CS 378: Computer Game Technology

Game Engine Architecture
Spring 2012

What is a Game Engine?

- Runtime system
  - Low-level architecture
    - 3-d system
    - Physics system
    - GUI system
    - Sound system
    - Networking system
  - High-level architecture
    - Game objects
      - Attributes
      - Behaviors
    - Game mechanics
- World editor
  - Tool(s) for defining world chunks (e.g. levels) and static and dynamic game objects

Game Engine Subsystems

- Runtime object model
- Realtime object model updating
- Messaging and event handling
- Scripting
- Level management and streaming
- Objectives and game flow management

What are Game Objects?

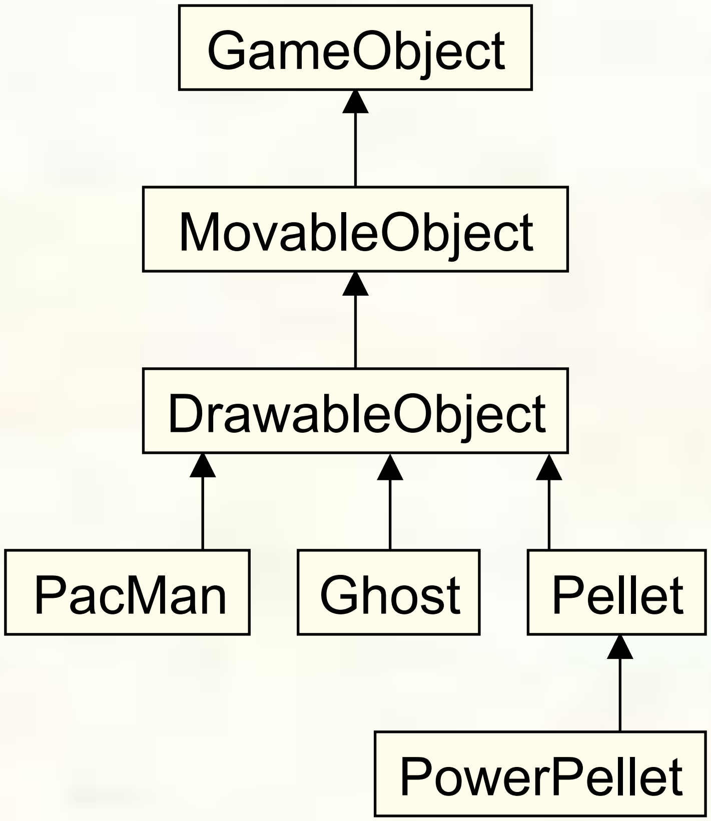
- Anything that has a representation in the game world
  - Characters, props, vehicles, missiles, cameras, trigger volumes, lights, etc.
- Created/modified by world editor tools
- Managed at runtime in the runtime engine
- Need to present an object model to designers in the editor
- Need to implement this object model at runtime efficiently

Runtime Object Model Architectures

- Object-centric
  - Objects implemented as class instances
  - Object's attributes and behaviors encapsulated within the class(es)
  - Game world is a collection of game object class instances
- Property-centric
  - Object attributes are implemented as data tables, one per attribute
  - Game objects are just IDs of some kind
  - Properties of an object are distributed across the tables, keyed by the object's id
  - Object behaviors implicitly defined by the collection of properties of the object
  - Properties may be implemented as hard-coded class instances
  - Like a relational database system in some ways

Object-centric Architectures

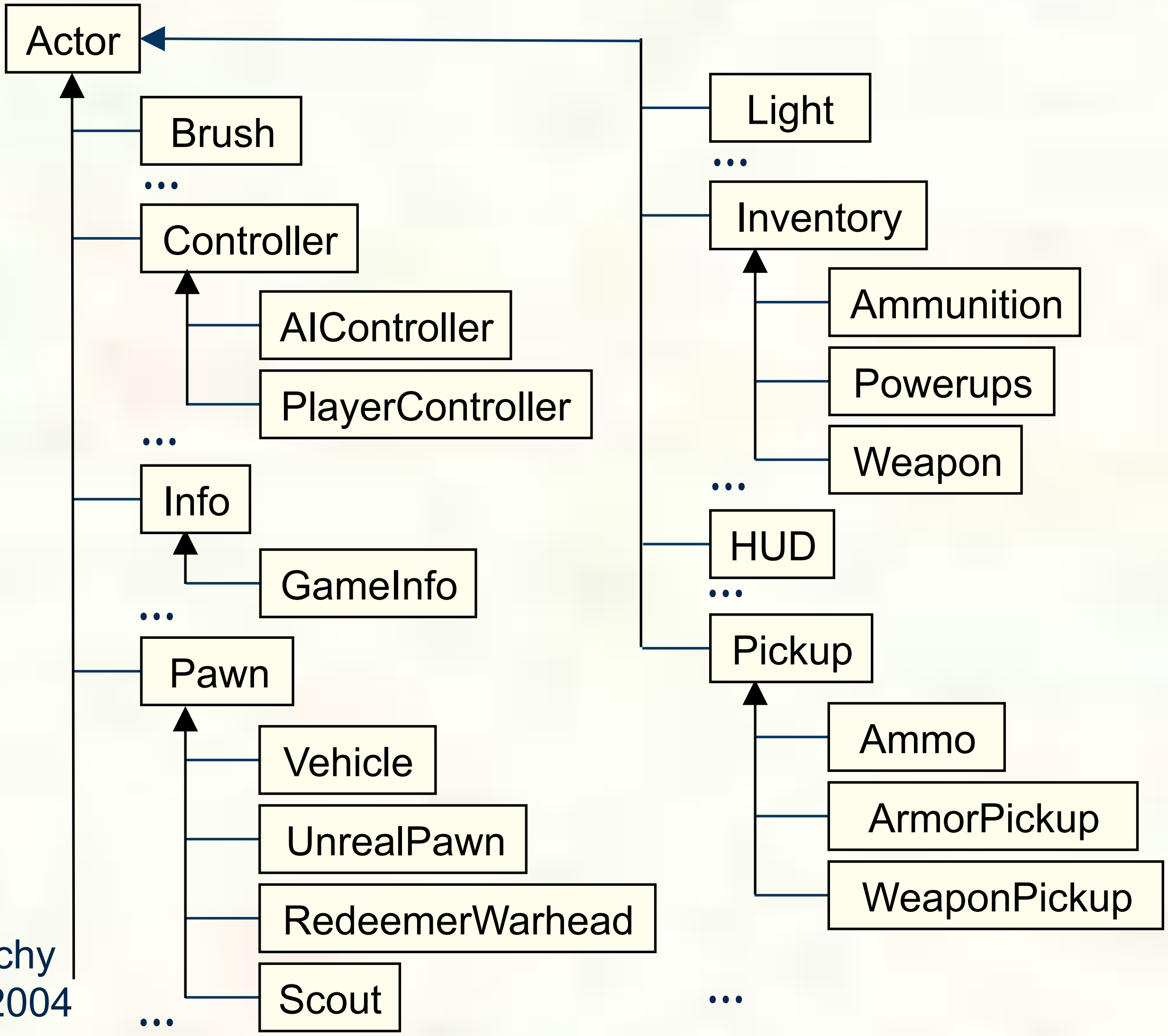
- Natural taxonomy of game object types
- Common, generic functionality at root
- Specific game object types at the leaves



Hypothetical PacMan Class Hierarchy

Monolithic Class Hierarchies

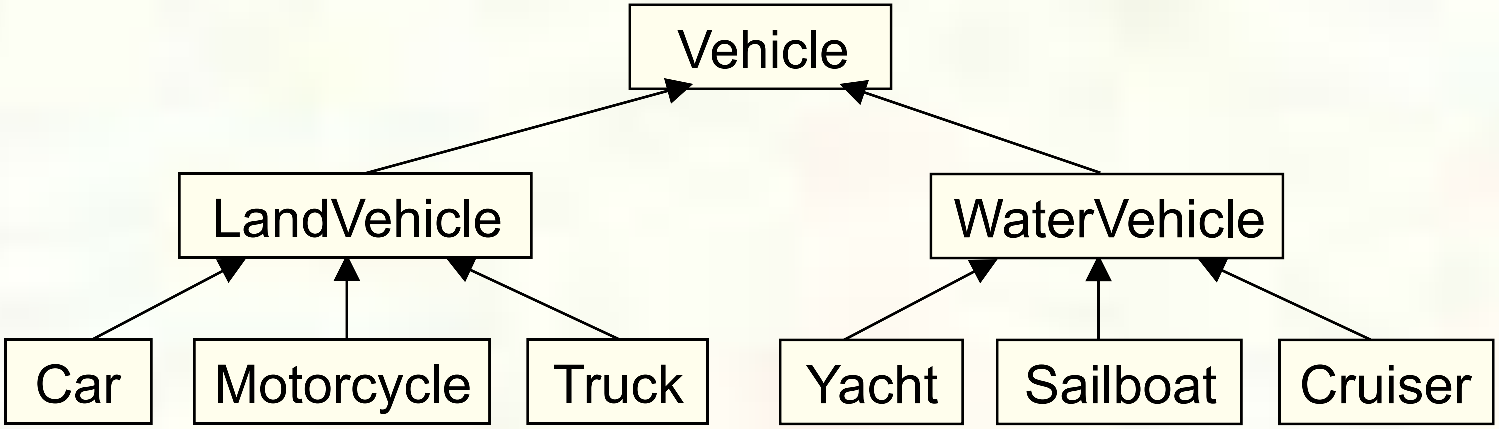
- Very intuitive for small simple cases
- Tend to grow ever wider and deeper
- Virtually all classes in the game inherit from a common base class



Part of object class hierarchy from Unreal Tournament 2004

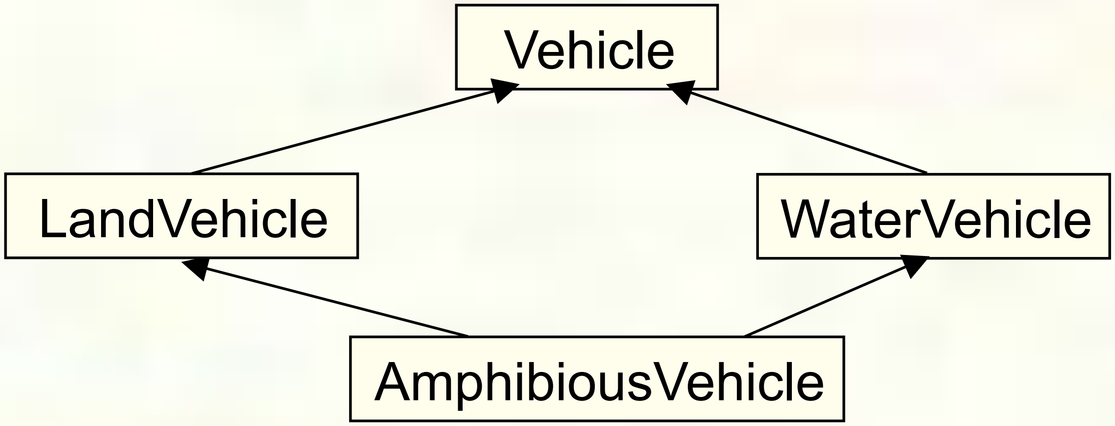
Problems with Monolithic Hierarchies

- Hard to understand, maintain, and modify classes
  - Need to understand a lot of parent classes
- Hard to describe multidimensional taxonomies
  - What if you want to classify objects along more than one axis?
  - E.g. how would you include an amphibious vehicle in the class hierarchy below?



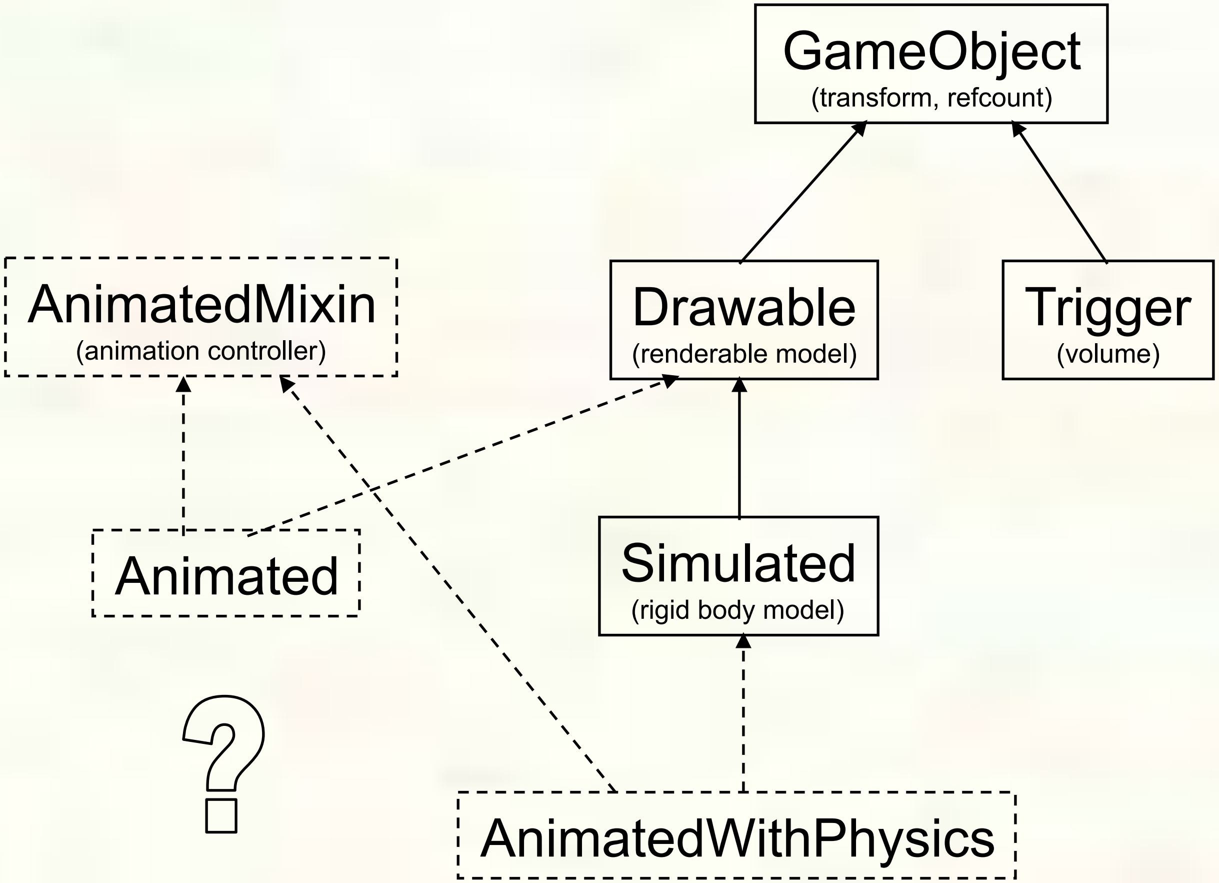
Tempted to use Multiple Inheritance?

- NOOOO!!!!!
- There's a reason languages like Java don't have it
- Derived classes often end up with multiple copies of base class members



Mix-in classes

- *Mix-in classes* (stand alone classes with no base class) can solve the deadly diamond problem
- Another approach is to use *composition* or *aggregation* in addition to *inheritance*

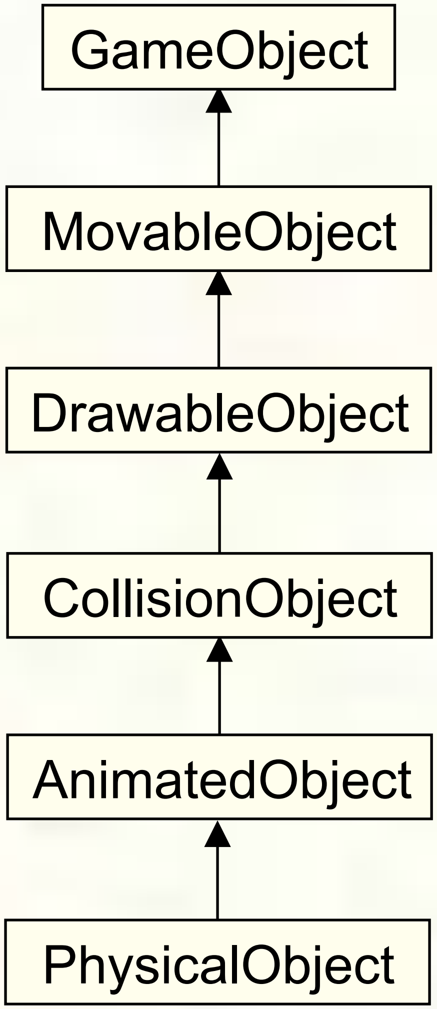


Observations

- Not every set of relationships can be described in a directed acyclic graph
- Class hierarchies are hard to change
- Functionality drifts upwards
- Specializations pay the memory cost of the functionality in siblings and cousins

Components vs. Inheritance

- A simple generic GameObject specialized to add properties up to full blown physical simulation
- What if (as in your current games) you want to use physical simulation on objects that don’t use skeletal animation?


GameObject

MovableObject

DrawableObject

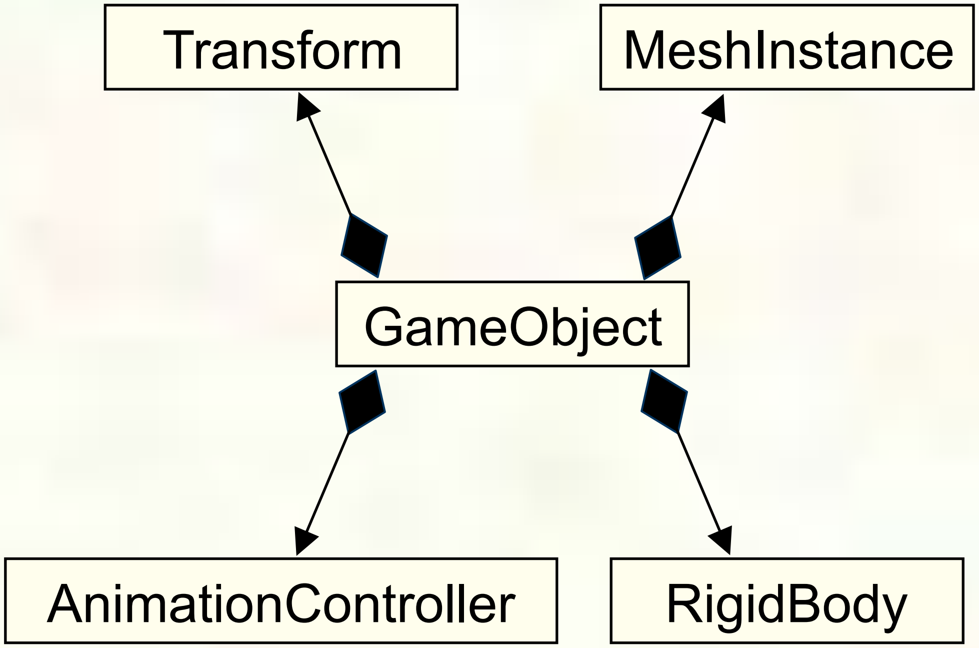
CollisionObject

AnimatedObject

PhysicalObject

Components vs. Inheritance

- One "hub" object that contains pointers to instances of various service class instances as needed.



Component-based example

```
class GameObject {
protected:
    //  My transform (position, rotation, scale)
    Transform m_transform;
    // Standard components
    MeshInstance* m_pMeshInst;
    AnimationController* m_pAnimController;
    RigidBody* mpRigidBody
public:
    GameObject() {
        // Assume no components by default.  Derived classes will override
        m_pMeshInst = NULL;
        m_pAnimController = NULL;
        m_pRigidBody = NULL;
    }
    ~GameObject() {
        // Automatically delete any components created by derived classes
        delete m_pMeshInst;
        delete m_pAnimController;
        delete m_pRigidBody;
     // …
};
```

Component-based example

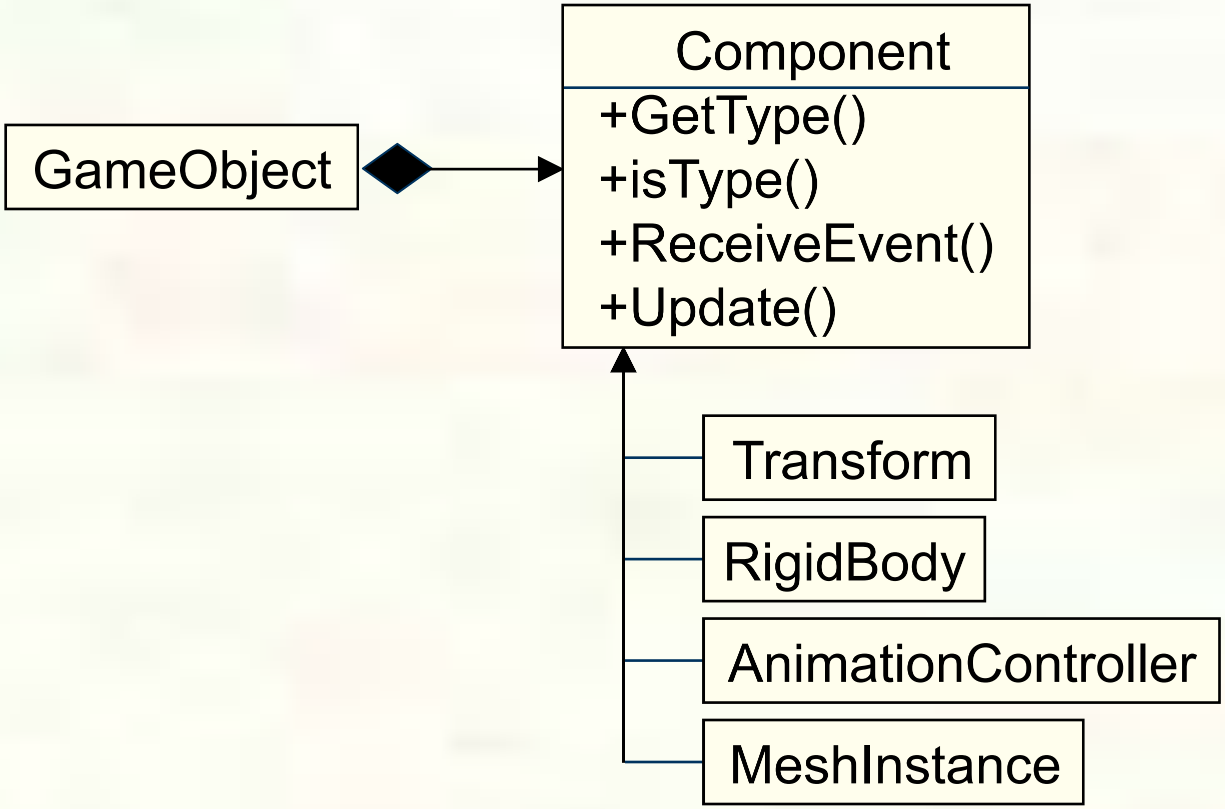
```
class Vehicle : public GameObject {
protected:
    // Add some more components specific to vehicles
    Chassis* m_pChassis;
    Engine*  m_pEngine;
    // …
public:
    Vehicle() {
        // Construct standard GameObject components
        m_pMeshInst = new MeshInstance;
        m_pRigidBody = new RigidBody;
        m_pAnimController = new AnimationController(*m_pMeshInst);
        // Construct vehicle-specific components
        m_pChassis = new Chassis(*this, *m_pAnimController);
        m_pEngine = new Engine(*this);
    }
    ~Vehicle() {
        // Only need to destroy vehicle-specific components
        delete m_pChassis;
        delete m_pEngine;
    }
};
```

Example properties

- “Hub” class owns its components (it manages their lifetimes, i.e. creates and destroys them)
- How does it know which components to create?
- In this simple case, the GameObject class has pointers to all possible components, initialized to NULL
- Only creates needed components for a given derived class
- Destructor cleans up all possible components for convenience
- All optional add-on features for derived classes are in component classes

More flexible (and complex) alternative

- Root GameObject contains a linked list of generic components
- Derive specific components from the component base class
- Allows arbitrary number of instances and types of components



Why not get rid of GameObject?

- If a GameObject instance becomes just an empty container of pointers to components with an object ID, why not just get rid of the class entirely?
- Create a component for a game object by giving the component class instance for that object the object's unique ID.
- Components logically grouped by an ID form a "game object"
- Need fast component lookup by ID
- Use factory classes to create components for each game object type
- Or, preferably use a "data driven" model to read a text file that defines object types
- How about inter-object communication? How do you send a message to an "object" and get the proper response?
  - Know a priori which component gets a given message
  - Multicast to all of the components of an object

Property-centric Architectures

- Think in terms of properties (attributes) of objects rather than in terms of objects
- For each property, build a table containing that property's values keyed by object ID
- Now you get something like a relational database
  - Each property is like a column in a database table whose primary key is the object ID
- Where are the object's behaviors defined?
  - Each type of property can be implemented as a *property class*
  - Do it with scripts, have one of an object's properties by ScriptID
  - Scripts can also be the target of messages

Pros and cons

- Pros
  - More memory-efficient
    - Only store properties in use, no unused data members in objects
  - Easier to construct in a data-driven way
    - Define new attributes with scripts, less recoding of class definitions
  - Can be more cache-friendly
    - Data tables loaded into contiguous locations in cache
    - Struct of arrays (rather than array of structs) principle
- Cons
  - Hard to enforce relationships among properties
  - Harder to implement large-scale behaviors if they're composed of scattered little pieces of fine-grained behavior
  - Harder to debug, can't just put a game object into a watch window in the debugger and see what happens to it.