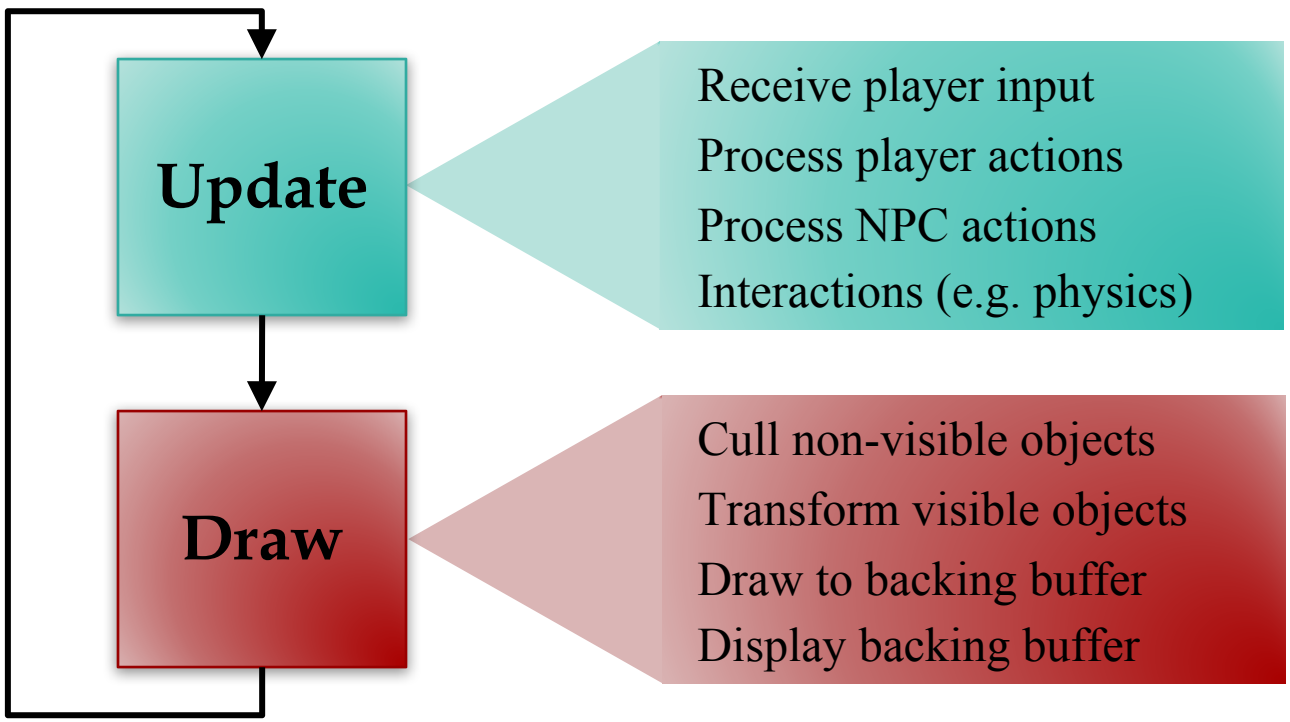
the gamedesigninitiative at cornell university

Lecture 8

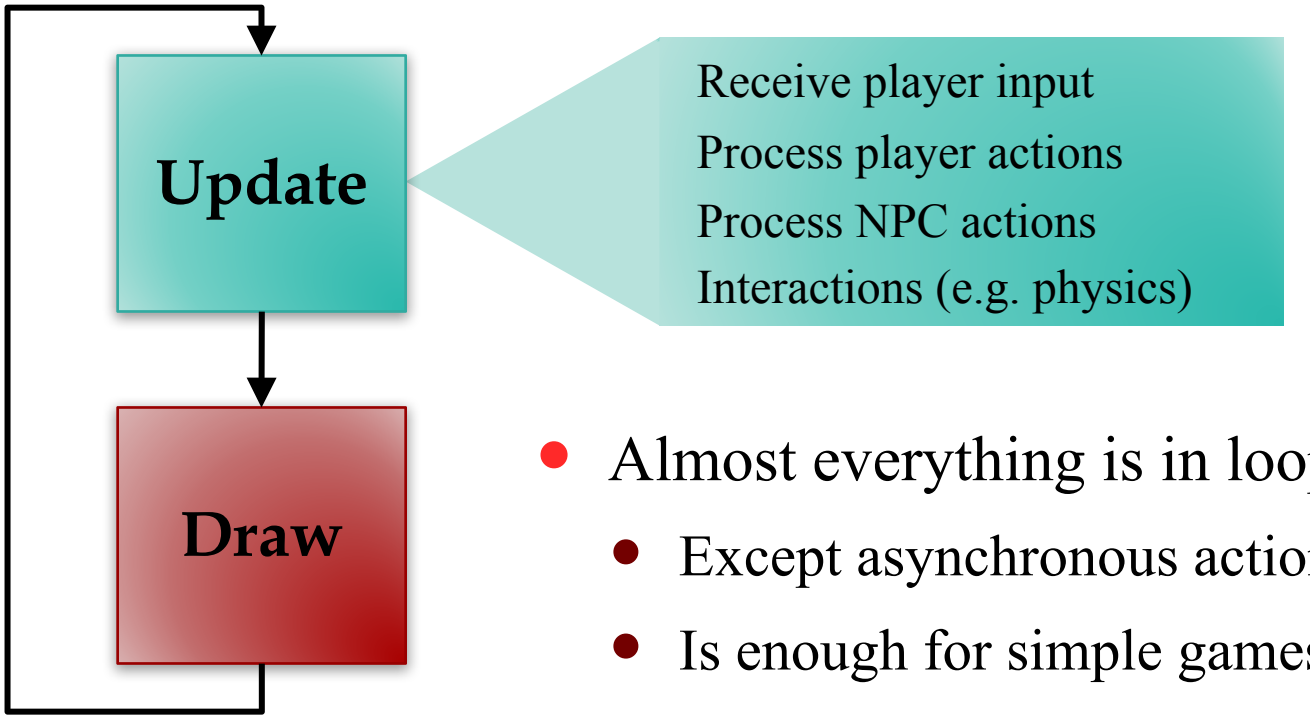
# Game Architecture Revisited

# Recall: The Game Loop

60 times/s
=
16.7 ms

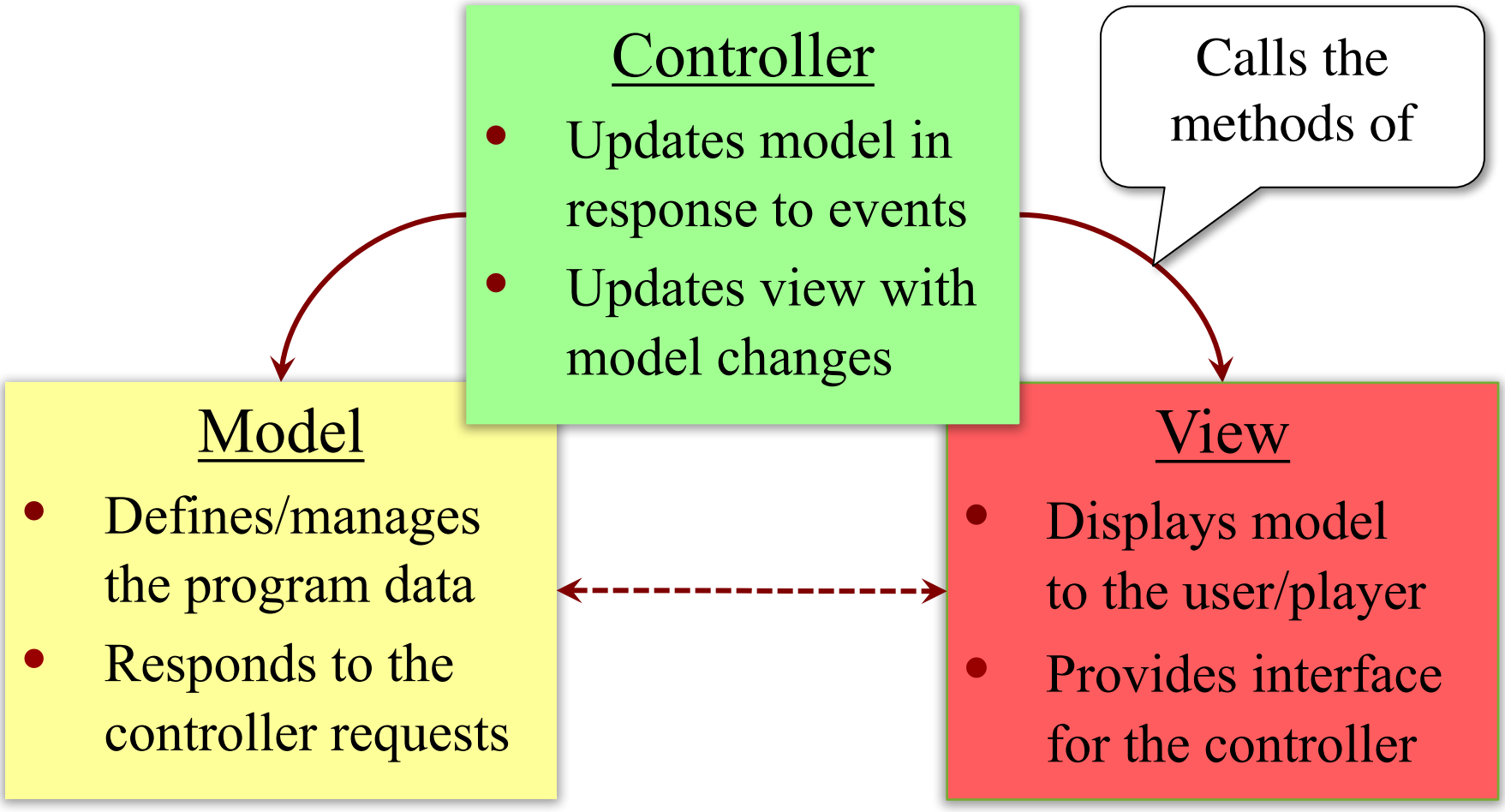


# The Game Loop



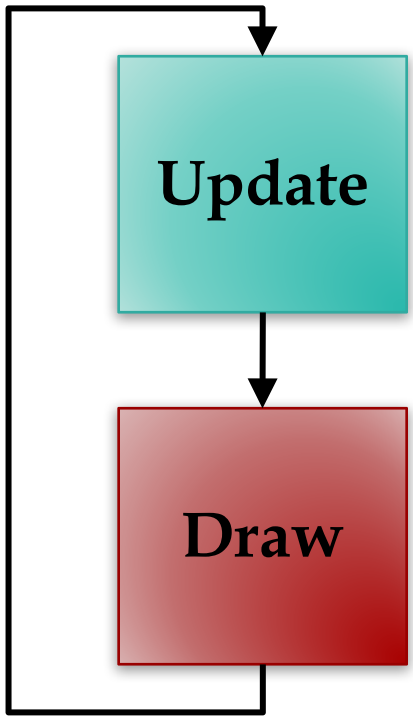

- Almost everything is in loop
  - Except asynchronous actions
  - Is enough for simple games
- How do we organize this loop?
  - Do not want spaghetti code
  - Distribute over programmers

# Model-View-Controller Pattern

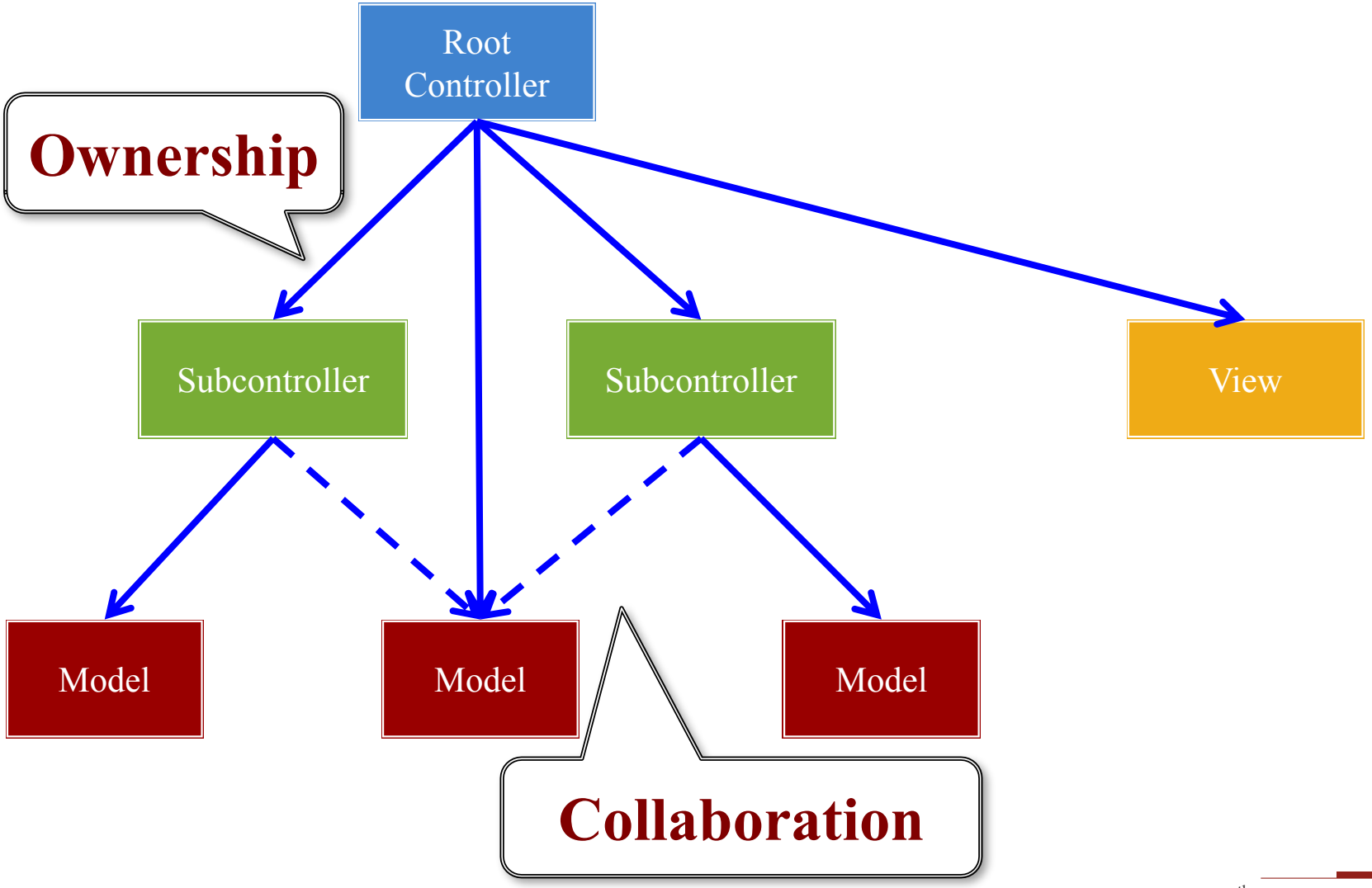


# The Game Loop and MVC

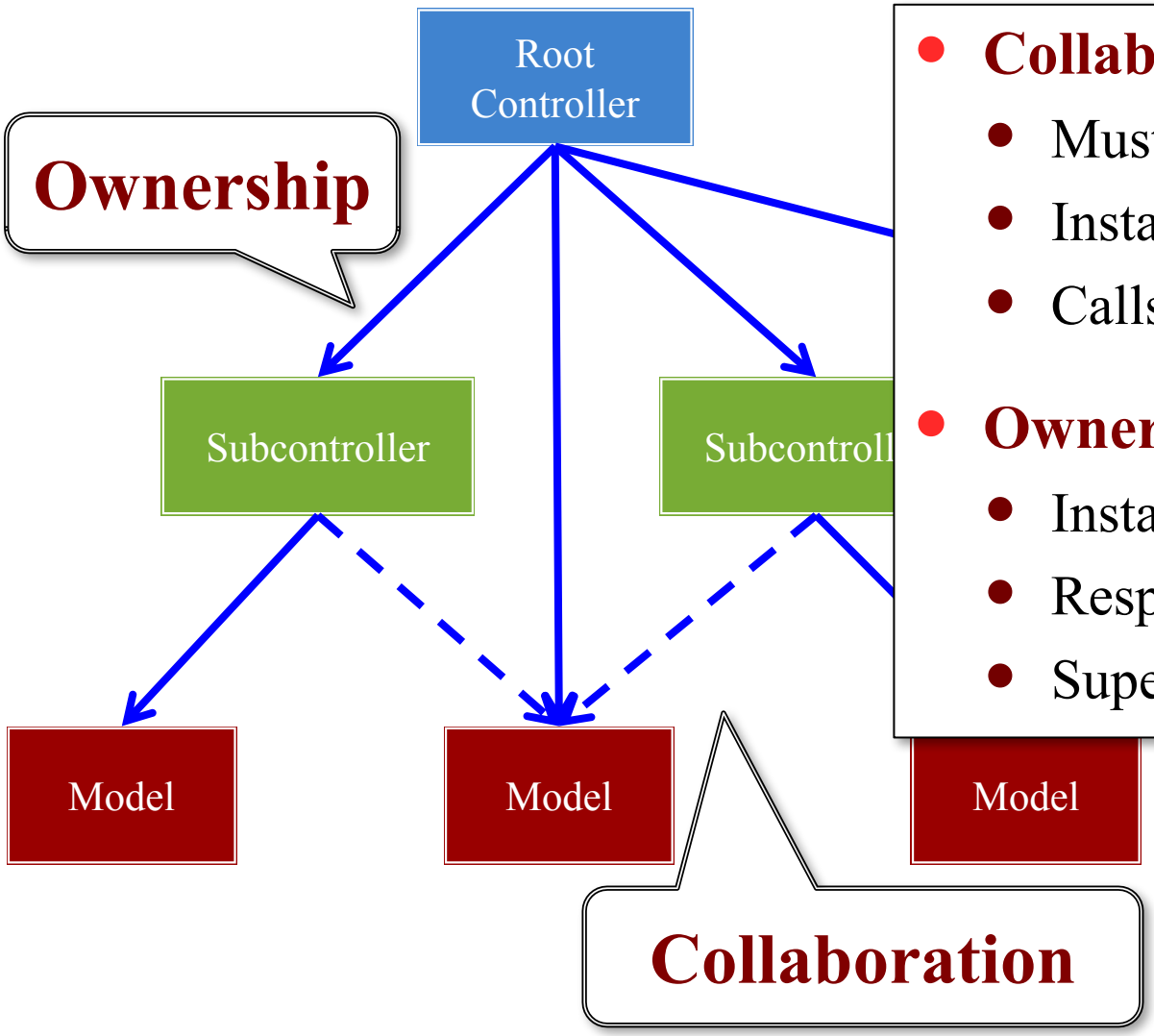
- Model: The game state
  - Value of game resources
  - Location of game objects
- View: The draw phase
  - Rendering commands only
  - Major computation in update
- Controller: The update phase
  - Alters the game state
  - Vast majority of your code



# Application Structure

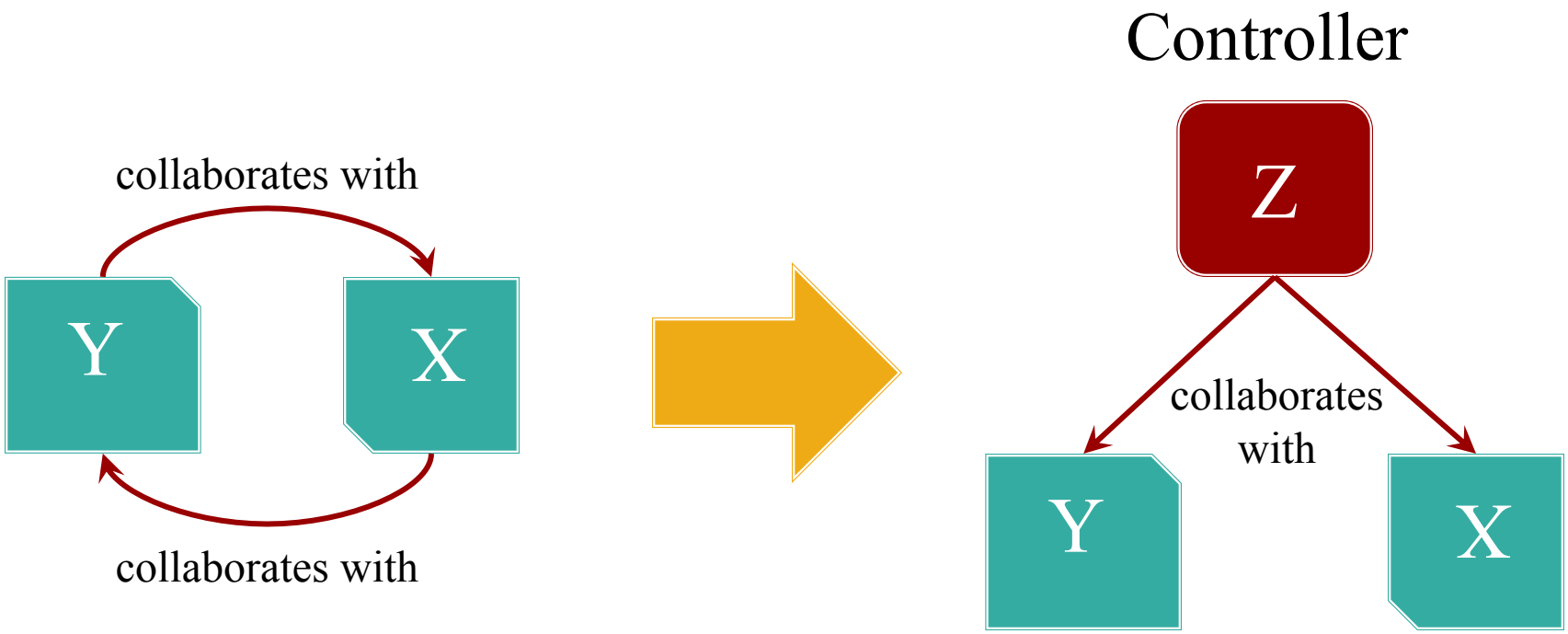


# Application Structure

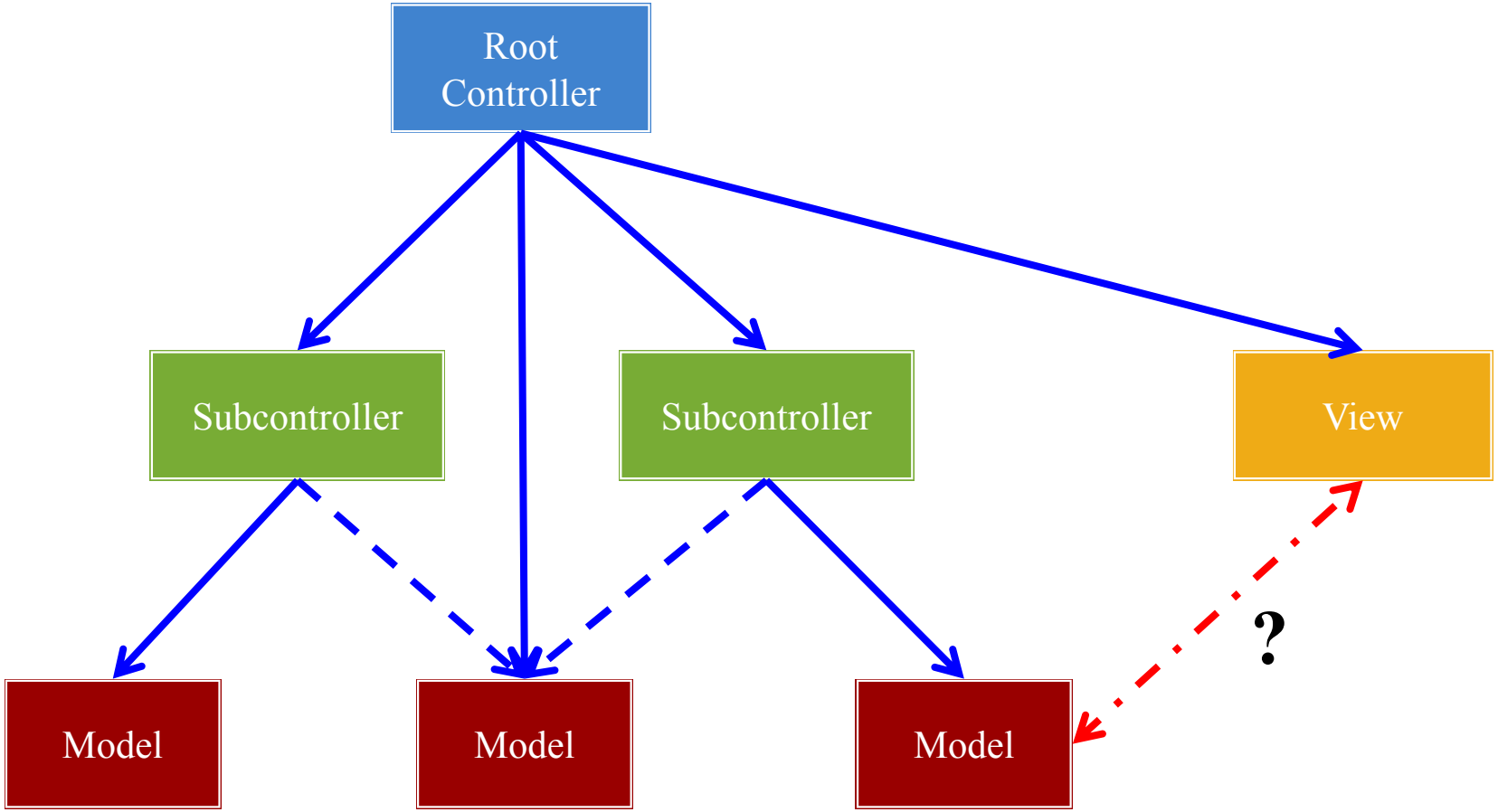



- **Collaboration**
  - Must import class/interface
  - Instantiates an object **OR**
  - Calls the objects methods
- **Ownership**
  - Instantiated the object
  - Responsible for disposal
  - Superset of collaboration

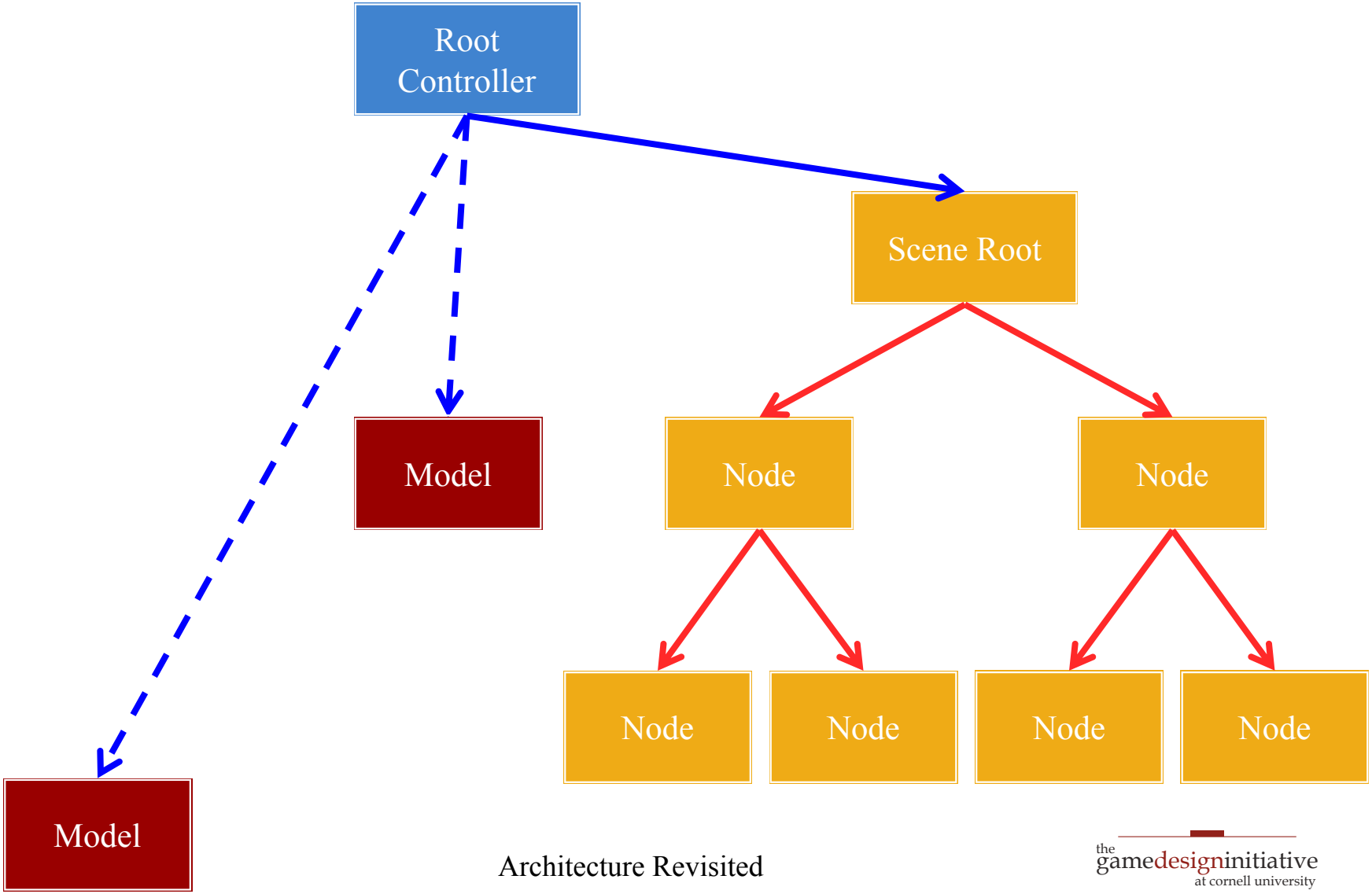
# Avoid Cyclic Collaboration



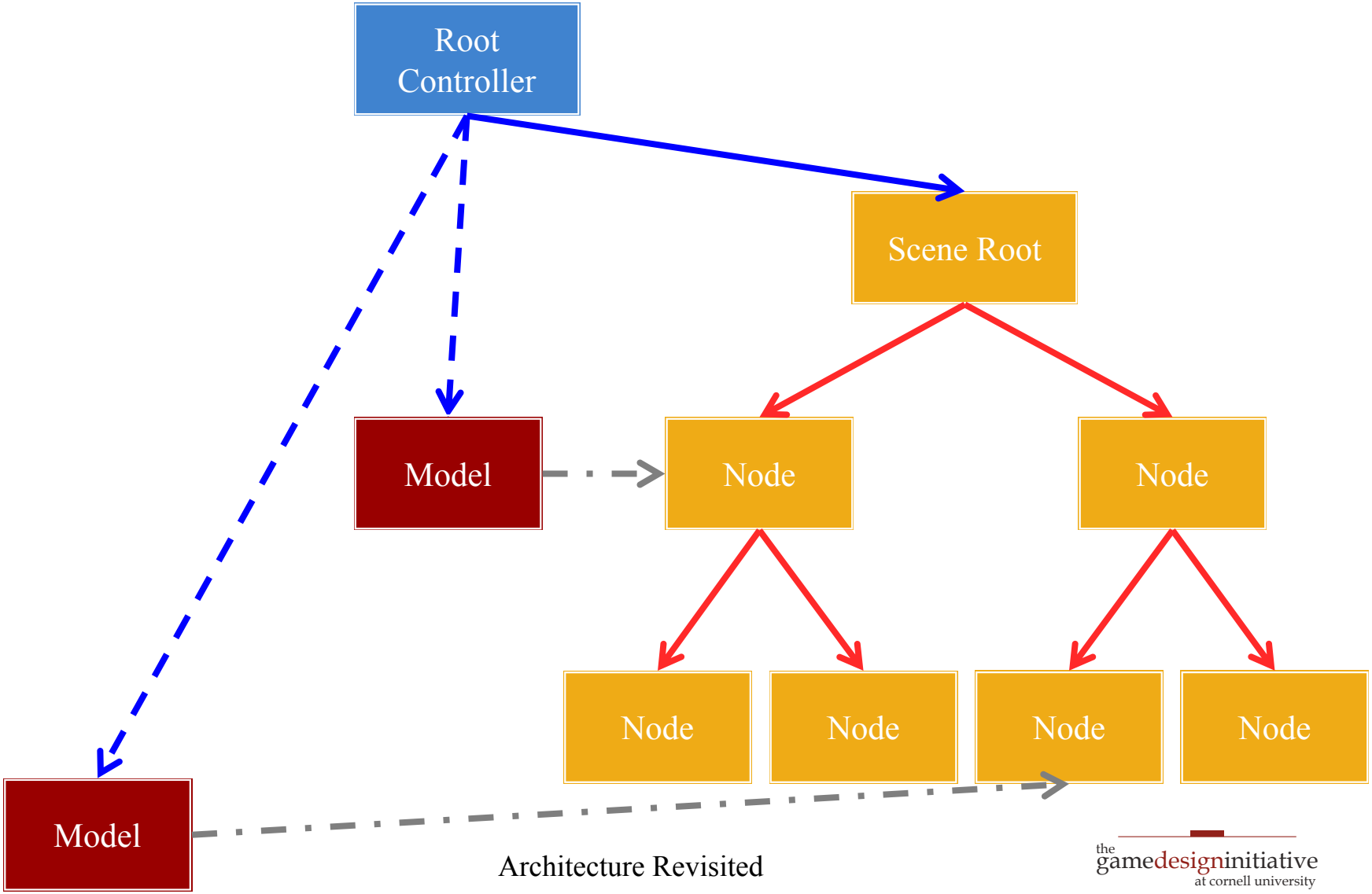
# Application Structure



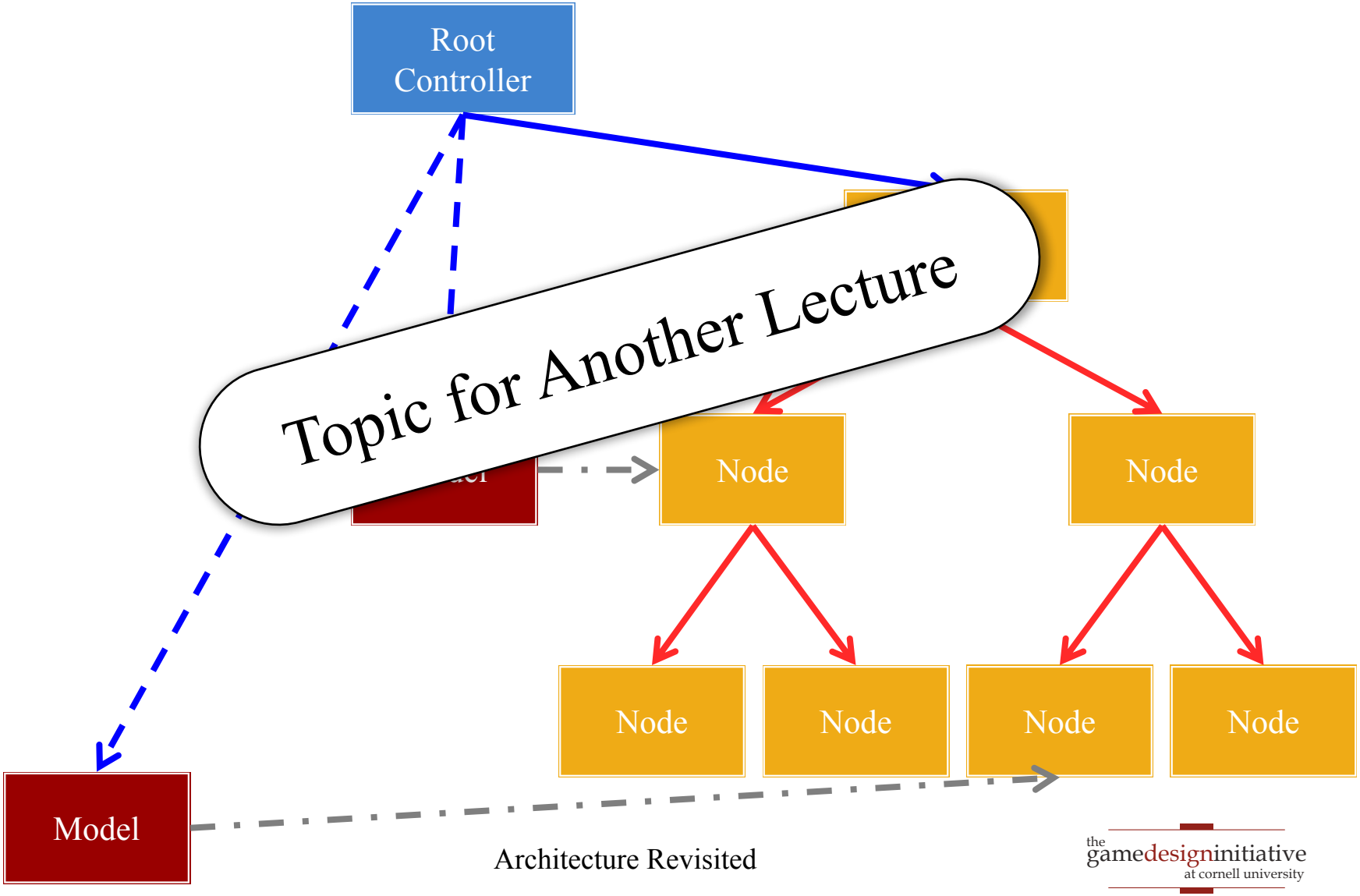
# CUGL Views: Scene Graphs



# CUGL Views: Scene Graphs



# CUGL Views: Scene Graphs



# Model-Controller Separation (Standard)

## Model

- Store/retrieve **object data**
  - Limit access (getter/setter)
  - Preserve any invariants
  - Only affects this object
- Implements **object logic**
  - Complex actions on model
  - May affect multiple models
  - **Example**: attack, collide

## Controller

- Process **user input**
  - Determine action for input
  - **Example**: mouse, gamepad
  - Call action in the model

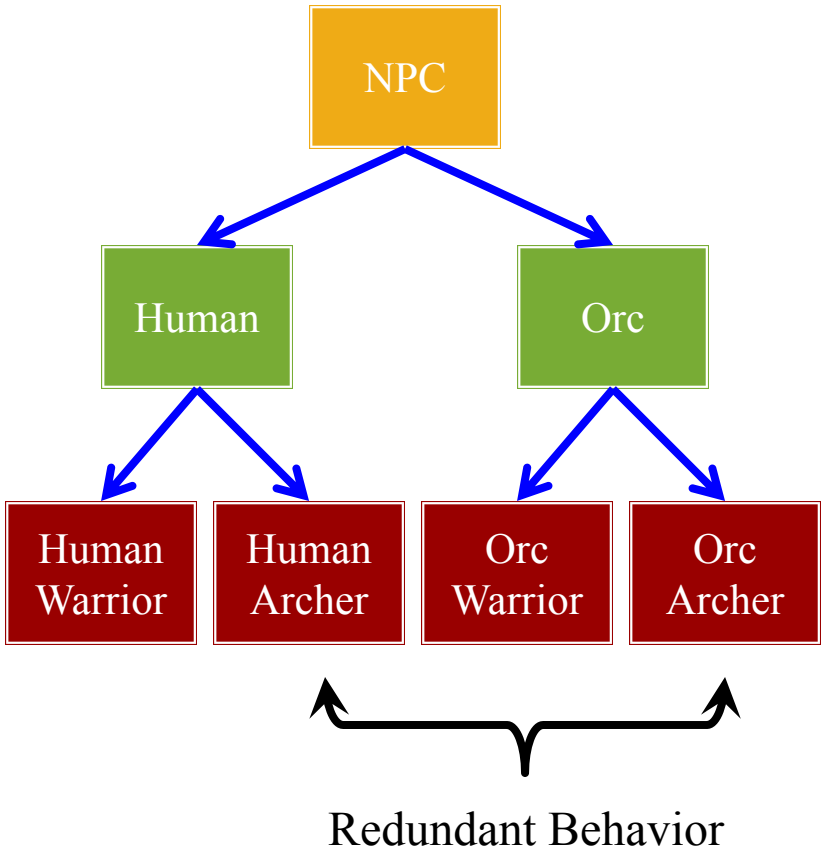
Traditional controllers are "lightweight"

# Classic Software Problem: Extensibility

- **Given**: Class with some base functionality
  - Might be provided in the language API
  - Might be provided in 3rd party software
- **Goal**: Object with *additional* functionality
  - Classic solution is to subclass original class first
  - **Example**: Extending GUI widgets (e.g. Swing)
- But subclassing does not always work…
  - How do you extend a *Singleton* object?

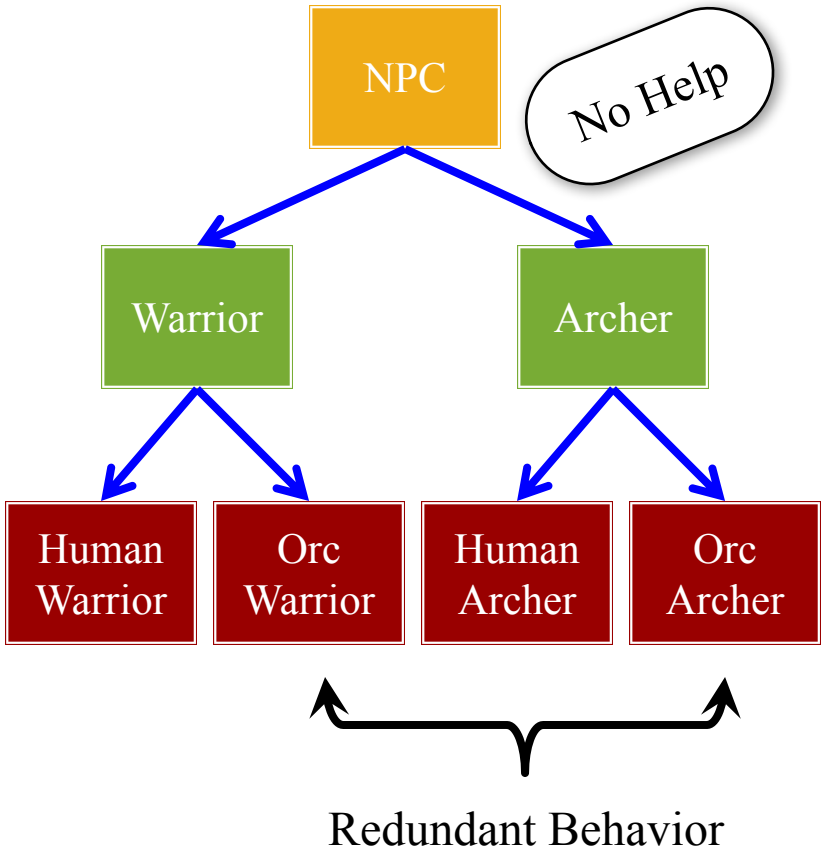
# Problem with Subclassing

- Games have *lots* of classes
  - Each game entity is different
  - Needs its own functionality (e.g. object methods)
- Want to avoid **redundancies**
  - Makes code hard to change
  - Common source of bugs
- Might be tempted to **subclass**
  - Common behavior in parents
  - Specific behavior in children



# Problem with Subclassing

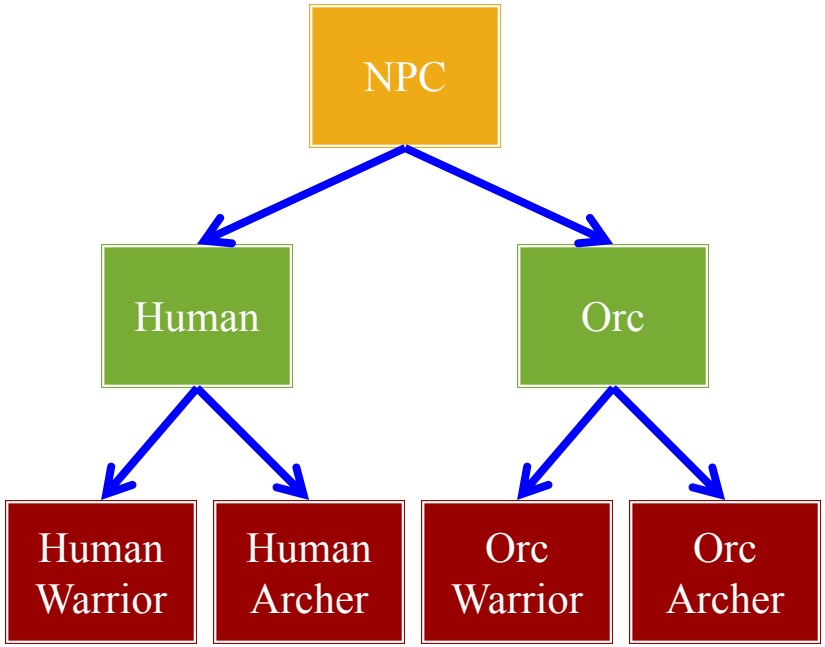
- Games have *lots* of classes
  - Each game entity is different
  - Needs its own functionality (e.g. object methods)
- Want to avoid **redundancies**
  - Makes code hard to change
  - Common source of bugs
- Might be tempted to **subclass**
  - Common behavior in parents
  - Specific behavior in children



# Model-Controller Separation (Standard)

## Model

- Store/retrieve **object data**
  - Limit access (getter/setter)
  - Preserve any invariants
  - Only affects this object
- Implements **object logic**
  - Complex actions on model
  - May affect multiple models
  - **Example**: attack, collide




Redundant Behavior

# Model-Controller Separation (Alternate)

## Model

- Store/retrieve **object data**
  - Limit access (getter/setter)
  - Preserve any invariants
  - Only affects this object

In this case, models are lightweight

## Controller

- Process **game actions**
  - Determine from input or AI
  - Find *all* objects effected
  - Apply action to objects
- Process **interactions**
  - Look at current game state
  - Look for "triggering" event
  - Apply interaction outcome

# Does Not Completely Solve Problem


- Code **correctness** a concern
  - Methods have specifications
  - Must use according to spec
- Check correctness via **typing**
  - Find methods in object class
  - **Example**: `orc.flee()`
  - Check type of parameters
  - **Example**: `force_to_flee(orc)`
- **Logical** association with type
  - Even if not part of class

# Issues with the OO Paradigm

- Object-oriented programming is very **noun-centric**
  - All code must be organized into classes
  - Polymorphism determines capability via type
- OO became popular with **traditional MVC pattern**
  - Widget libraries are nouns implementing view
  - Data structures (e.g. CS 2110) are all nouns
  - Controllers are not necessarily nouns, but lightweight
- Games, interactive media break this paradigm
  - View is animation (process) oriented, not widget oriented
  - Actions/capabilities only loosely connected to entities

# Programming and Parts of Speech

## Classes/Types are Nouns

- Methods have verb names
- Method calls are sentences
  - `subject.verb(object)`
  - `subject.verb()`
- Classes related by *is-a*
  - Indicates class a subclass of
  - **Example**: String `is-a` Object
- Objects are class *instances*

## Actions are Verbs

- Capability of a game object
- Often just a simple function
  - `damage(object)`
  - `collide(object1,object1)`
- Relates to objects via *can-it*
  - **Example**: Orc `can-it` attack
  - Not necessarily tied to class
  - **Example**: swapping items

# Duck Typing: Reaction to This Issue

- "Type" determined by its
  - Names of its methods
  - Names of its properties
  - If it "quacks like a duck"
- Python has this capability
  - hasattr(<object>,<string>)
  - True if object has attribute or method of that name
- This has many **problems**
  - Correctness is a *nightmare*

**Java:**

```
public boolean equals(Object h) {
    if (!(h instanceof Person)) {
        return false;}
    Person ob= (Person)h;
    return name.equals(ob.name);
}
```

**Python:**

```
def __eq__(self,ob):
    if (not (hasattr(ob,'name'))
        return False
    return (self.name == ob.name)
```

# Duck Typing: Reaction to This Issue

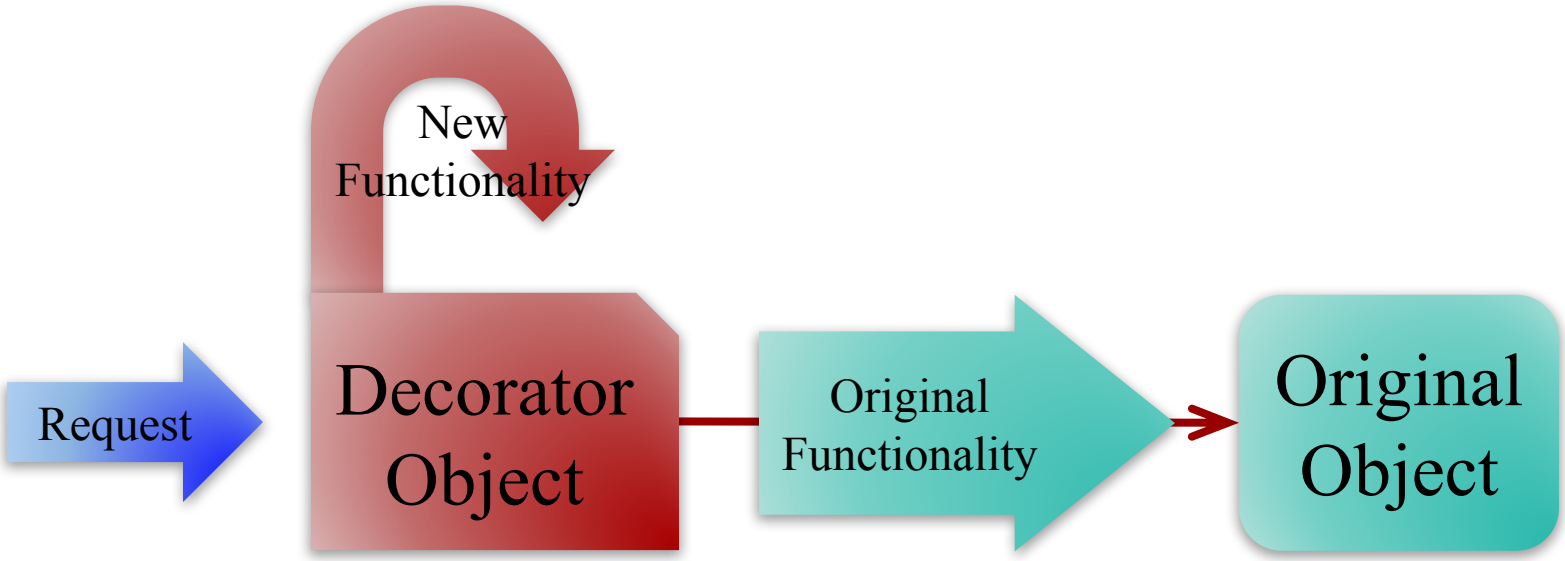
- "Type" determined by its
  - Names of its methods
  - Names
  - If it "qu
- Python ha
  - hasattr
  - True if
    or meth
- This has many **problems**
  - Correctness is a *nightmare*

**Java:**

```
public boolean equals(Object h) {
                                  rson)) {
                                  )h;
                                  (ob.name);
                                  name'))
    return False
return (self.name == ob.name)
```

- What do we really want?
  - Capabilities over properties
  - Extend capabilities without necessarily changing type
  - Without using new languages
- Again, use *software patterns*

# Possible Solution: Decorator Pattern



# Java I/O Example

```
InputStream input = System.in;
```

Built-in console input

```
Reader reader = new InputStreamReader(input);
```

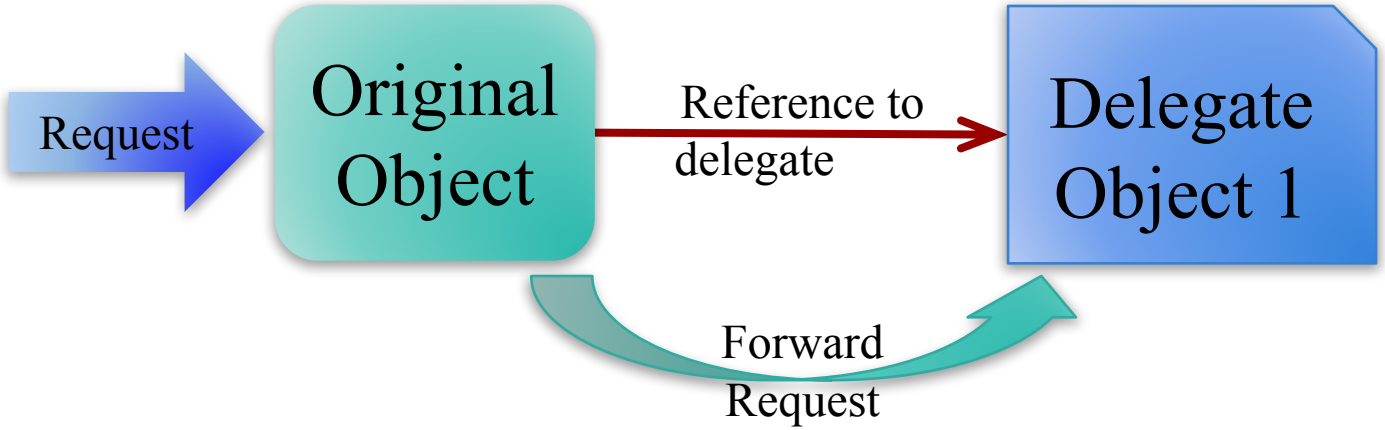
Make characters easy to read

```
BufferedReader buffer = new BufferedReader(reader);
```

Read whole line at a time

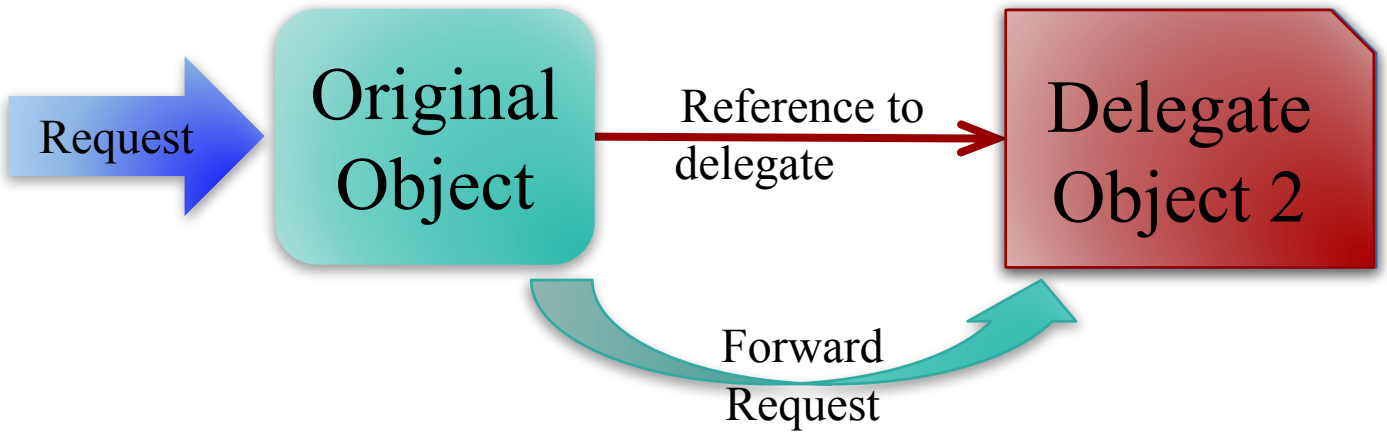
Most of java.io works this way

# Alternate Solution: Delegation Pattern




*Inversion* of the Decorator Pattern

# Alternate Solution: Delegation Pattern




*Inversion* of the Decorator Pattern

# Example: Sort Algorithms

```
public class SortableArray extends ArrayList{
  private Sorter sorter = new MergeSorter();   new QuickSorter();
  public void setSorter(Sorter s) { sorter = s; }
  public void sort() {
      Object[] list = toArray();
      sorter.sort(list);
      clear();
      for (o:list) { add(o); }
   }
}
```

```
public interface Sorter {
   public void sort(Object[] list);
}
```

# Comparison of Approaches

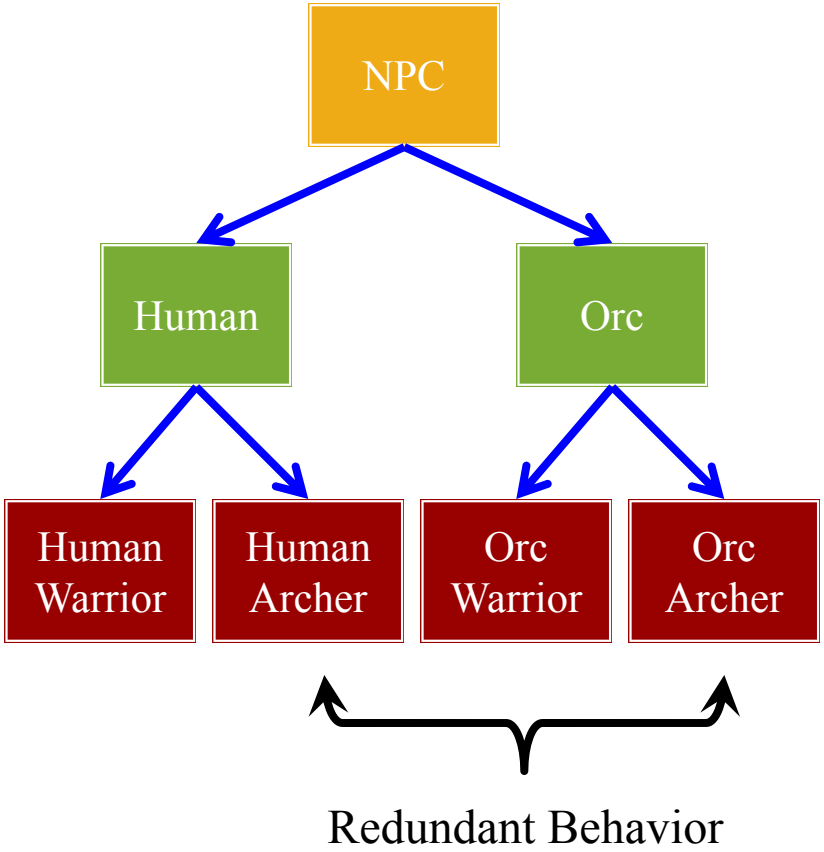
## Decoration

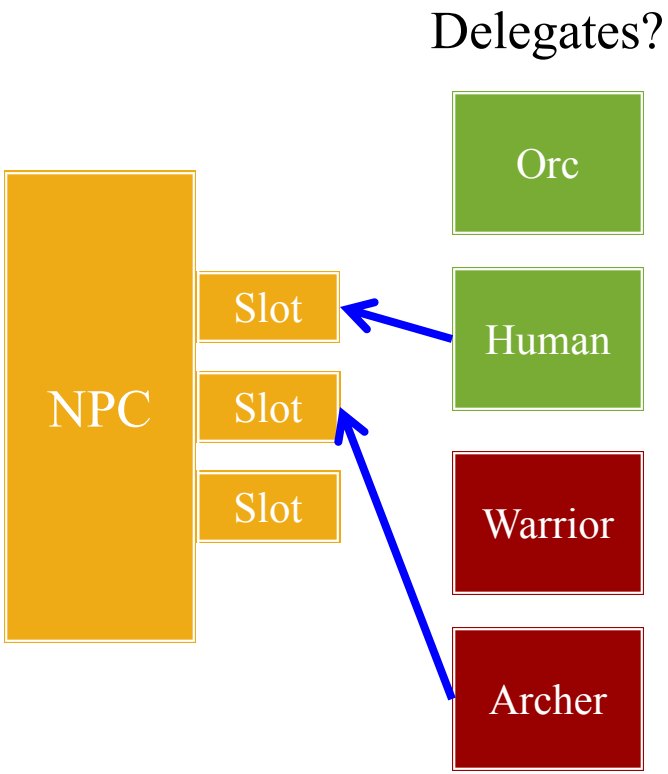
- Pattern applies to *decorator*
  - Given the original object
  - Requests through decorator
- **Monolithic** solution
  - Decorator has all methods
  - "Layer" for more methods (e.g. Java I/O classes)
- Works on *any* object/class

## Delegation

- Applies to *original object*
  - You designed object class
  - All requests through object
- **Modular** solution
  - Each method can have own delegate implementation
  - Like higher-order functions
- Limited to classes you make

# The Subclass Problem Revisited



# A C++ Solution: Mix-Ins

- Orthogonal class design
  - Start with common base
  - Only one level of superclass
  - Other classes can combine
- Needs C++ templates
  - Templatize base class
  - Nest templates to mix
- Builds an inheritance tree
  - But tree details not needed
  - Stacking is commutative

```
class Number {
   int n;
   void set(int v) { n = v; }
   int get() const { return n; }
};

template <typename BASE>
class Undo : public BASE {
   int pre;
   void set(int v) { pre=BASE::get(); BASE::set(v); }
   void undo()     { BASE::set(pre); }
};

template <typename BASE>
class Redo : public BASE {
   int post;
   void set(int v) { post= v; BASE::set(v); }
   void redo()     { BASE::set(post); }
};

typedef Redo< Undo<Number> > ReUndoNumber;
```

# Summary

- Games naturally fit a **specialized MVC** pattern
  - Want *lightweight* models (mainly for serialization)
  - Want *heavyweight* controllers for the game loop
  - View is specialized rendering with few widgets
- Proper design leads to unusual OO patterns
  - Subclass hierarchies are unmanageable
  - **Component-based design** better models actions
  - More advanced patterns supported by C++.