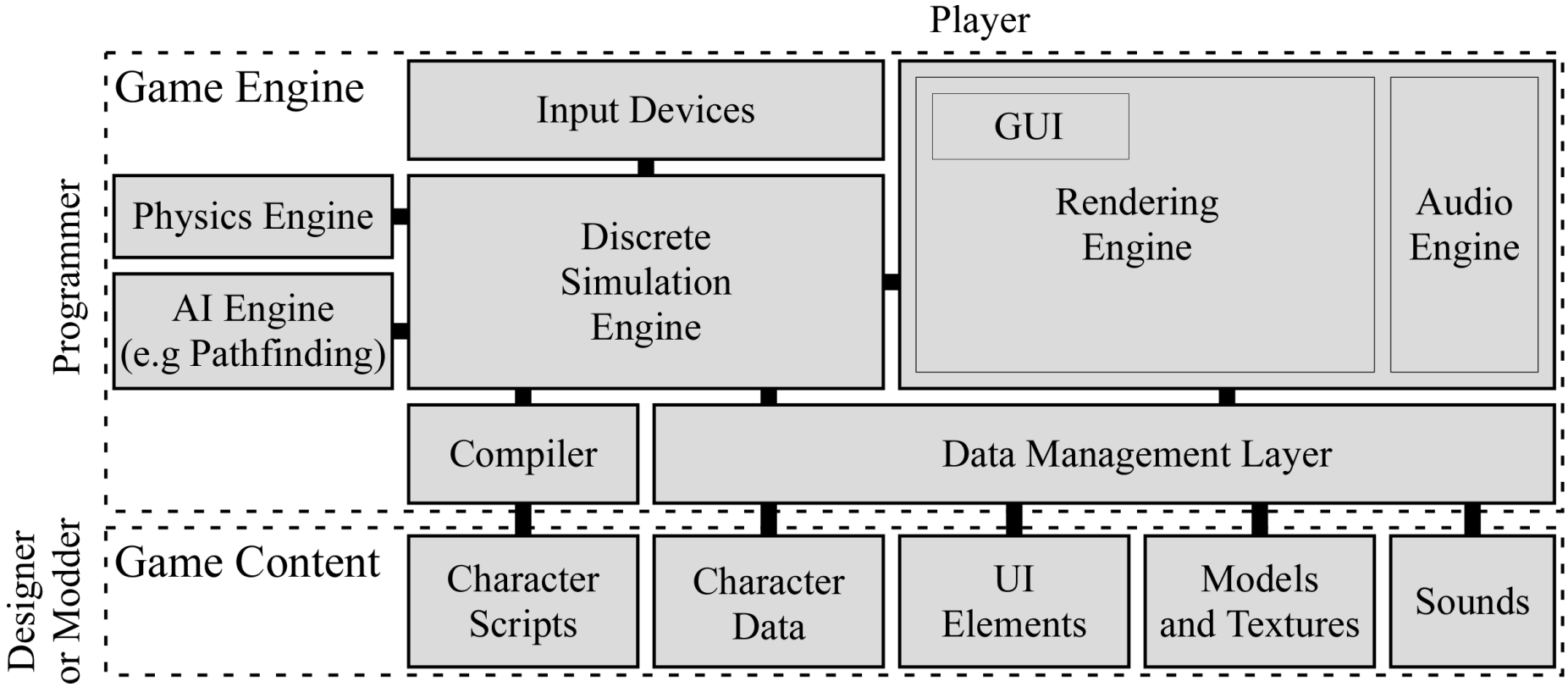
the gamedesigninitiative at cornell university

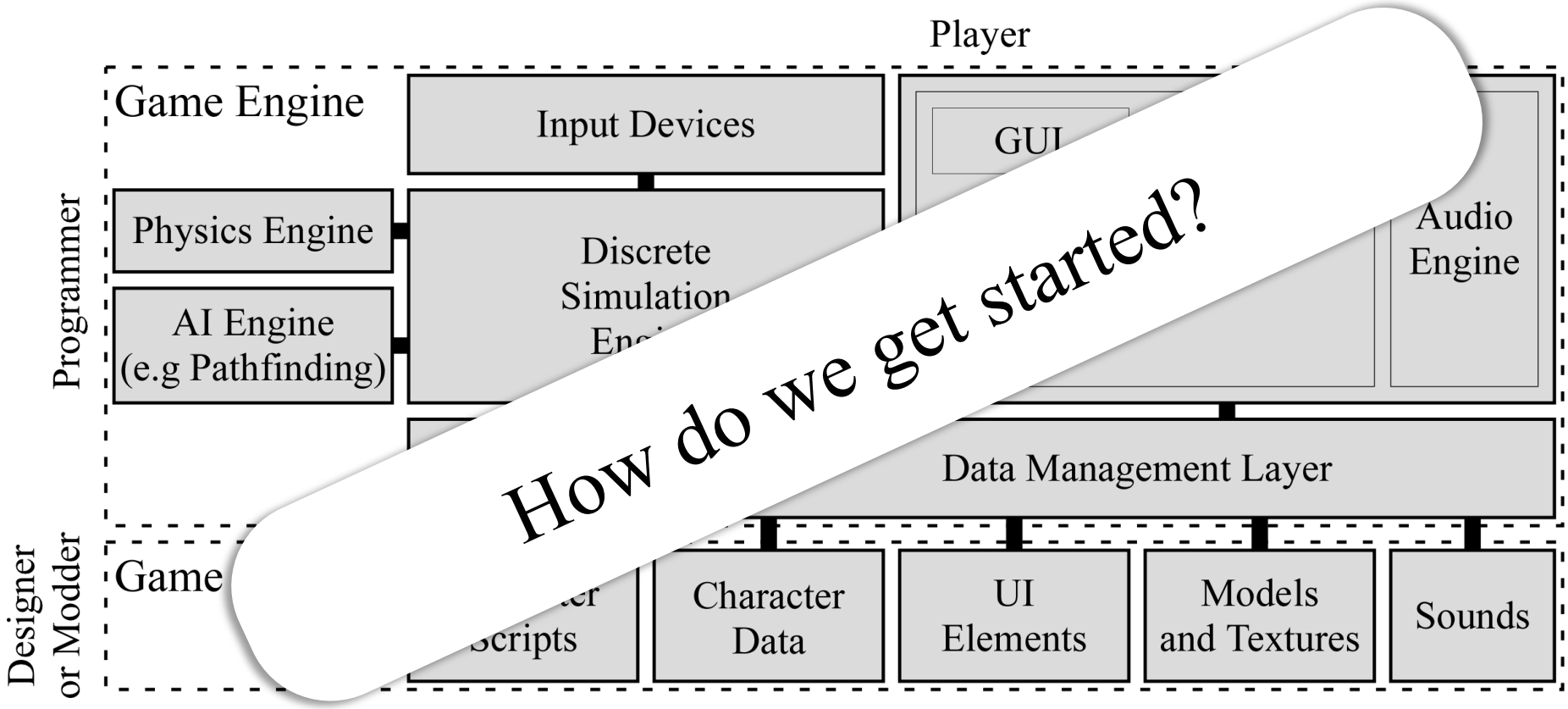
Lecture 11

Architecture Patterns

Architecture: The Big Picture



Architecture: The Big Picture



Utilizing Software Patterns

- **Pattern**: reusable solution to a problem
  - Typically a template, not a code library
  - Tells you how to design your code
  - Made by someone who ran into problem first
- In many cases, pattern gives you the interface
  - List of headers for non-hidden methods
  - Specification for non-hidden methods
  - Only thing missing is the implementation

2110 all over again

the gamedesigninitiative at cornell university

Example: Singletons

- **Goal**: Want to limit class to a single instance
  - Do not want to allow users to construct new objects
  - But do want them to access the single object
- **Application**: Writing to the console/terminal
  - Want a unique output stream to write to console
  - Many output streams would conflict w/ each other
  - Given by a unique object in Java (System.out)
  - A class with static methods in C# (not a singleton)

the gamedesigninitiative at cornell university

Creating a Singleton in Java

```
public class Singleton {
    public static final Singleton instance = new Singleton();

    private Singleton() {
        // Initialize all fields for instance
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```


the gamedesigninitiative at cornell university

Creating a Singleton in Java

```
public class Singleton {

    public static final Singleton instance = new Singleton();

    private Singleton() {

        // I                                    instance
    }

    public static Singleton getInstance() {

        return instance;
    }
}
```

Provide as an immutable constant

Keep user from instantiating

Static method is an alternative to providing access with a constant

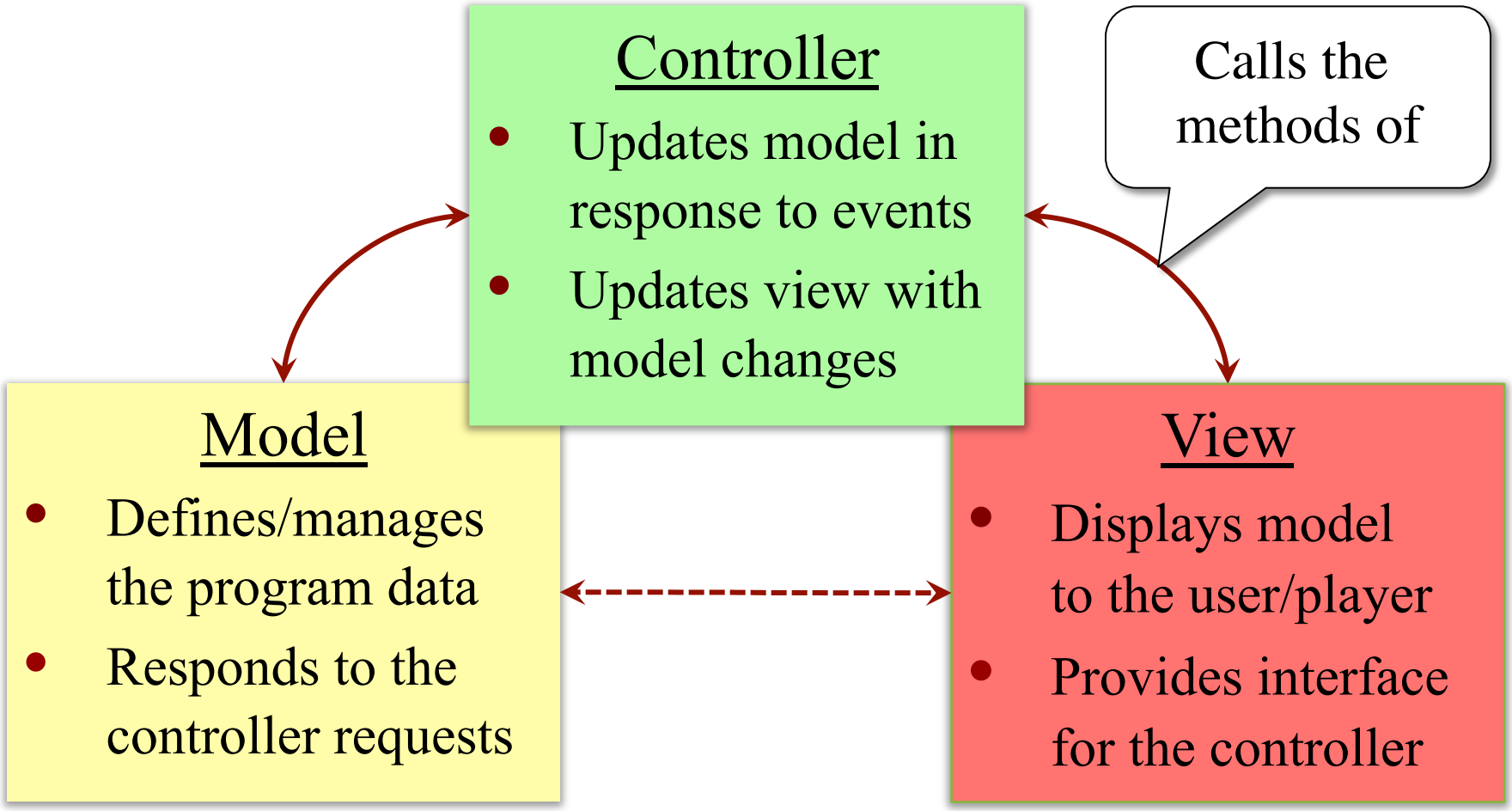
the gamedesigninitiative at cornell university

Architecture Patterns

- Essentially same idea as **software pattern**
  - Template showing how to organize code
  - But does not contain any code itself
- Only difference is **scope**
  - **Software pattern**: simple functionality
  - **Architecture pattern**: complete program
- Classic pattern: Model-View-Controller (MVC)
  - Most popular pattern in *single client* applications

the gamedesigninitiative at cornell university

Model-View-Controller Pattern



the gamedesigninitiative at cornell university

Example: Temperature Converter

- Model: (TemperatureModel.java)
  - Stores one value: fahrenheit
  - But the methods present two values
- View: (TemperatureView.java)
  - Constructor creates GUI components
  - Recieves user input but does not "do anything"
- Controller: (TemperatureConverter.java)
  - **Main class**: instantiates all of the objects
  - "Communicates" between model and view

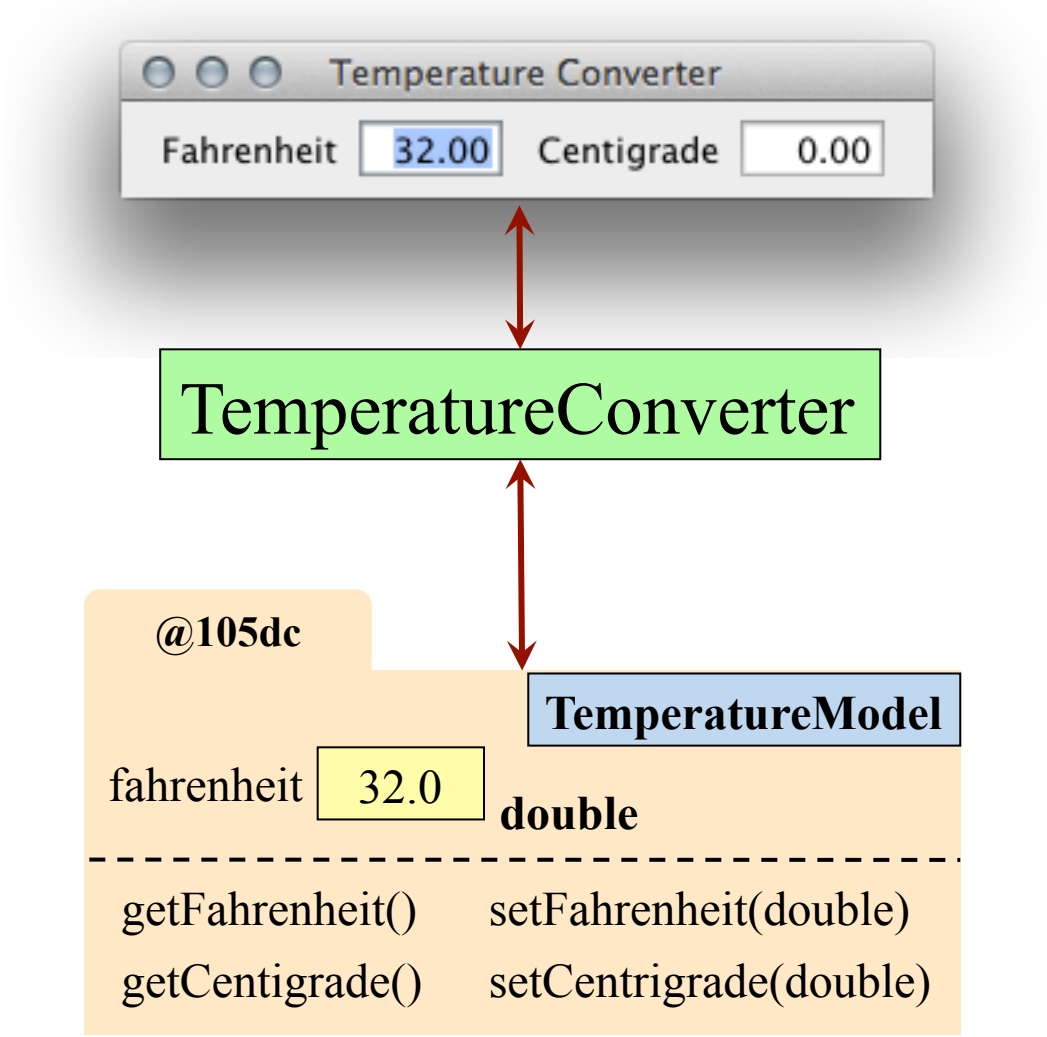
the gamedesigninitiative at cornell university

TemperatureConverter Example

View

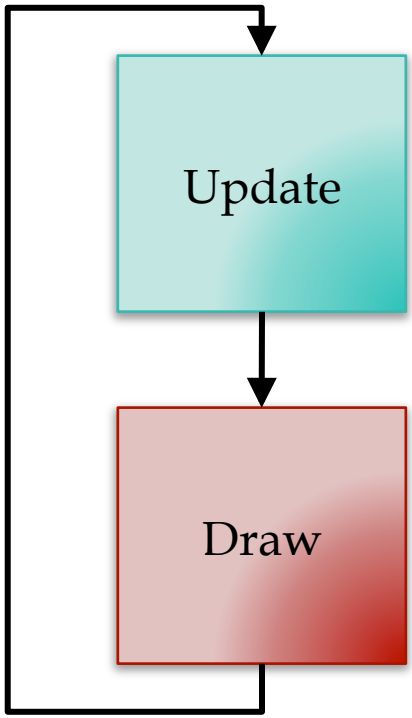
Controller

Model



The Game Loop and MVC

- Model: The game state
  - Value of game resources
  - Location of game objects
- View: The draw phase
  - Focus of upcoming lectures
- Controller: The update phase
  - Alters the game state
  - Topic of previous lecture



the gamedesigninitiative at cornell university

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"

the gamedesigninitiative at cornell university

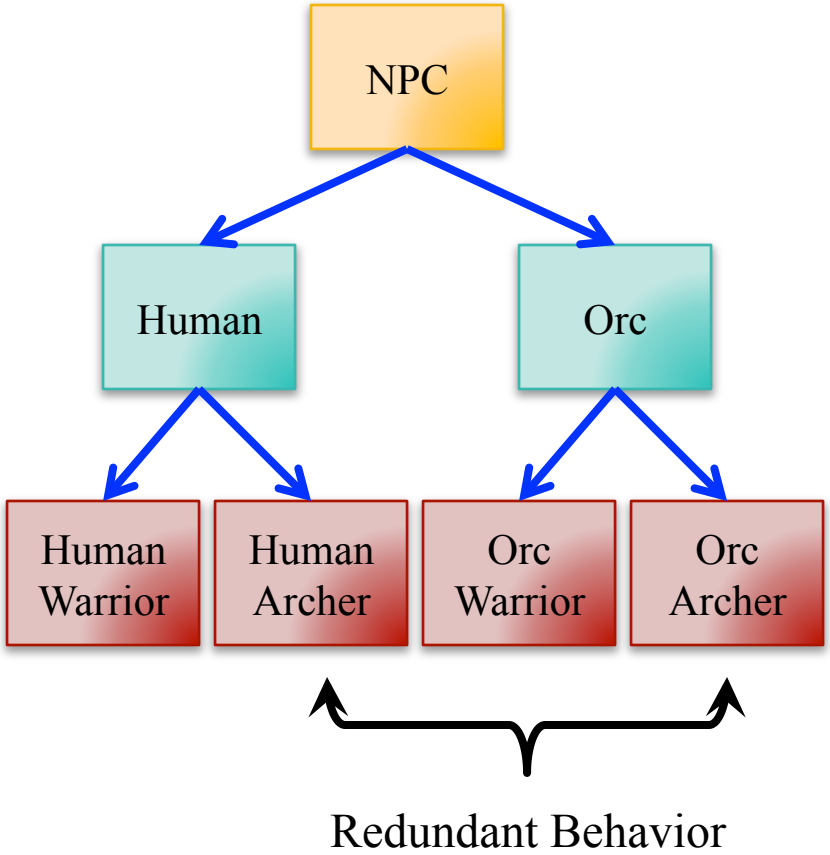
Classic Software Problem: Extensibility

- **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?

the gamedesigninitiative at cornell university

Problem with Subclassing

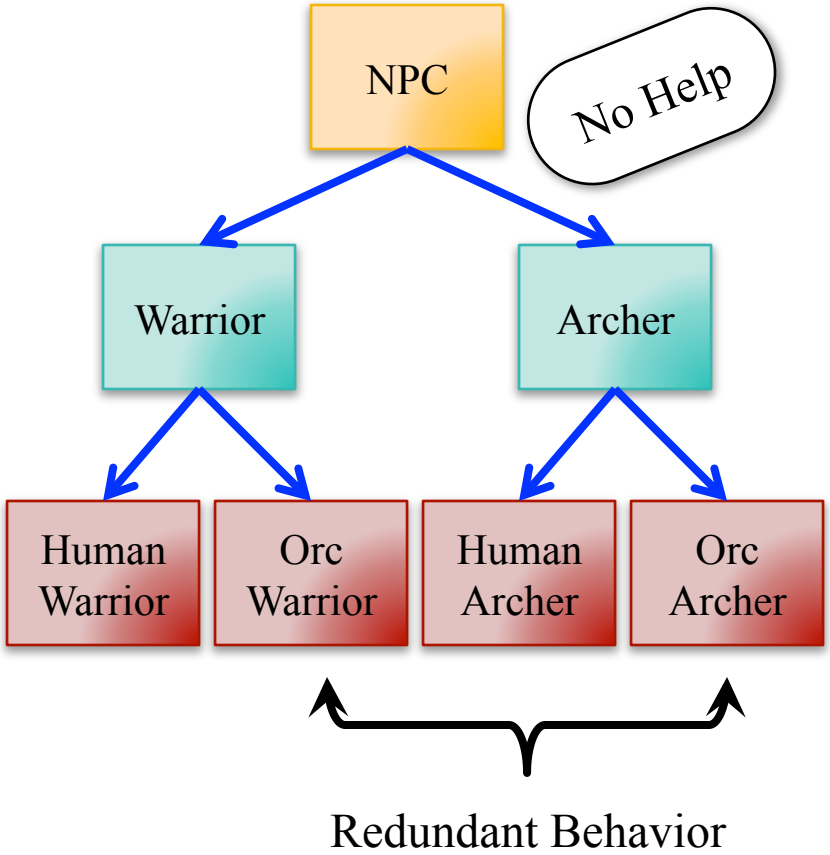
- Games have *lots* of classes
  - Each game entity is different
  - Needs its own functionality (e.g. object methods)
- Want to avoid **redundancies**
  - Makes code hard to change
  - Common source of bugs
- Might be tempted to **subclass**
  - Common behavior in parents
  - Specific behavior in children



the gamedesigninitiative at cornell university

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

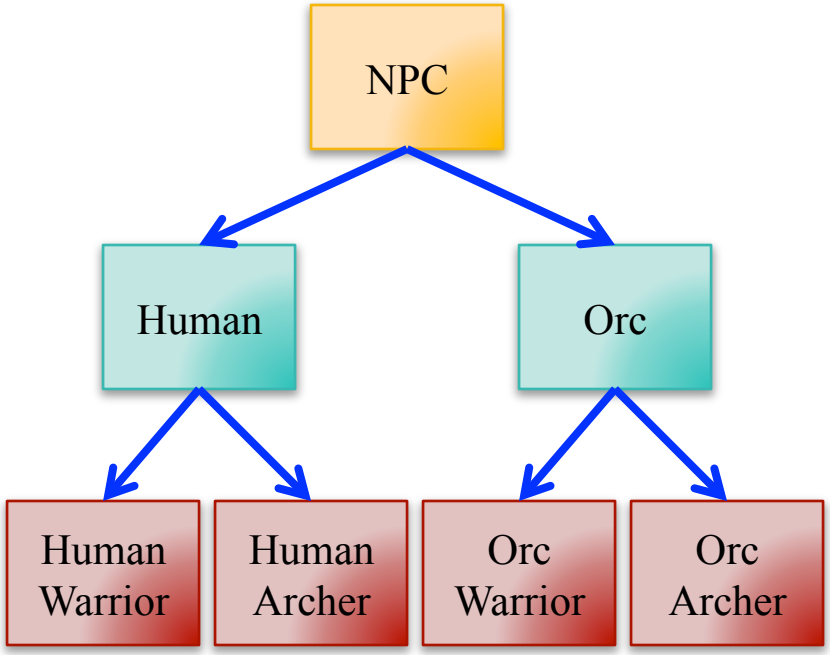


the gamedesigninitiative at cornell university

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

the gamedesigninitiative at cornell university

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

the gamedesigninitiative at cornell university

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

the gamedesigninitiative at cornell university

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

the gamedesigninitiative at cornell university

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` flee
  - Not necessarily tied to class
  - **Example**: swapping items

the gamedesigninitiative at cornell university

Programming and Parts of Speech

Classes/Types are Nouns

- Methods have verb names
- Method calls are sentences
  - subject.verb(object)
  - [illegible]
- Cla[illegible] *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 [illegible]
- [illegible]
- Relates to objects via *can-it*
  - **Example**: Orc can-it flee
  - Not necessarily tied to class
  - **Example**: swapping items

Will solve this next time.

the gamedesigninitiative at cornell university