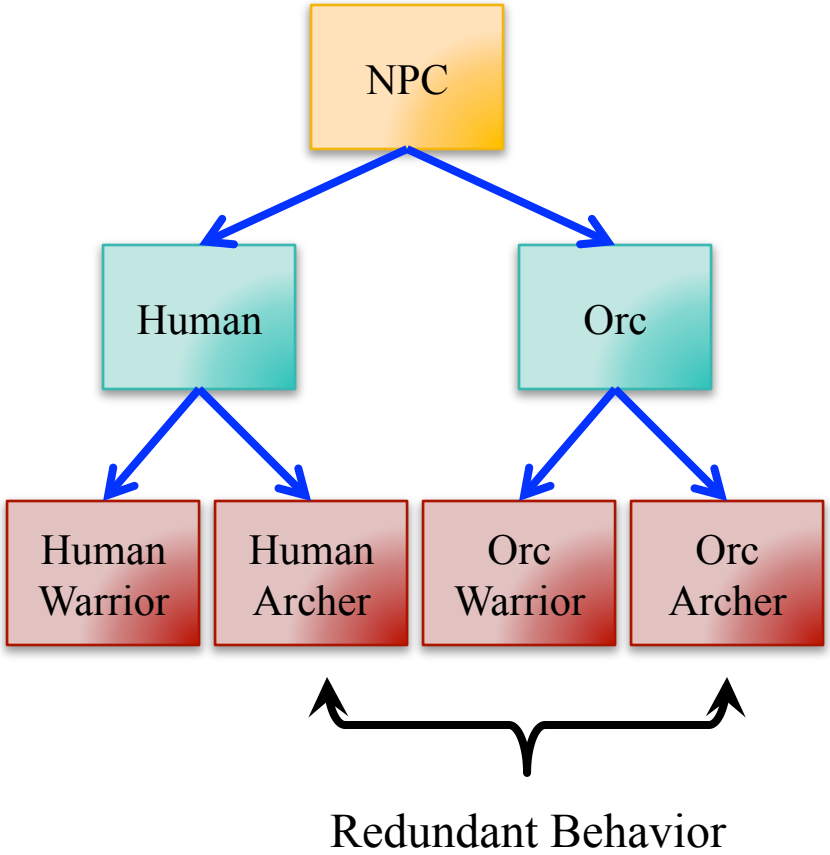
the gamedesigninitiative at cornell university

Lecture 8

Component Design

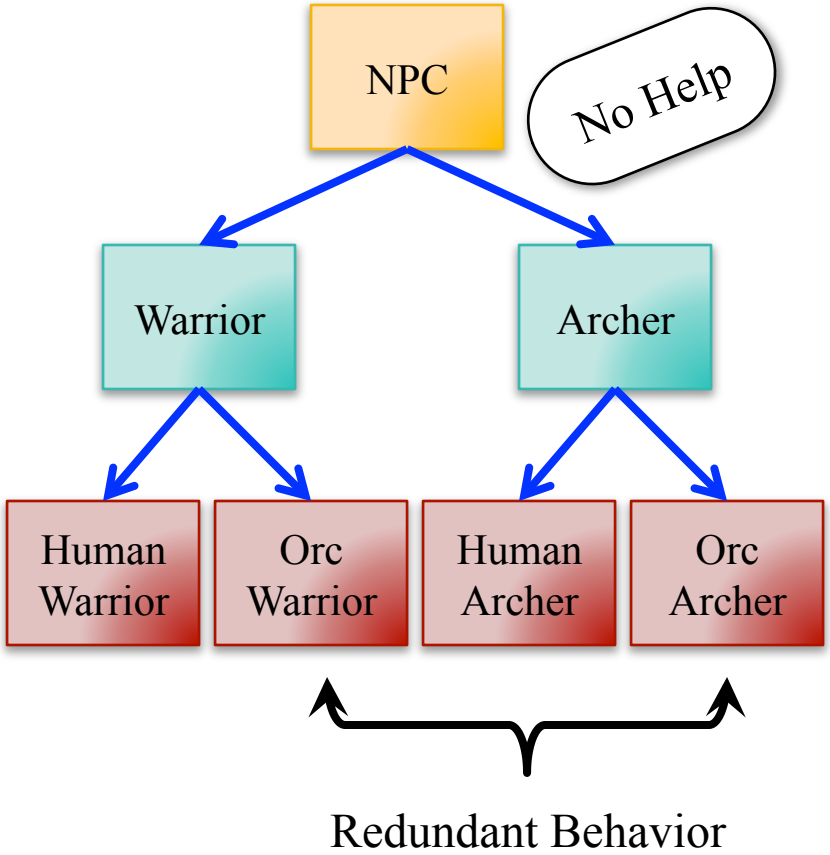
Recall: Problem with Subclassing

- 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

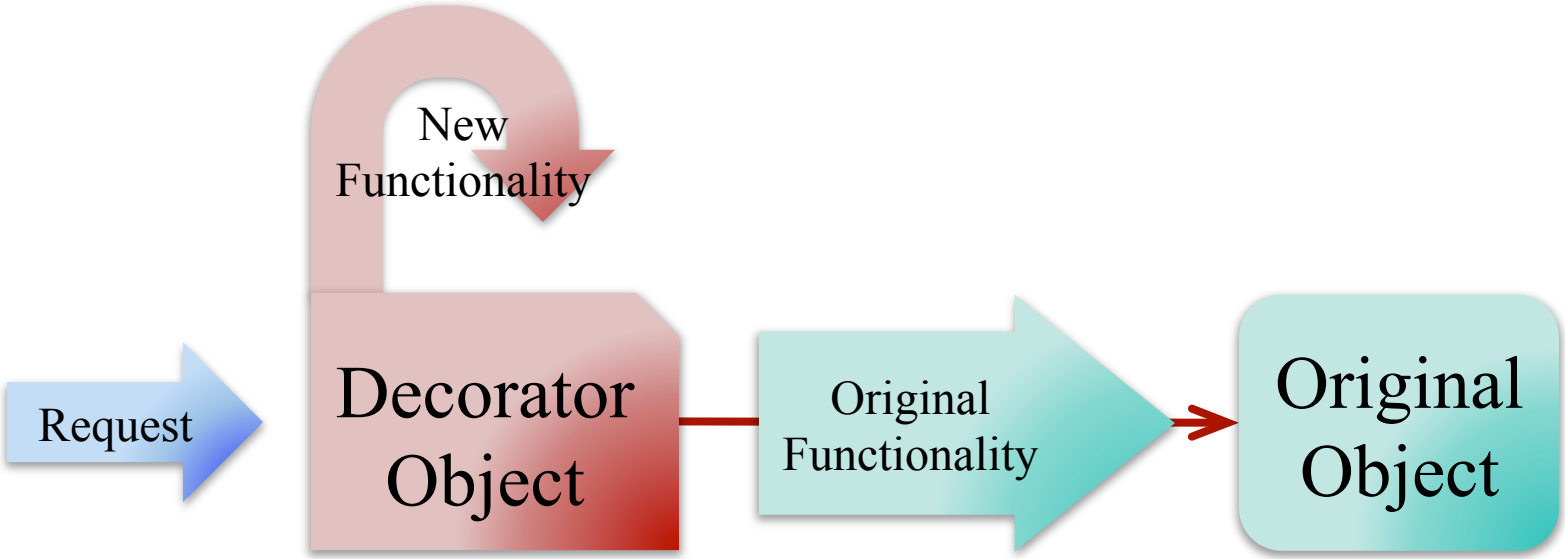


Recall: Problem with Subclassing

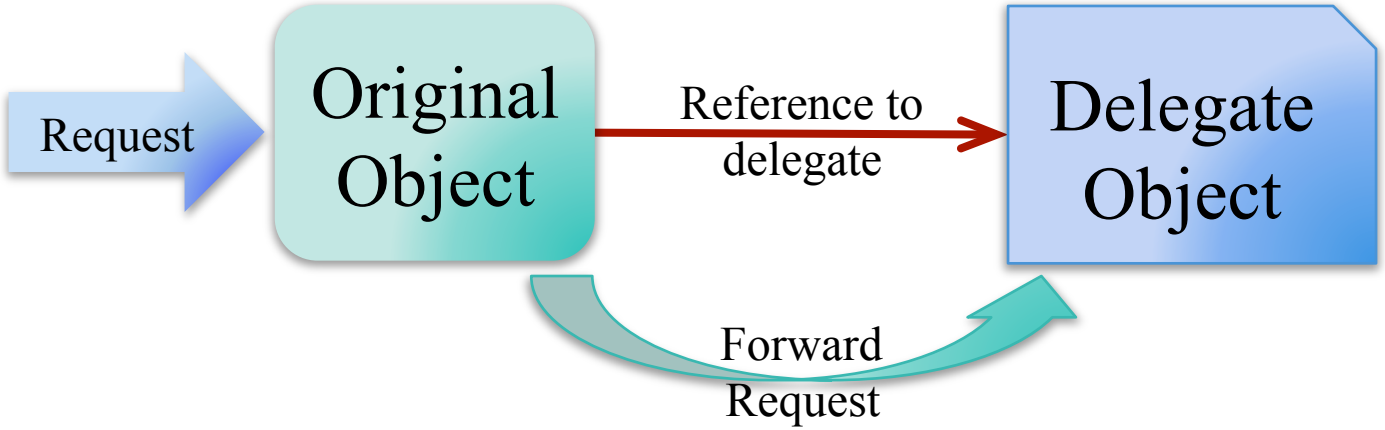
- Games have *lots* of classes
  - Each game entity is different
  - Needs its own functionality (e.g. object methods)
- Want to avoid **redundancies**
  - Makes code hard to change
  - Common source of bugs
- Might be tempted to **subclass**
  - Common behavior in parents
  - Specific behavior in children



Alternative: Decorator Pattern



Alternate: Delegation Pattern



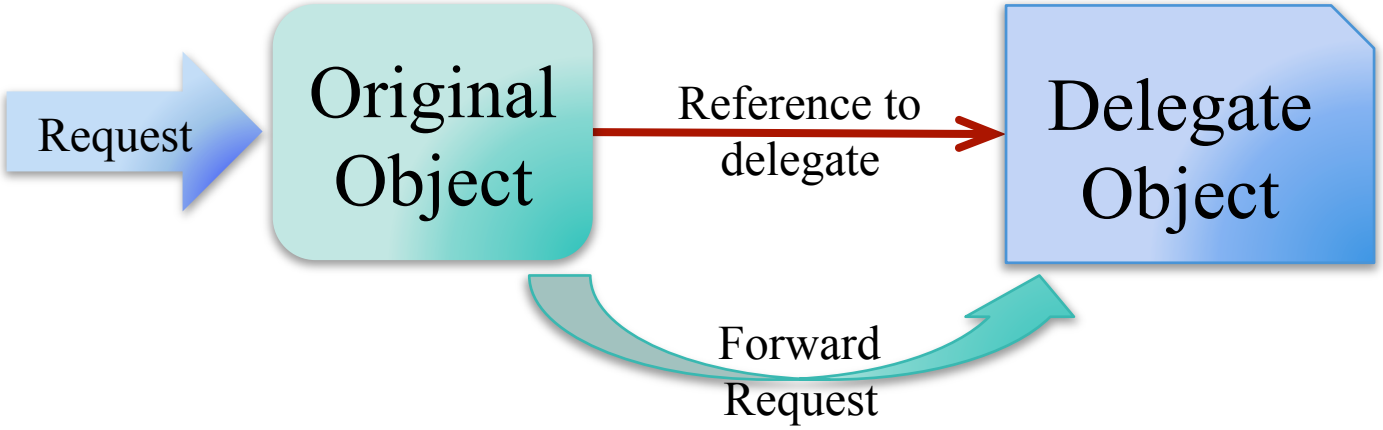
Inversion of the Decorator Pattern

Issues with Static Typing

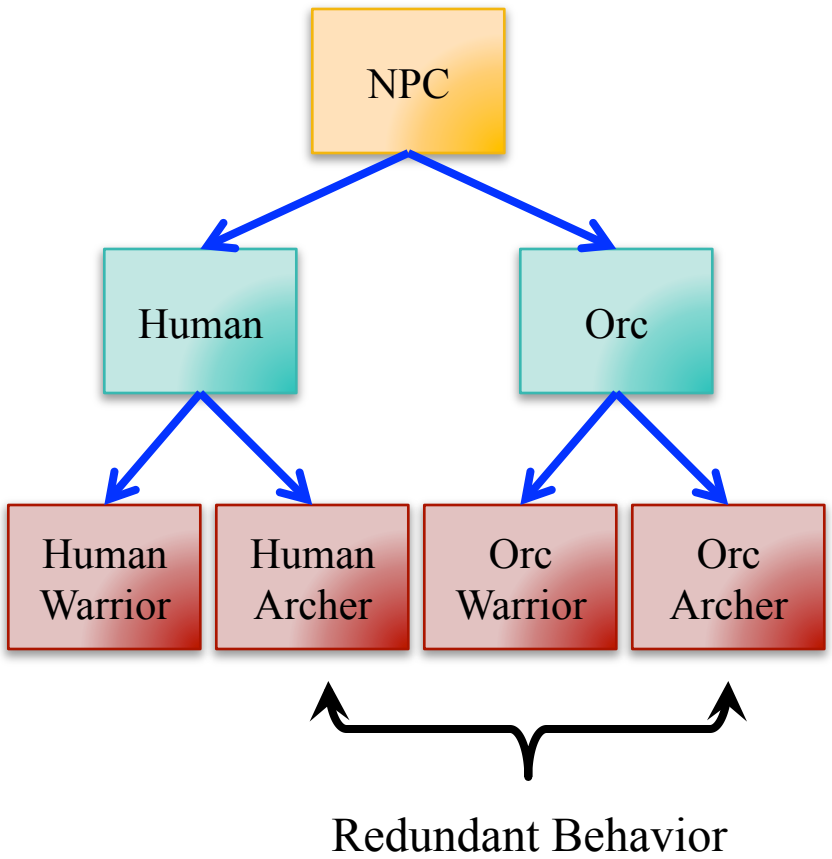
Original object class

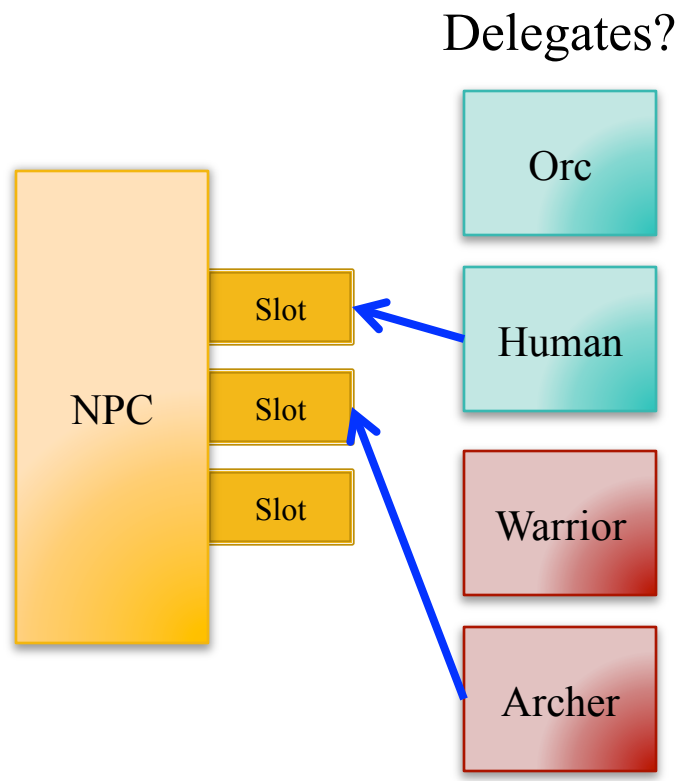
Method in original class

```
obj.request(arg1,..., argn)
```

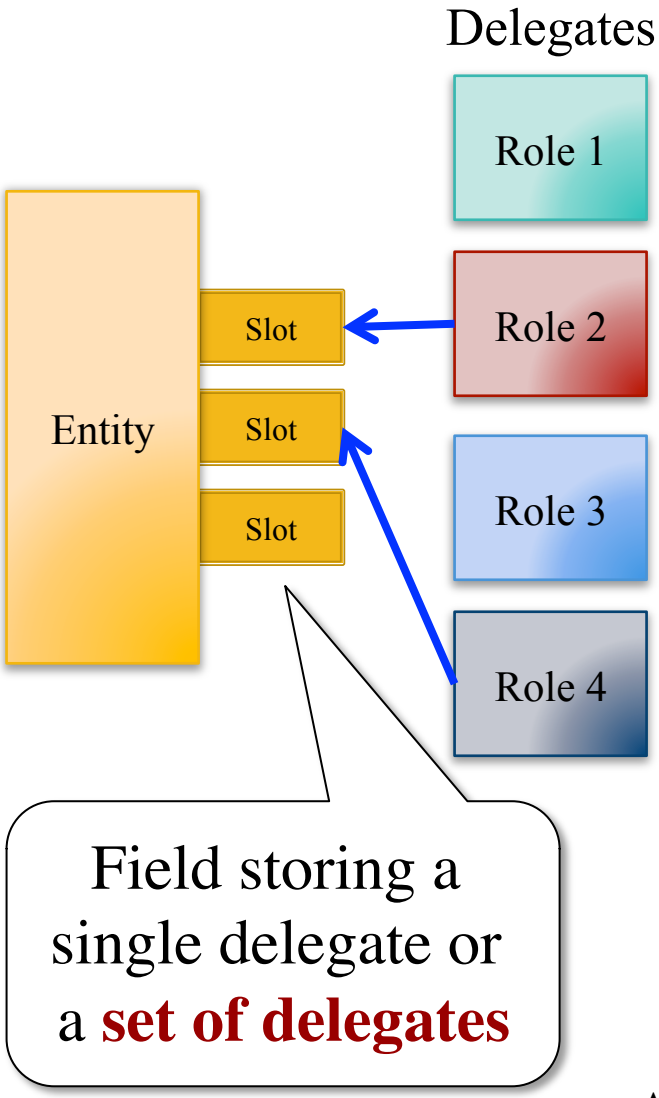


The Subclass Problem Revisited





Component-Based Programming



- **Role**: Set of capabilities
  - Class with very little data
  - A collection of methods
- Add it to object as *delegate*
  - Object gains those methods
  - Acts as a "function pointer"
- *Can-it*: search object roles
  - Check class of each role
  - Better than duck typing
  - Possible at compile time?

Aside: Objective-C

- Delegates are emphasized in **Objective-C**
  - ObjC *protocols* function like Java interfaces
  - But implement with a delegate, not a subclass
- Often leverages **message passing**
  - Method call is considered a message
  - Can call *any* method on *any* object
  - Will get runtime error if method not supported
  - Heavily optimized in latest Apple compilers

Aside: Objective-C

- Delegates are emphasized in **Objective-C**
  - ObjC *protocols* function like Java interfaces
  - But implement with a delegate [illegible] ass
- Often leverag[illegible]g
  - Meth[illegible]ered a message
  - Can call *any* method on *any* object
  - Will get runtime error if method not supported
  - Heavily optimized in latest Apple compilers

Addresses Static Typing Issue

Working With Static Typing

Uniform Method

```
o.doSomething(kitchenSink)
```

Direct Access

```
if (o has role) {
    extract role from o
    role.doSomething(info)
}
```

Working With Static Typing

Uniform Method

```
o.doSomething(kitchenSink)
```

Direct Access

```
if (o has role) {
    extract role from o
    role.doSomething(info)
}
```

Problem:
Multiple capabilities

Working With Static Typing

Uniform Method

```
o.doSomething(kitchenSink)
```

Direct Access

```
if (o has role) {
    extract role from o
    role.doSomething(info)
}
```

Problem:
Multiple capabilities

Problem:
Big switch statements

Entities Need Both Is-a and Can-it

Table

Chair

Objects share same capabilities *in theory*.
But certain actions are **preferred** on each.

Solving the Problem

Can-It

- Indicated by object **roles**
  - Component indicates role
  - Check type of each role
  - So again a typing solution
- "Formal duck typing"
  - Not limited to method name
  - Role type indicates action name, not a noun

Is-A

- Indicated by object **class**
- But could do more than that
  - Keep a bag of types
  - Contains of the descriptions that apply to this object
  - **Example**: table, heirloom
- Why would we do this?

Fast Can-It Testing: Bit Vectors

- Give each object a bit-vector to cover types
  - Each role is a position in the bit vector
  - **Example**: 0100101

Fast Can-It Testing: Bit Vectors

- Give each object a bit-vector to cover types
  - Each role is a position in the bit vector
  - **Example**: 0100101

    Can Build

    Can Fight

    Can Heal
  - Use array if more than 32/64 roles

Fast Can-It Testing: Bit Vectors

- Give each object a bit-vector to cover types
  - Each role is a position in the bit vector
  - **Example**: `0100101`

    Can Build

    Can Fight

    Can Heal
  - Use array if more than 32/64 roles
- Test is quick logical operation.
  - **Example**: `vector & CAN_FIGHT`
  - Faster than dynamic typing (why?)

Model-Controller Separation Revisited

Model

- Store/retrieve **object data**
  - Preserve any invariants
  - Data may include delegates
  - Determines `is-a` properties

Controller

- Process **interactions**
  - Look at current game state
  - Look for "triggering" event
  - Apply interaction outcome

Components

- Process **game actions**
  - Attached to a entity (model)
  - Uses the model as context
  - Determines `can-it` properties

Model-Controller Separation Revisited

Model

- Store/retrieve **object data**
  - Preserve any invariants
  - Data may include delegates
  - Determines is-a properties

Controller

- Process **interactions**
  - Look at current game state
  - Look for "triggering" event
  - Apply interaction outcome

Components

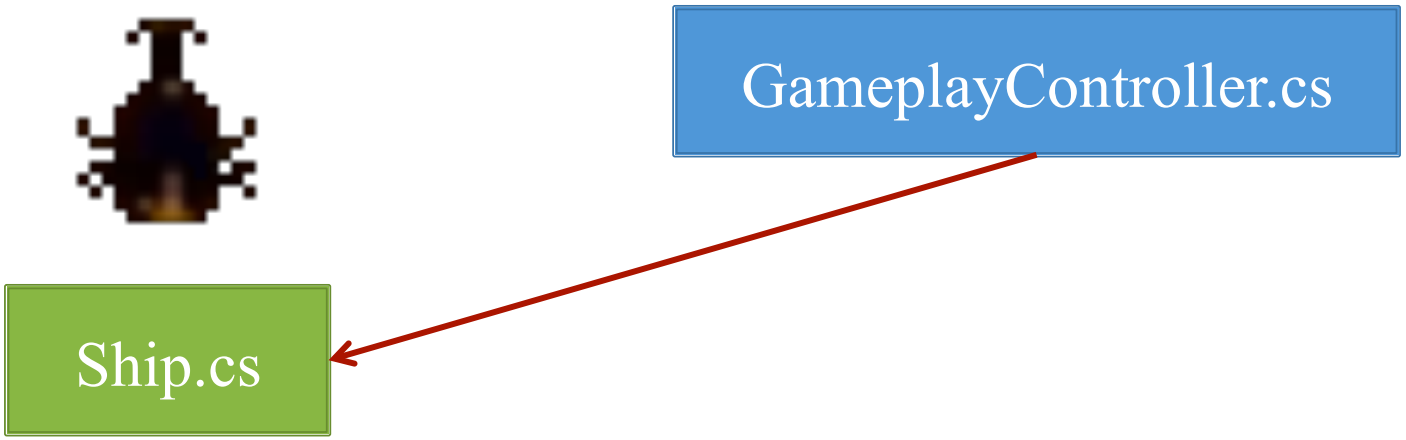
- Process **game actions**
  - Attached to a entity (model)
  - Uses the model as context
  - Determines can-it properties

?

Is this always the best idea?

Recall: Lab 3 in CS 3152

Processing Actions in Lab 3



- Decides whether to shoot
- Stores intent in the object
- But **DOES NOT** shoot

- Waits until objects commit
- Checks intent in Ship object
- Performs action for intent

Processing Actions in Lab 3

GameplayController.cs

Ship.cs

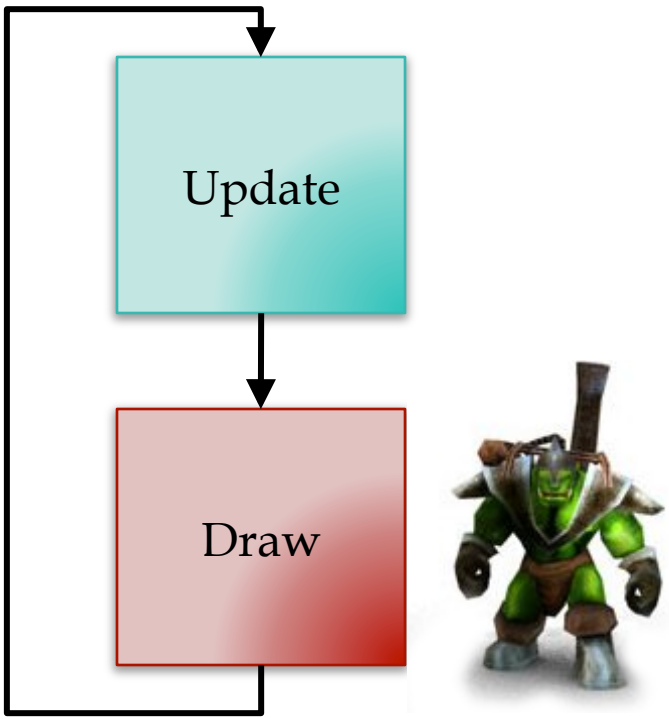
Attach components to controller?

- Decides whether to shoot
- Stores intent in the object
- But **DOES NOT** shoot

- Waits until objects commit
- Checks intent in Ship object
- Performs action for intent

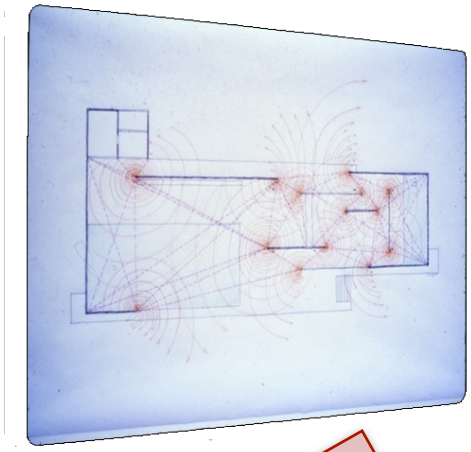
Related: Handling the View

- Way too much to draw
  - Backgrounds
  - UI elements
  - Individual NPCs
  - Other moveable objects
- Cannot cram all in Draw
- Put it in game object?
  - But objects are models
  - Violates MVC **again**



First Attempt: A Drawing Canvas

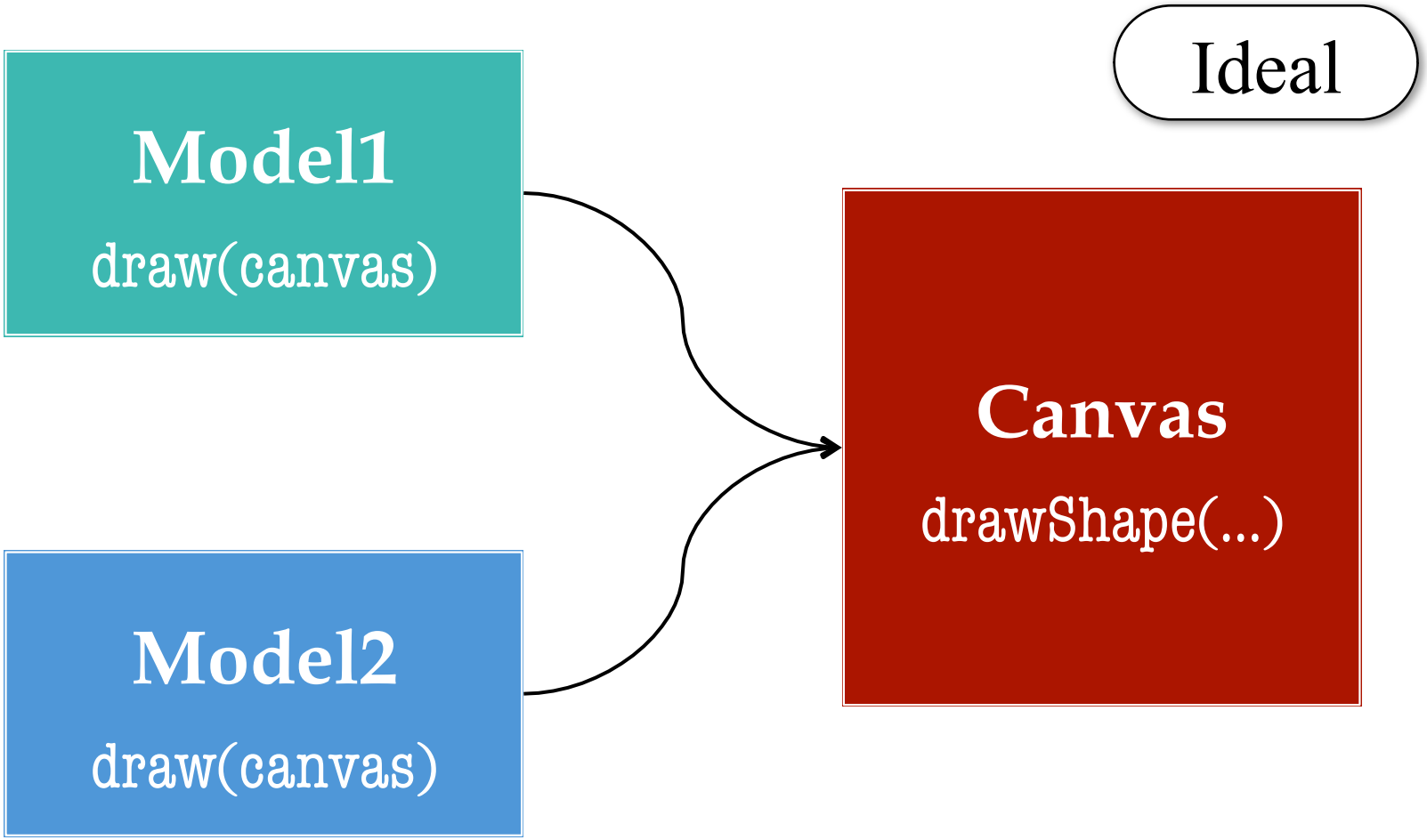
- Treat display as a **container**
  - Often called a canvas
  - Cleared at start of frame
  - Objects added to container
  - Draw contents at frame end
- Canvas abstracts *rendering*
  - Hides animation details
  - Like working with widget
- Implement `draw(c)` in model
  - Classic heavyweight model
  - No problems with extension



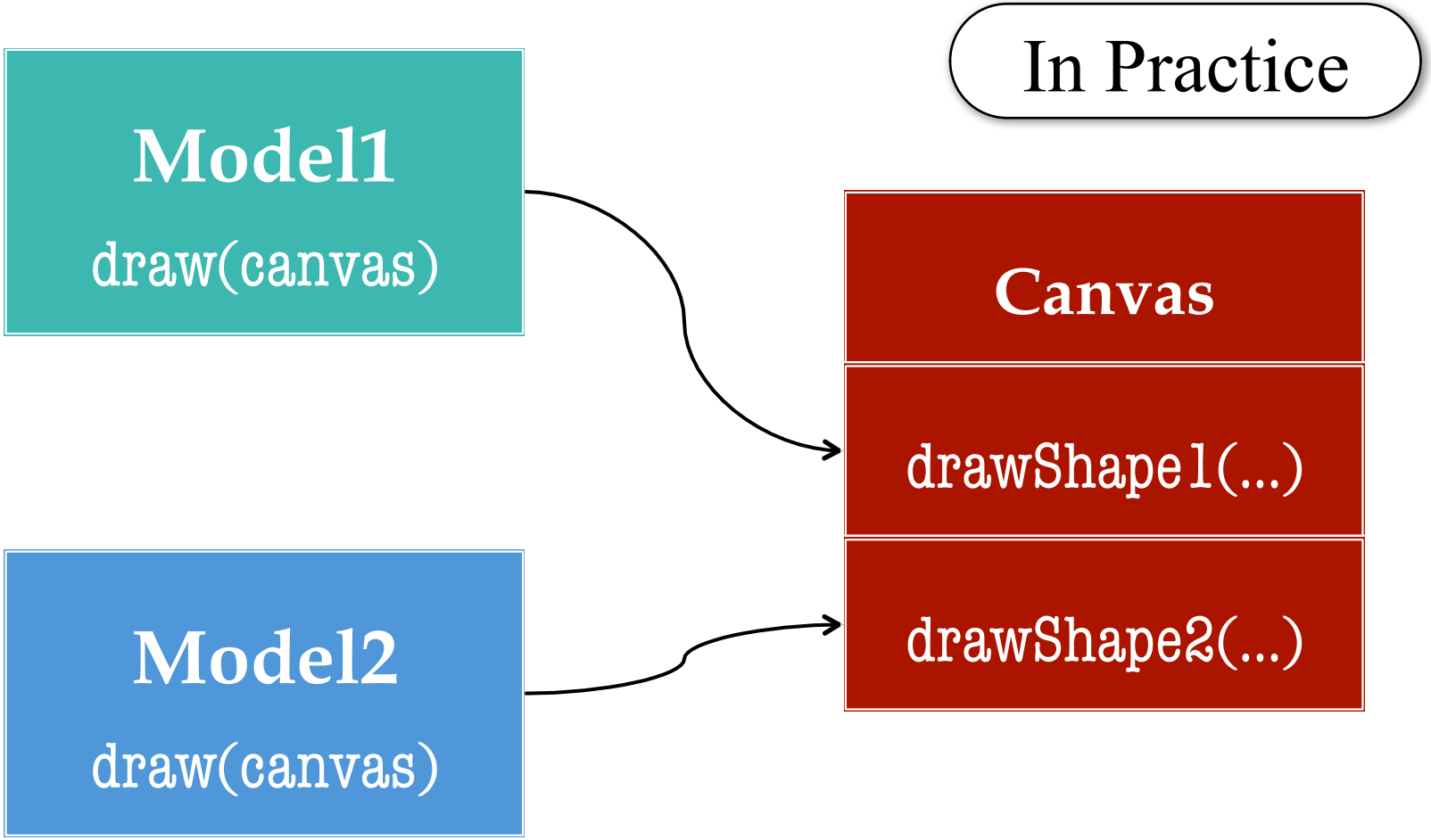
Passed as reference

```
void draw(Canvas c) {
    // Specify perspective
    // Add to canvas
}
```

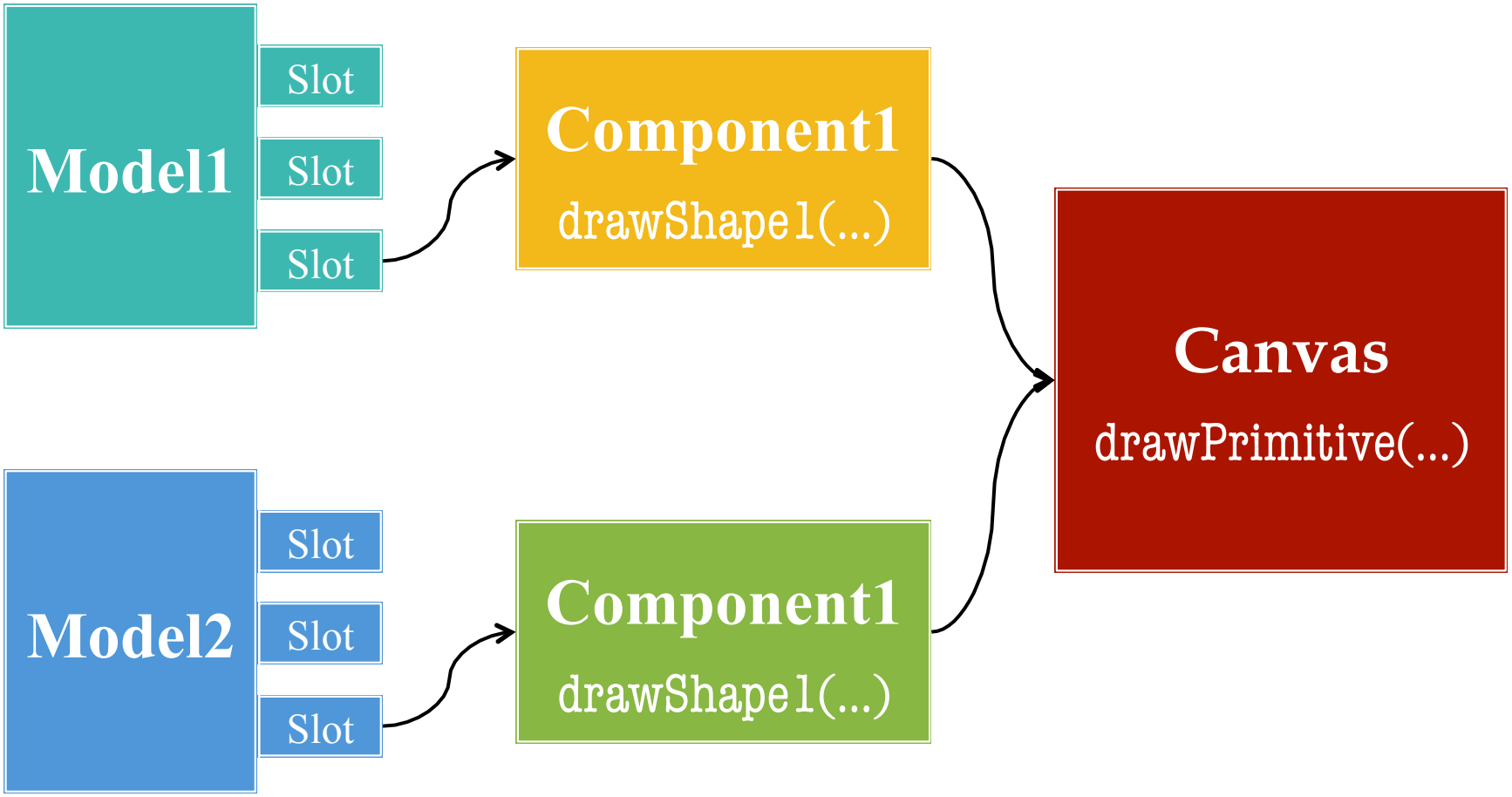
Problem: Canvas Methods



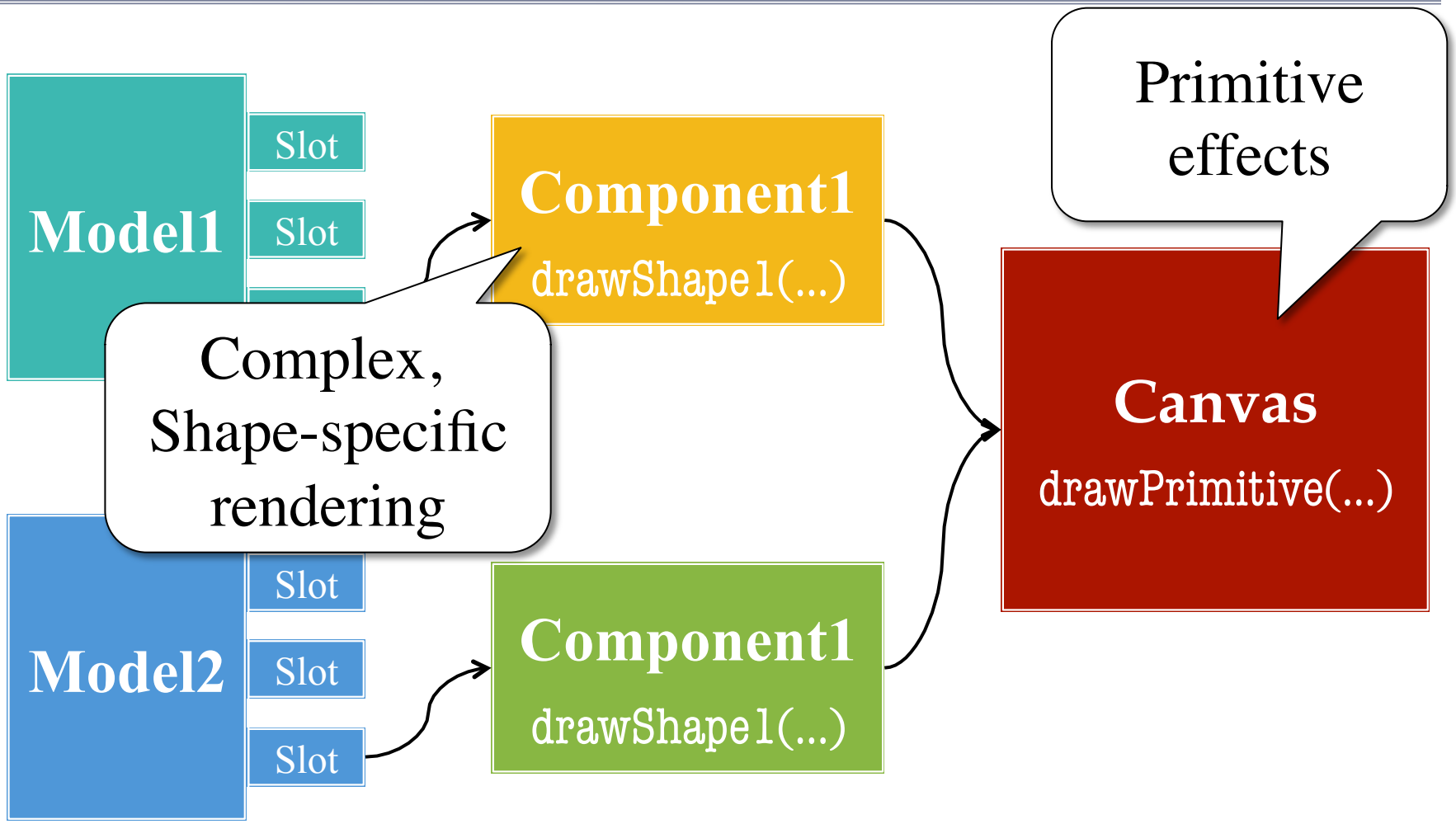
Problem: Canvas Methods



Views and Components



Views and Components



Components and AI

- NPCs use sense-think-act cycle
  - **Sense**: perceive the world
  - **Think**: choose an action
  - **Act**: update the game state
- **Act** is stored in components
  - Capabilities given by roles
- Where are **sense** and **think**?
  - In the component/role? Yes
  - How do roles interact? This is hard

Why Do They Need to Interact?

- Because you are trying to pick an action
  - NPC can only do one action (why?)
  - *Think* needs all roles to pick best one
- Why not let **each** component do an action?
  - Some things can be simultaneously (run + shoot)
  - Only concern is *conflicting* actions (left vs. right)
  - Can we guarantee components never conflict?
- Do we have to solve this in thinking-phase?

Recall: Rule-Based AI

- Have a list of **rules**
  - All if-then statements
  - Rules fires if satisfied
  - Need to pick one (**conflict resolution**)
- Often resolve by **order**
  - Each rule has a *priority*
  - Higher priorities go first
  - "Flattening" conditionals

R_1: `if` $event_1$ `then` act_1

R_2: `if` $event_2$ `then` act_2

R_3: `if` $event_3$ `then` act_3

R_4: `if` $event_4$ `then` act_4

R_5: `if` $event_5$ `then` act_5

R_6: `if` $event_6$ `then` act_6

R_7: `if` $event_7$ `then` act_7

Treat Components the Same

- Each component a rule
  - Chooses an action
  - Or possibly none
  - More complex than simple if-then rules
- **Conflict resolution**
  - Priorities to components
  - Or priorities to actions
  - Resolved *globally*

C_1: warrior component

C_2: archer component

C_3: human component

C_4: orc component

Implies a global AI **subsytem**

Non-Conflict Resolution?

- What if they do not conflict?
  - Pick just one?
  - Apply them both?
- Only if both **commutative**
  - Treat action as $f: \mathbb{S} \rightarrow \mathbb{S}$
  - Require $f(g(s)) = g(f(s))$
  - Easy if state is disjoint
- *Animation* is a big problem
  - Each action has animation
  - Same solution as state?
- Commutative, disjoint:
  - `move(dx,dy)`:

    `x = x + dx`

    `y = y + dy`
  - `damage(d)`:

    `hp = hp - d`
- Commutative, not disjoint:
  - 2 `move` actions
  - Addition commutes
  - **Example**: walk, push

Scalable Games Language

States				Effects		
id	x	y	health	vx	vy	damage
1	12	342	100	-10	3	50
2	43	12	100	0	5	0
3	123	90	95	9	9	0

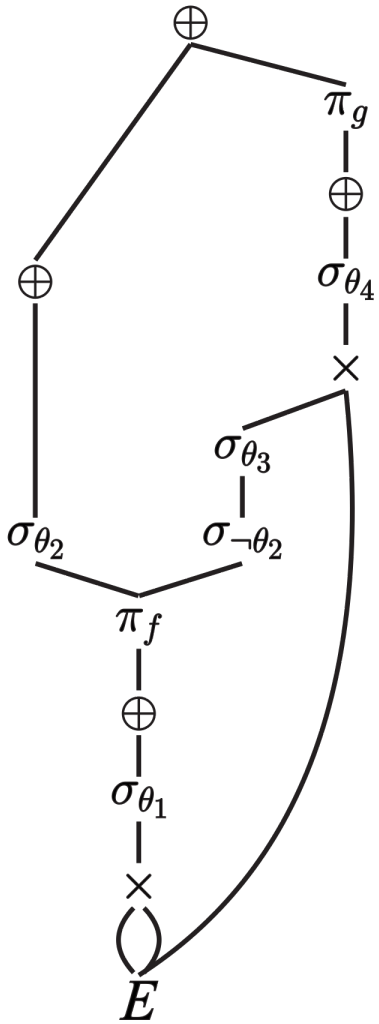
```
class Unit {
    public state int id;

    // State with update rules
    public state float x : (x+vx);
    public state float y : (y+vy);
    public state int hp: (hp-dmg);

    // effects
    public effect float vx : avg;
    public effect float vy : avg;
    public effect int dmg : sum;
    ...
```

- Assume **entities >> roles**
  - Common in certain genres
- Treat as a database problem
  - Entities are database rows
  - Roles are database queries
  - Gate checking is a DB join
- Updates must all commute
  - Language + schema design
  - Trivial conflict resolution
  - Surprisingly expressive

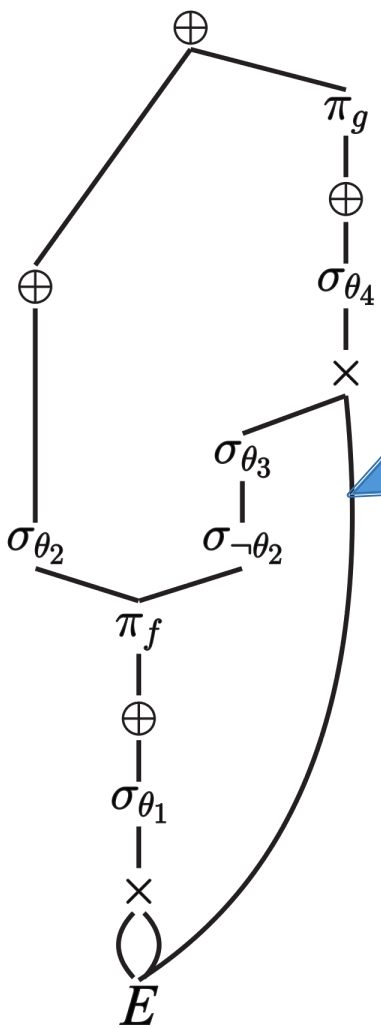
Scalable Games Language




```
public void run() {
    // Compute # skeletons, group center
    effect int c : sum;
    effect float sx : sum, float sy : sum;
    foreach(Unit u : Extent<Unit>) {
      if (isEnemySkeleton(u) &&
          dist(this,u) < range) {
          c <- 1; sx <- u.x; sy <- u.y;
      }
    }

    // If too many skeletons
    if (c > morale) {
      const float norm = (x-sx/c) * (x-sx/c)+
                         (y-sy/c) * (y-sy/c);
      // Run in opposite direction
      vx <- (x-sx/c)/norm;
      vy <- (y-sy/c)/norm;
    } ...
```

Scalable Games Language



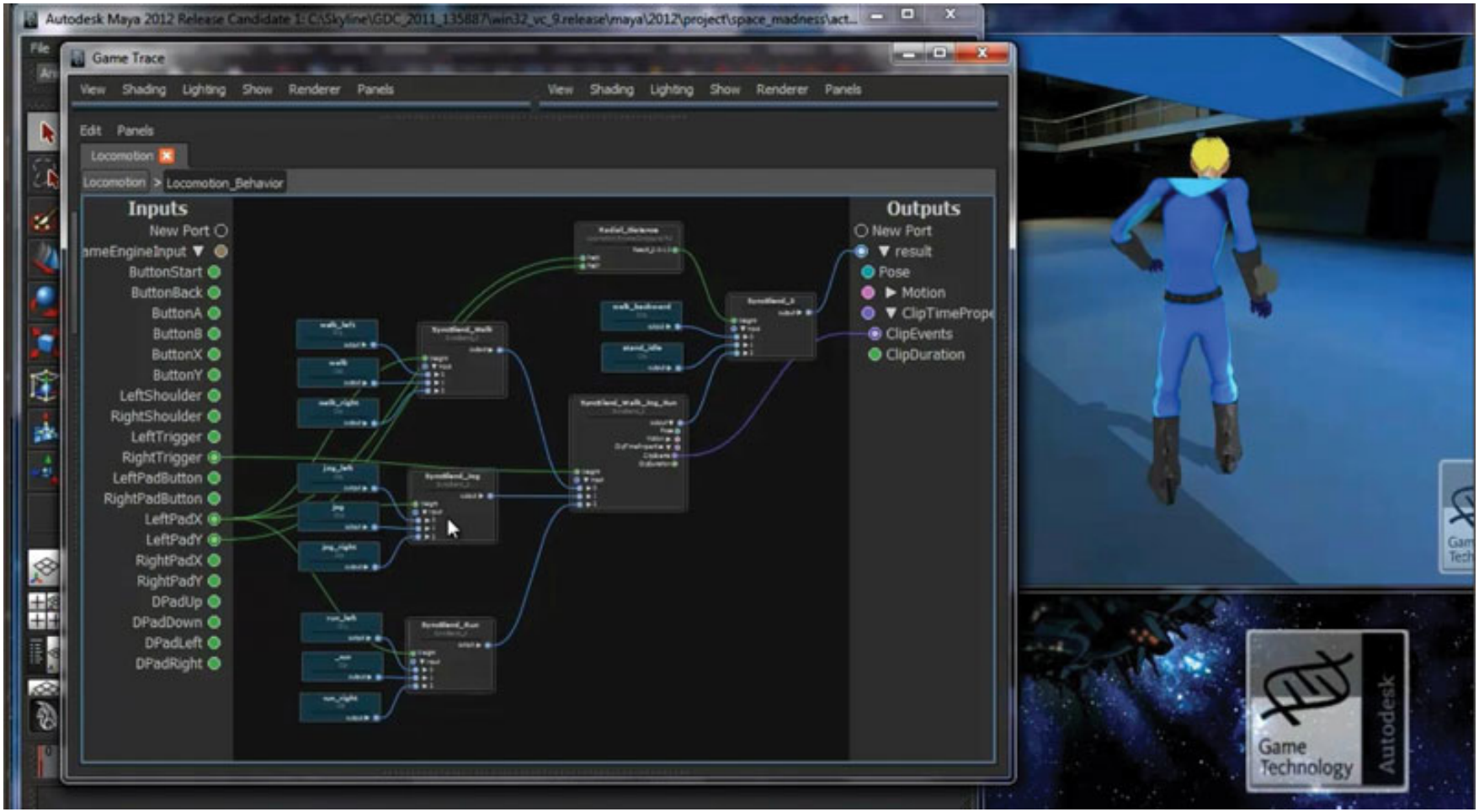

```
public void run() {
    // Compute # skeletons, group center
    effect int c : sum;
                                  t sy : sum;
                                  Unit>) {
                                  u) &&
                                  ) {
                                  u.y;
```

Classic DB optimizations

- Query rewrites
- Join selection
- Automatic indexing

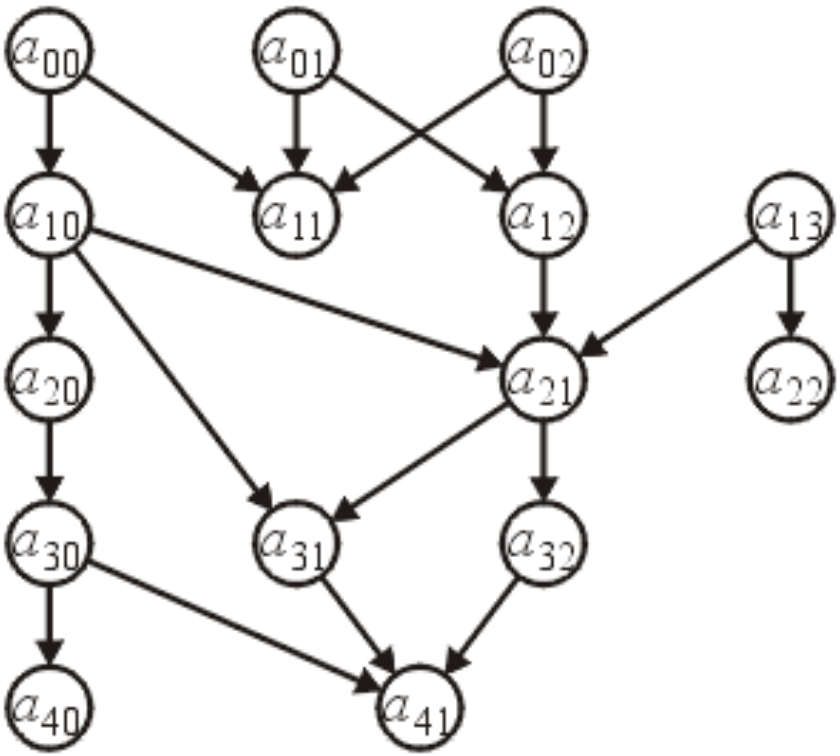
```
    // If too many skeletons
    if (c > morale) {
        const float norm = (x-sx/c) * (x-sx/c)+
                           (y-sy/c) * (y-sy/c);
        // Run in opposite direction
        vx <- (x-sx/c)/norm;
        vy <- (y-sy/c)/norm;
    } ...
```

Animation: Blend Trees



Conflict and Non-Conflict?

- Create a **DAG structure**
  - Directed acyclic graph
  - $a > b$ if a reachable from b
  - $a \mid b$ if neither $a < b$, $a > b$ (e.g. a, b are incomparable)
- Use for conflict resolution
  - If comparable, pick greatest
  - If incomparable commute
- Looks good, hard in practice
  - This is a **global** structure



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
  - Want **component-based design** to model actions
  - Interesting challenges when incorporating AI