**

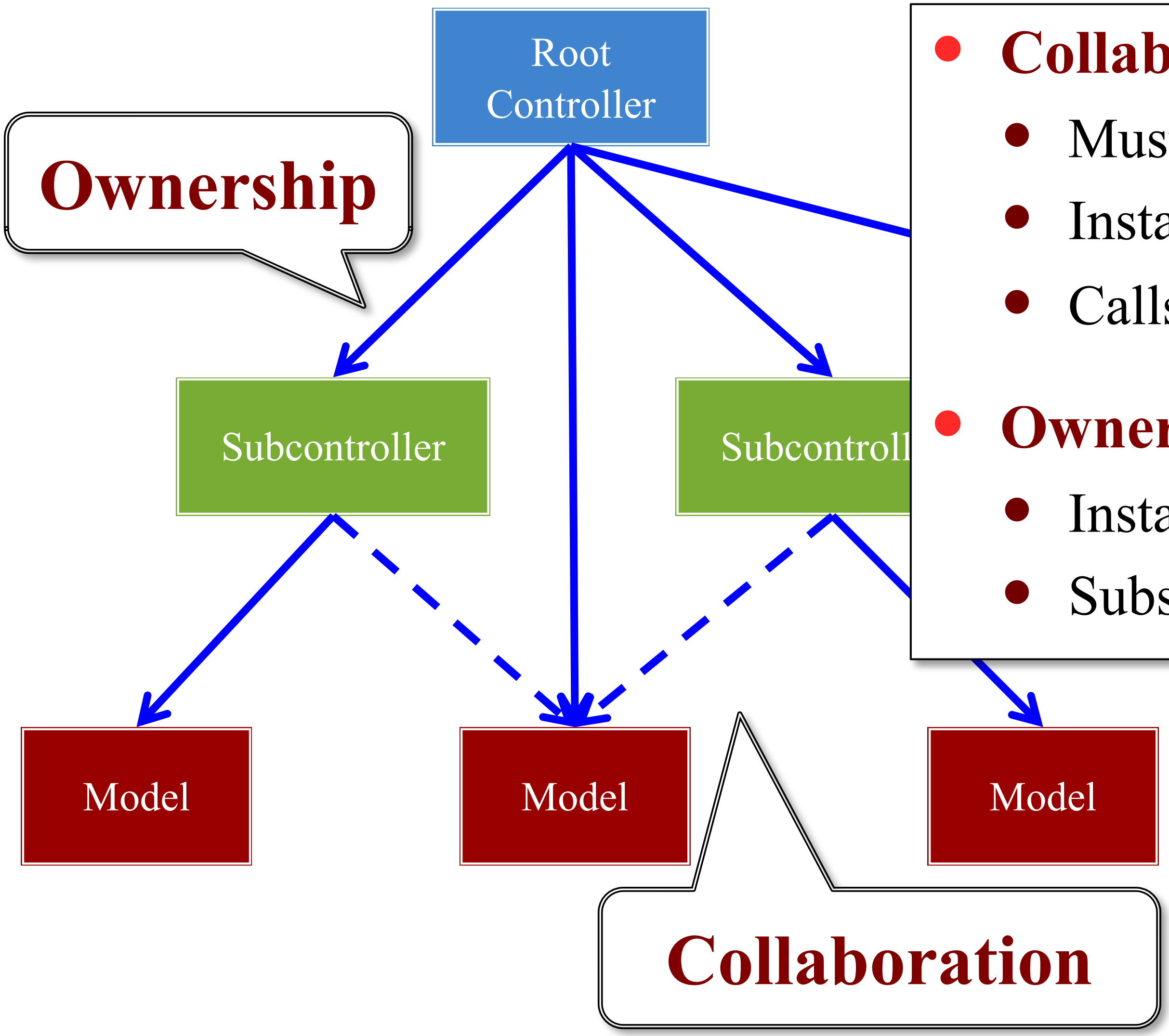
Module 1	
Responsibility	**Collaboration**
…	…
…	…
…	…

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

Application Structure

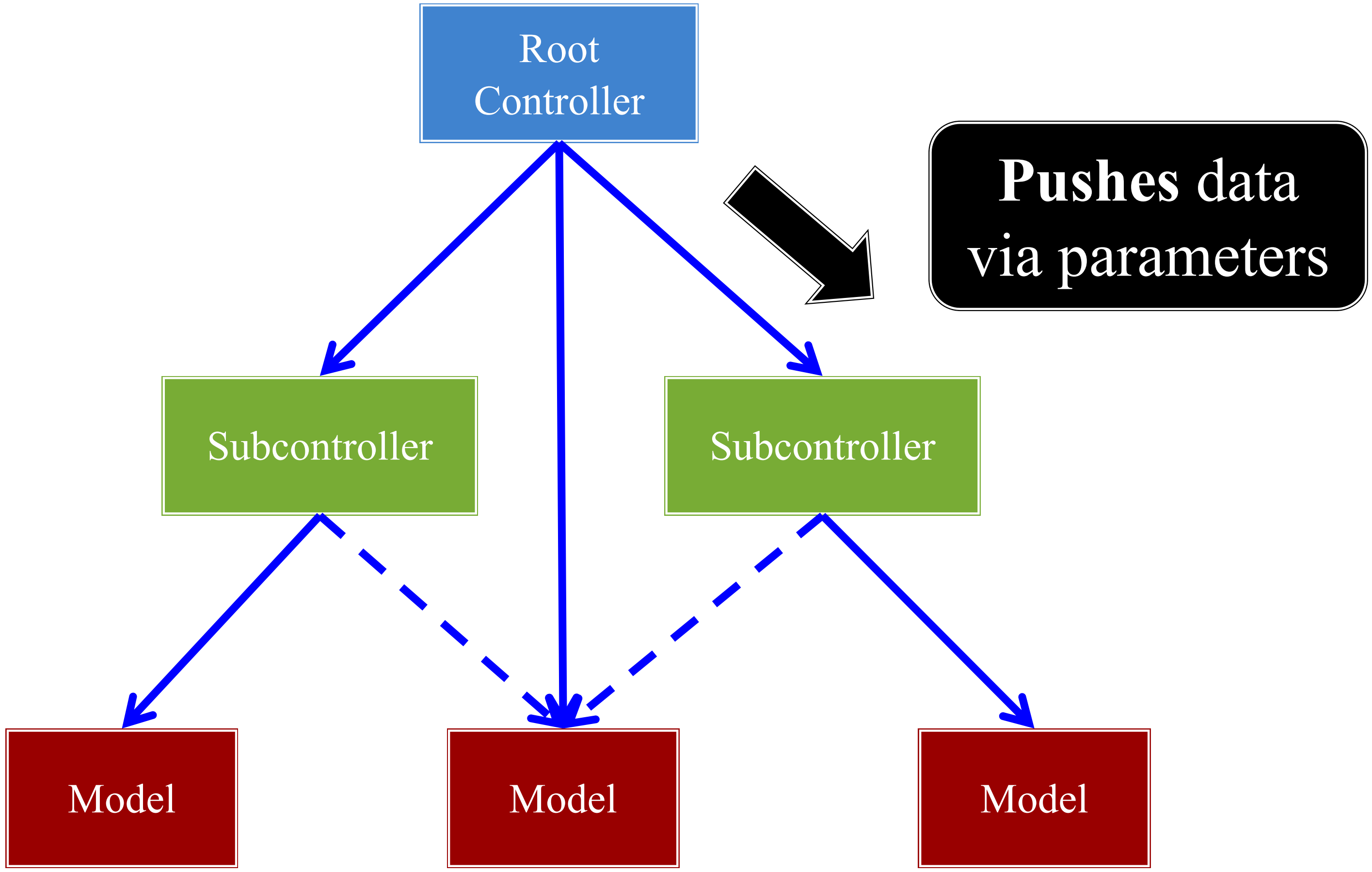


Application Structure



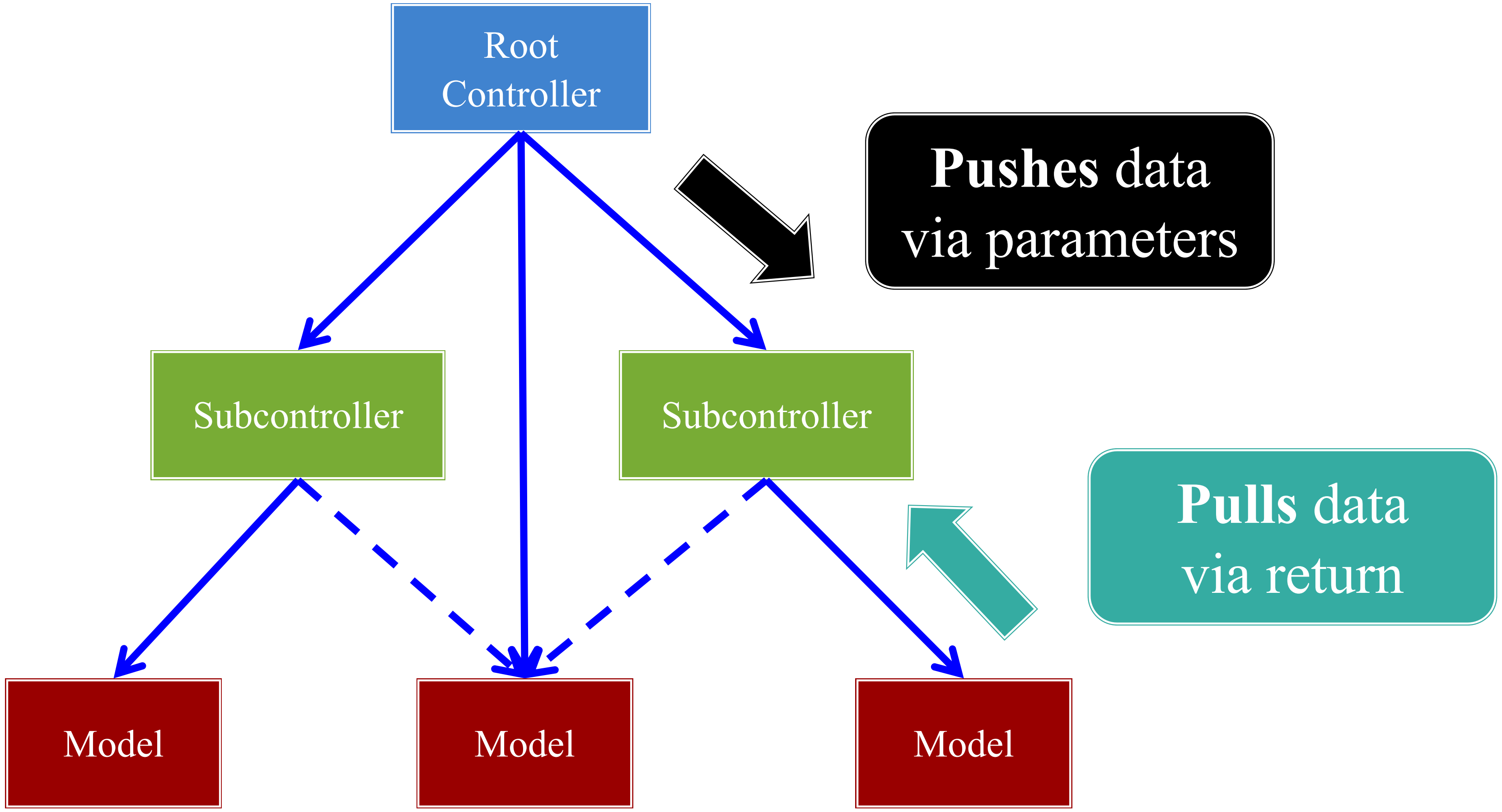
- **Collaboration**
  - Must import class/interface
  - Instantiates an object **OR**
  - Calls the objects methods
- **Ownership**
  - Instantiated the object
  - Subset of collaboration

Following the Information Flow



Pushes data via parameters

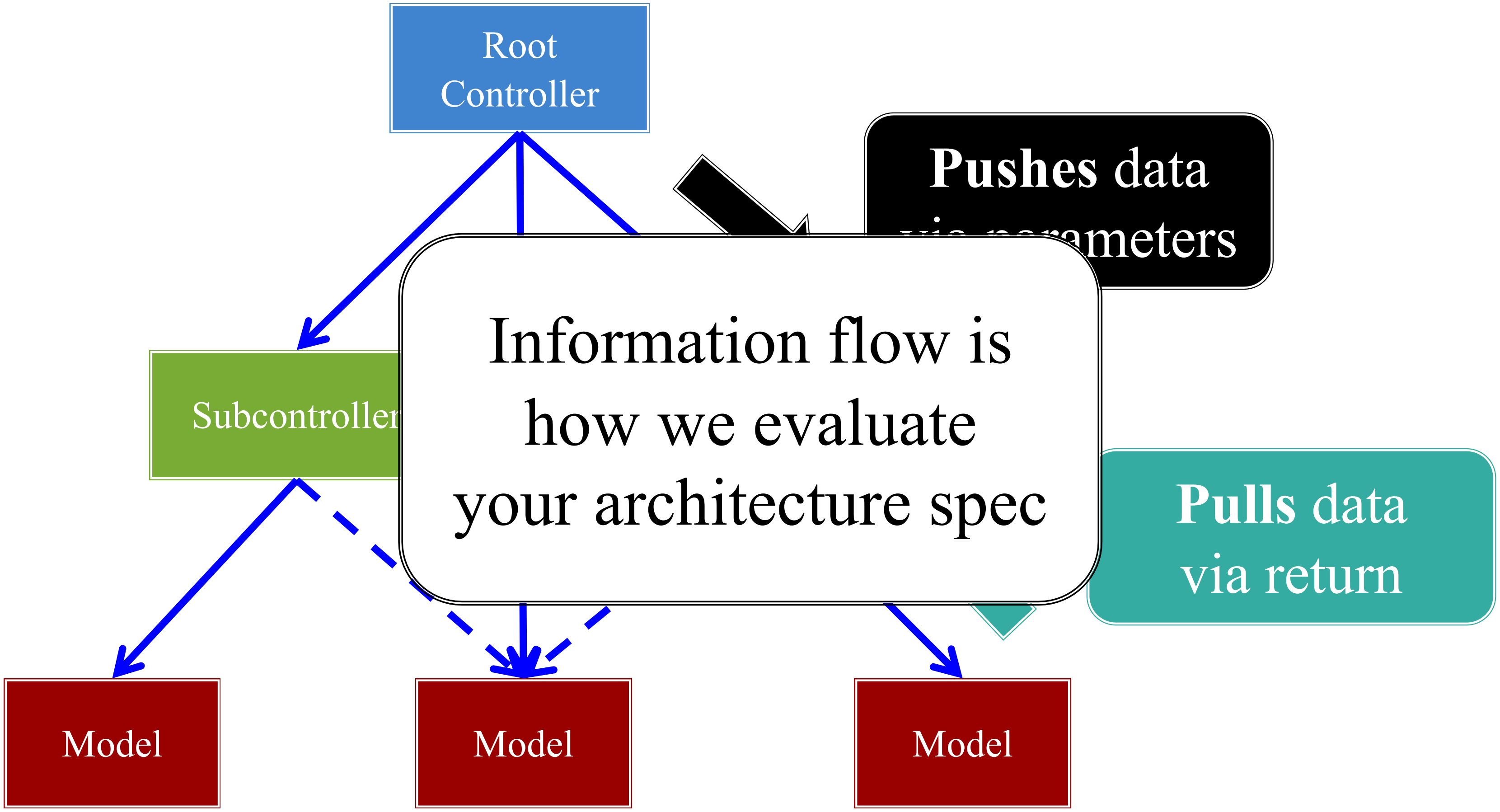
Following the Information Flow



Pushes data
via parameters

Pulls data
via return

Following the Information Flow

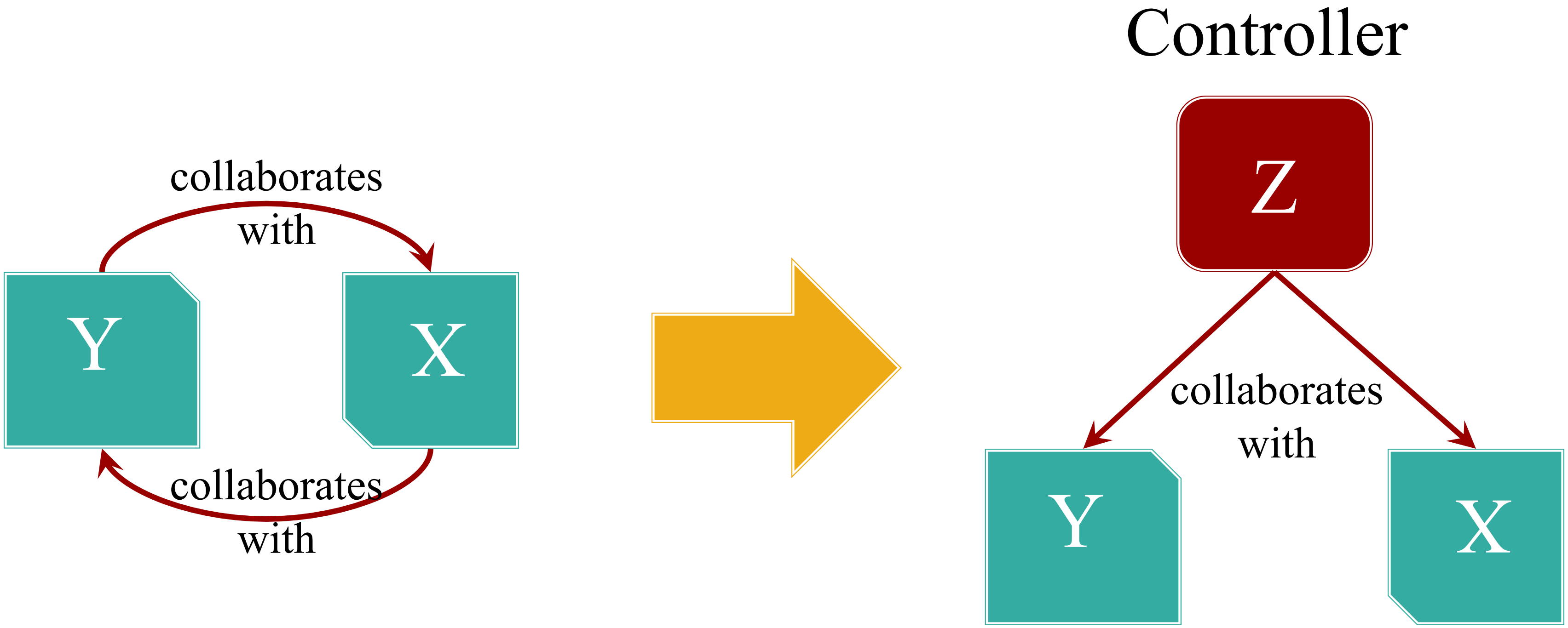


Pushes data via parameters

Information flow is how we evaluate your architecture spec

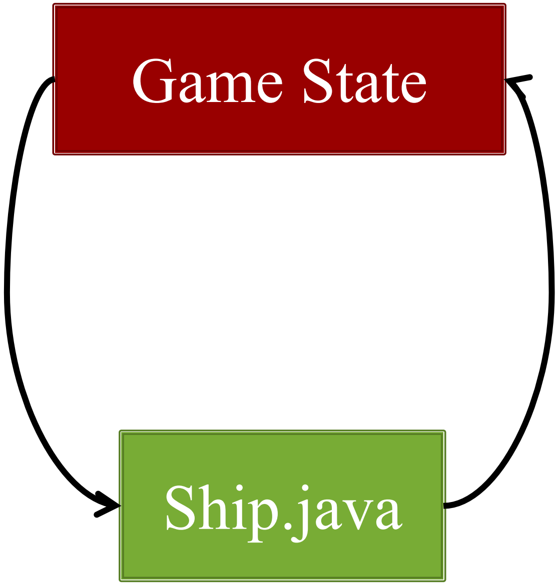
Pulls data via return

Avoid Cyclic Collaboration



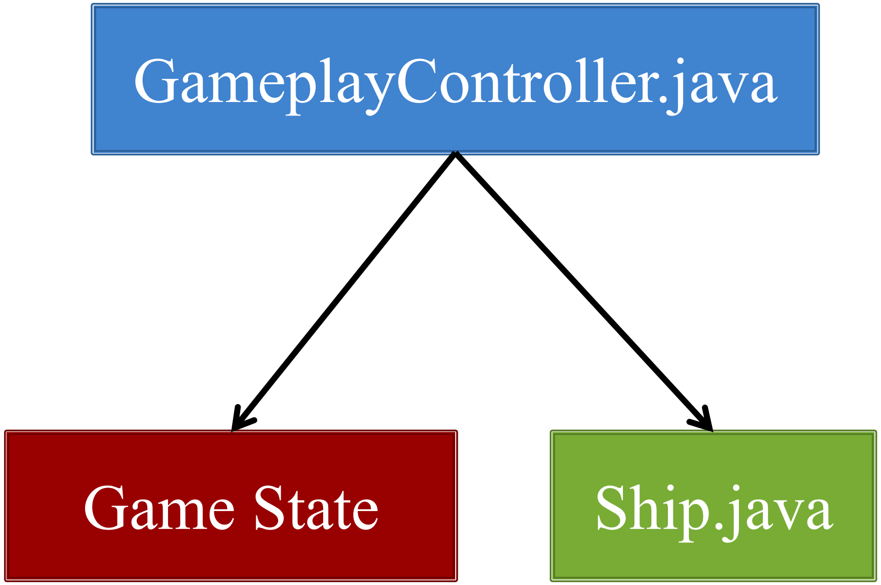
Avoid Cyclic Collaboration

- **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**



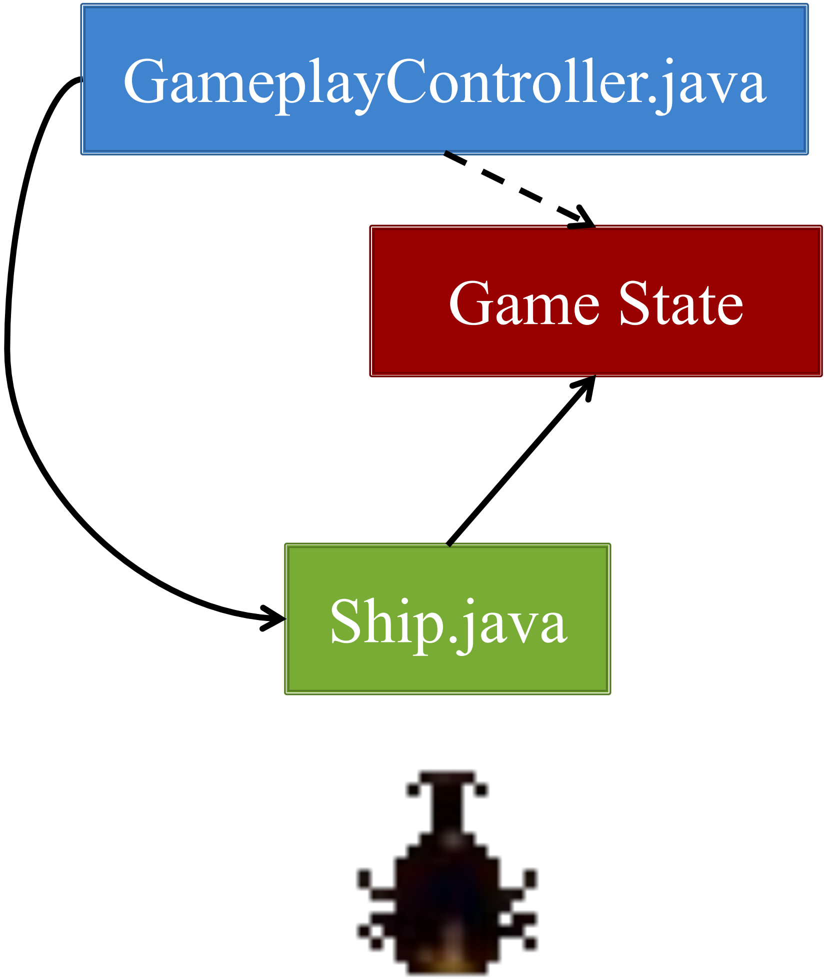
Avoid Cyclic Collaboration

- **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**

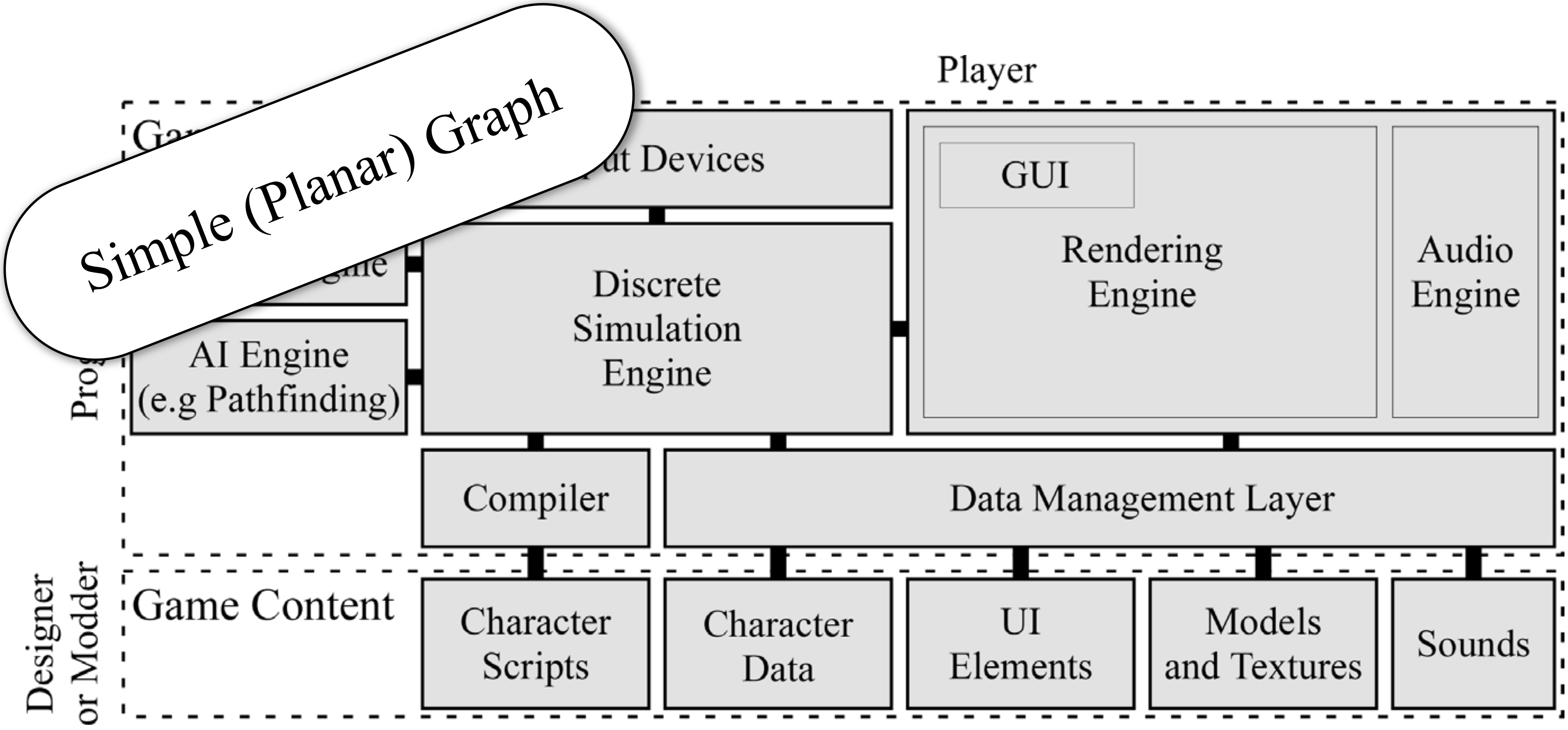


Alternative: Interfaces

- 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



CRC Index Card Exercise

Try to make collaborators adjacent

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

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

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)`

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++)

Documenting API

```
/**
 * 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

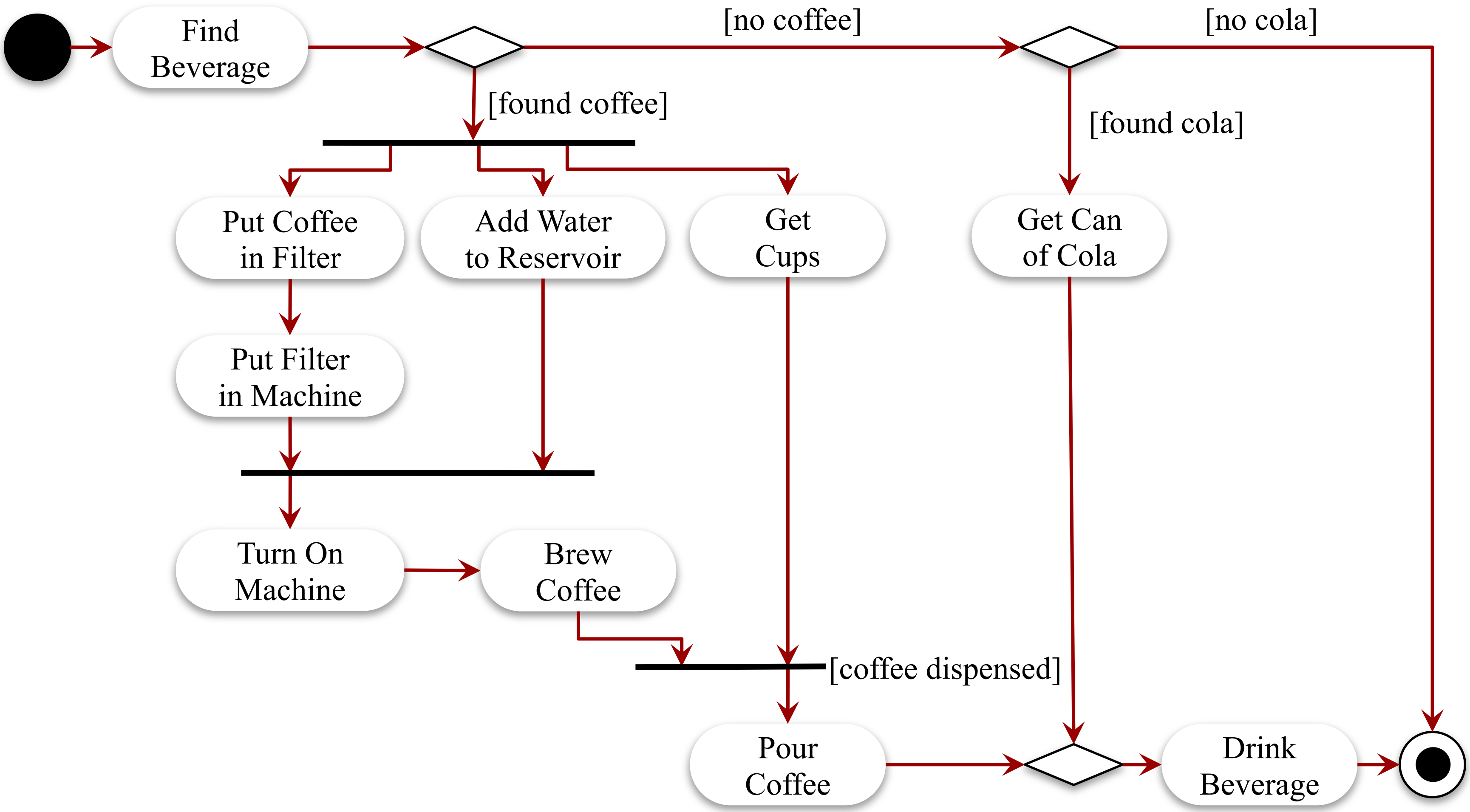
- How useful is it?
  - Extremely useful for documentation
  - Less useful for design (e.g. before implementation)
  - A language to program in another language

Activity Diagrams

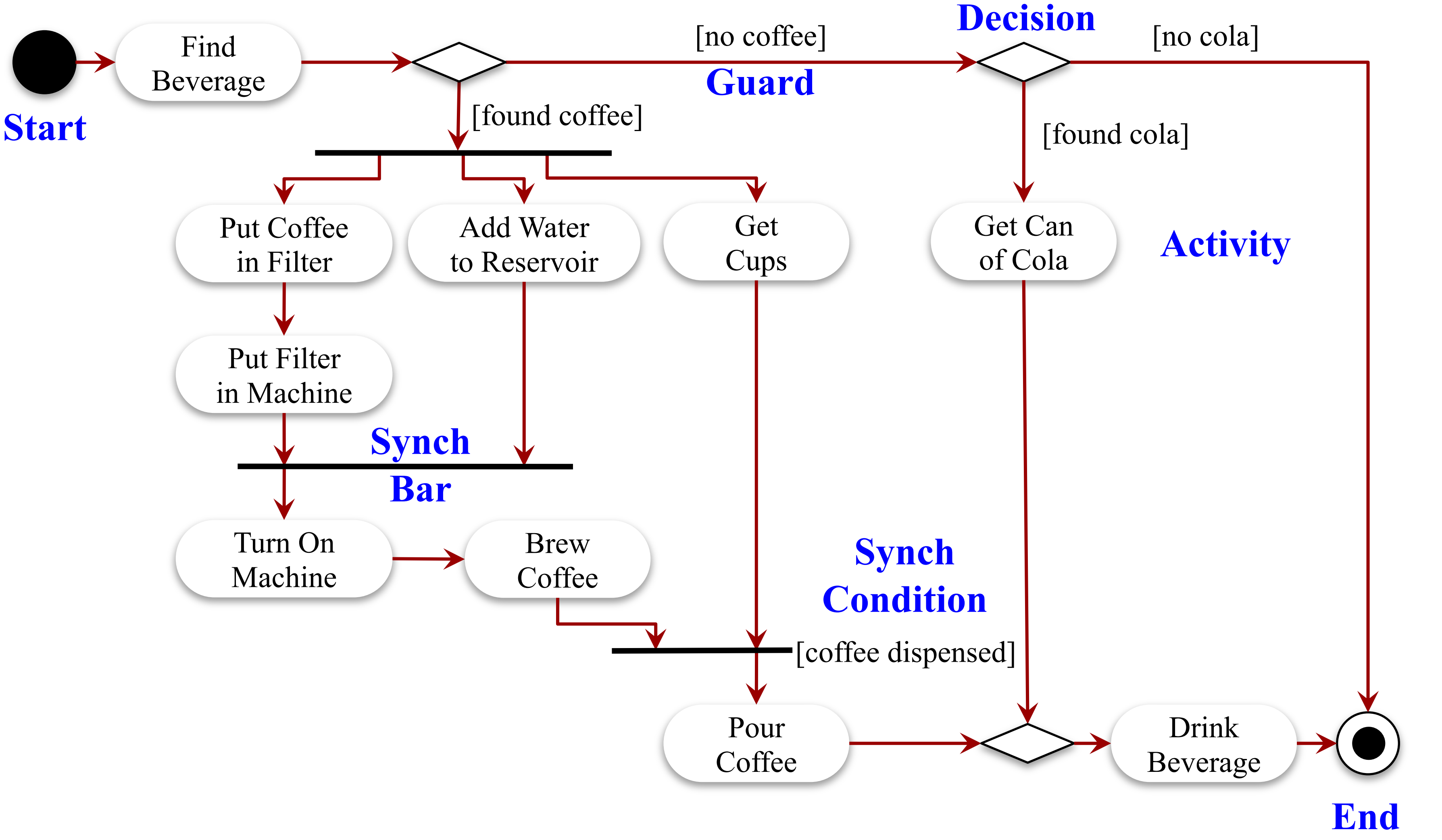
- Define the **workflow** of your program
  - Very similar to a standard flowchart
  - Can follow simultaneous paths (threads)
- Are a *component* of **UML**
  - But did not originate with UML
  - Mostly derived from **Petri Nets**
  - One of most useful UML *design* tools
- Activity diagrams are only UML we use

UNIFIED
MODELING
LANGUAGE

Activity Diagram Example

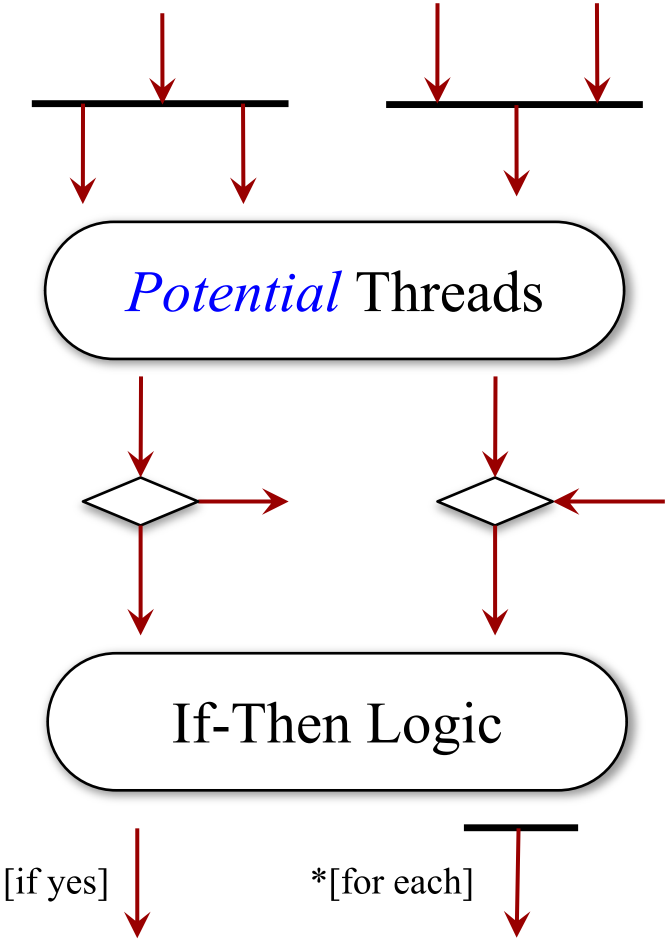


Activity Diagram Example

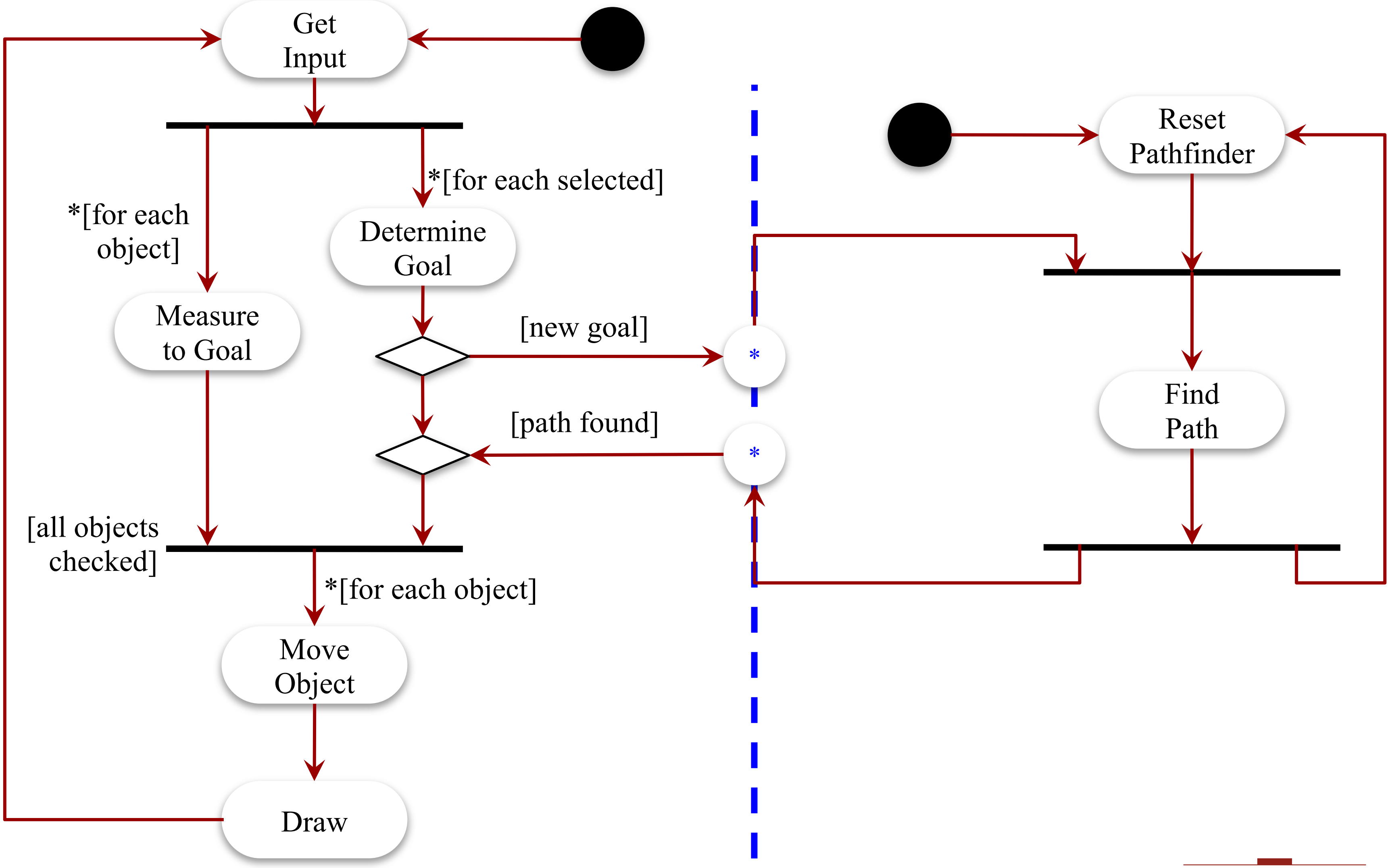


Activity Diagram Components

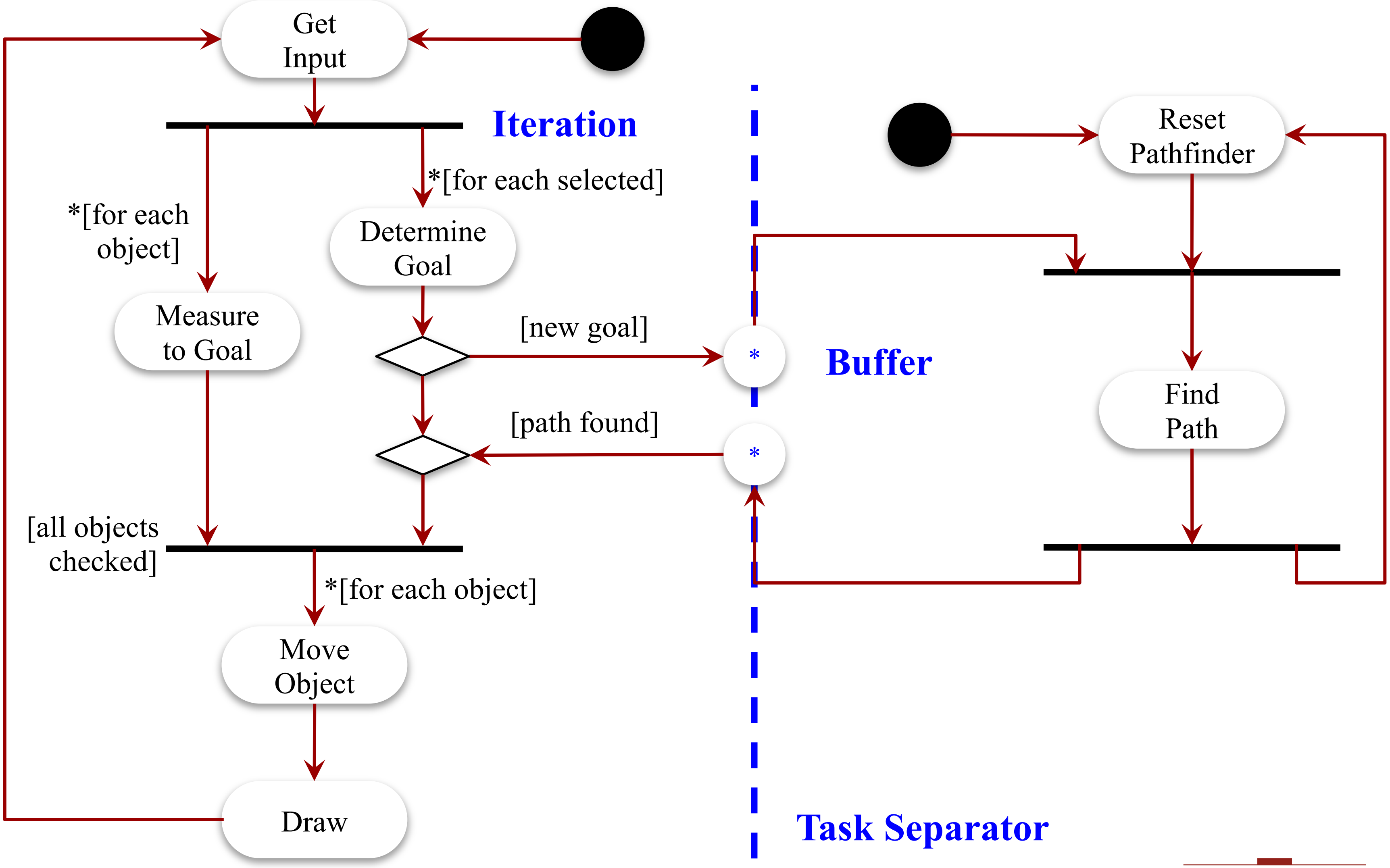
- **Synchronization Bars**
  - **In**: Wait until have happened
  - **Out**: Actions "simultaneous"
  - … or order does not matter
- **Decisions**
  - **In**: Only needs one input
  - **Out**: Only needs one output
- **Guards**
  - When we can follow edge
  - * is iteration over *container*



Asynchronous Pathfinding



Asynchronous Pathfinding



Get Input

Iteration

*[for each object]

*[for each selected]

Determine Goal

Measure to Goal

[new goal]

[path found]

[all objects checked]

*[for each object]

Move Object

Draw

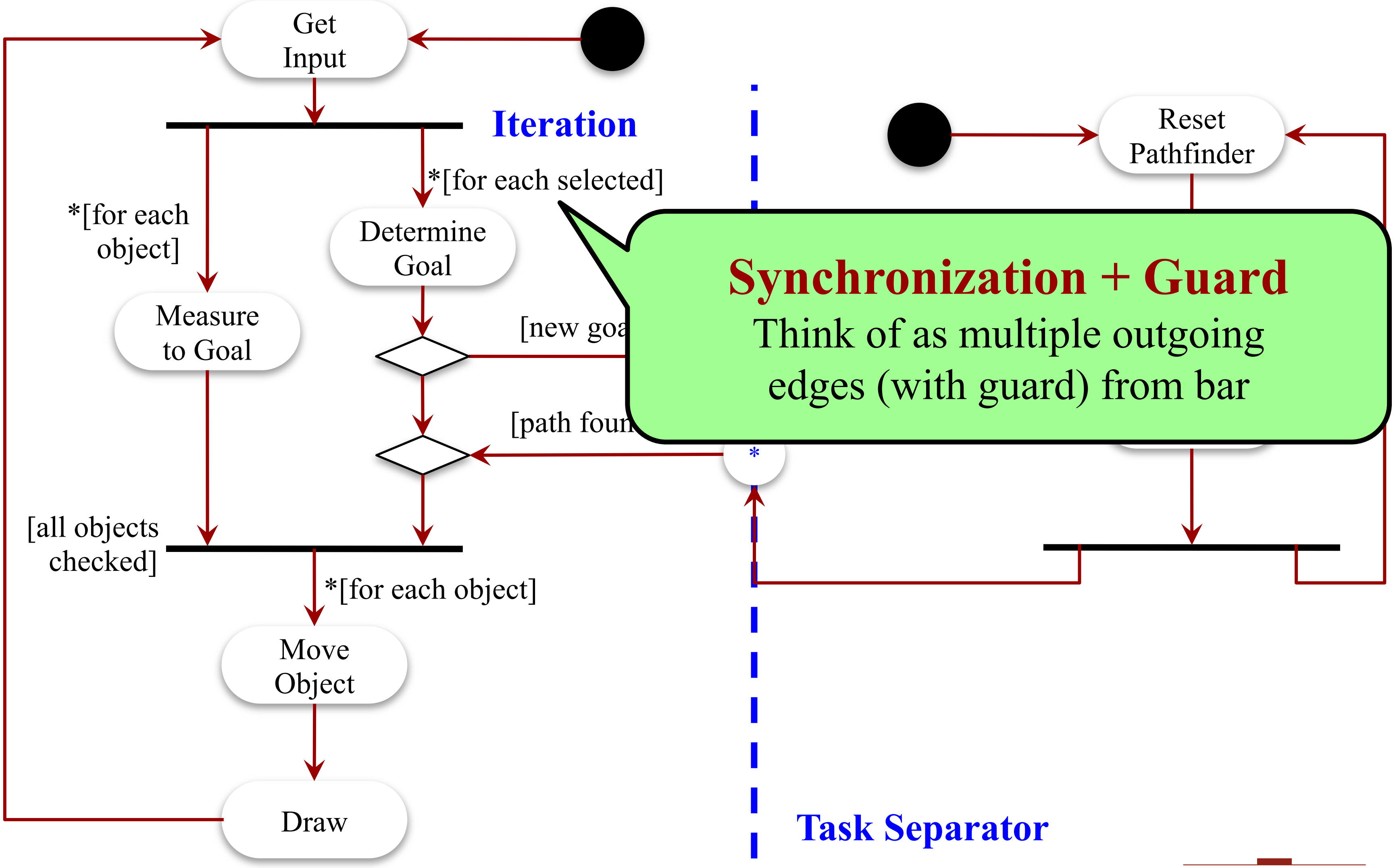
Reset Pathfinder

Buffer

Find Path

Task Separator

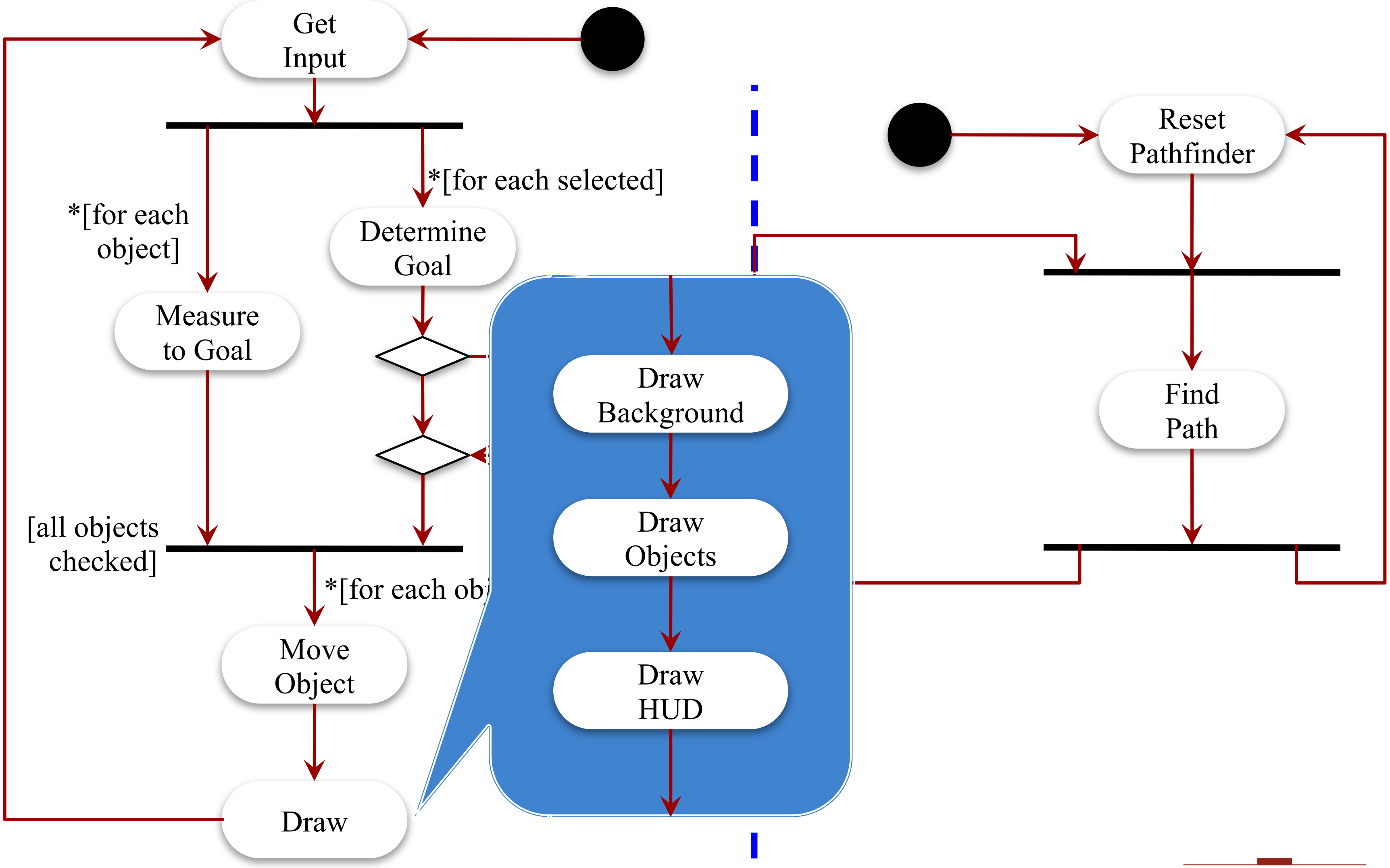
Asynchronous Pathfinding



Synchronization + Guard

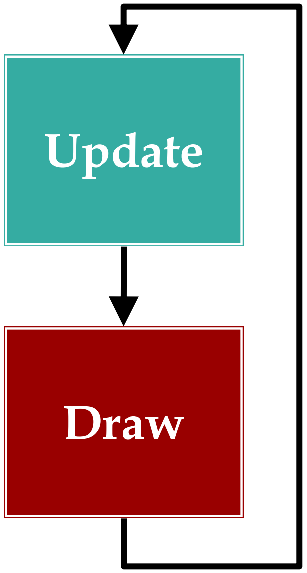
Think of as multiple outgoing edges (with guard) from bar

Expanding Level of Detail



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
  - Memory: RAM, Texture Memory
  - 2D Graphics: Drawing
  - Physics Engines: Collisions, Forces
  - Character AI: Sense-Think-Act cycle
  - Strategic AI: Asynchronous AI
- But there is more design coming too