Many complaints about OOP revolve around inheritance and its hierarchies

- Inflexible
- Hard to maintain
- Hard to understand
- Causes the very problems it’s trying to solve
EXAMPLE: 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
WHAT MONOLITHIC GIVES US

- Inheriting from a single base class works well with dynamic programming and systems
  - One place to implement all the features (reflection, serialization, garbage collection, etc) that we may want
- Allows the creation of a natural taxonomy of objects
  - Forms a directed acyclic graph of functionality
- Easy to reason about in many cases
PROBLEMS WITH MONOLITHIC HIERARCHIES

- Hard to understand, maintain, and modify classes
  - Need to understand a lot of parent classes
- Hard to describe multidimensional taxonomies
  - How to classify objects along more than one axis?
  - e.g. how would you include an amphibious vehicle?
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
  - Compiler cannot resolve ambiguities
MULTIPLE INHERITANCE

- C++ allows multiple inheritance
  - Can seem quite convenient if existing taxonomy doesn’t quite work in a particular case
- Problems arise since the constructor for the superclass is called when creating a derived class

```cpp
class Bar {
public:
  Bar();
};

class Foo: Bar {
public:
  Foo();
};
```

When Foo() is called, copy of Bar created then copy of Foo
SO WHAT HAPPENS WHEN WE CONSTRUCT FOO NOW?

class Bar {
    public:
    Bar();
};

class Baz: Bar {
    public:
    Baz();
};

class Foo: Bar, Baz {
    public:
    Foo();
};

1) Bar constructor called
2) Bar constructor called
3) Baz constructor called
4) Foo constructor called
THE DEADLY DIAMOND PROBLEM

- Two copies of all of Bar’s members
  - Bar::Foo::function()
  - Bar::Baz::Foo::function()
- Compiler ambiguities if Bar and Baz implement the same function
  - Call on Bar::Foo::function() or Bar::Baz::Foo::function()?
- Results in a compiler error
SOLVE WITH VIRTUAL INHERITANCE?

- Common C++ wisdom is use of virtual inheritance (i.e. virtual base classes) to prevent multiple copies

- Sure, but better idea: don’t use multiple inheritance
  - Assumptions about the hierarchical taxonomy may be flawed and need redesign
  - Not every object fits within a monolithic hierarchical taxonomy
Interfaces and Mix-ins in OOP

- Interfaces are an abstract type that does not contain data but does contain method signatures.
- Mix-ins are classes that contain functions which are usable by other classes that do not inherit from the mix-in class.
- These paradigms allow for single-inheritance languages to express multiple types of functionality without multiple inheritance issues.
  - High-level concepts -- actual implementation will be language-specific.
- C++ does not natively support either of these.
  - Create interfaces using pure virtual functions.
  - Create mix-ins using...multiple inheritance...
MIX-IN EXAMPLE

AnimatedMixin
(animation controller)

GameObject
(transform, refcount)

Drawable
(renderable model)

Trigger
(volume)

Simulated
(rigid body model)

Animated

AnimatedWithPhysics
MOVING BEYOND TAXONOMIES

- Classical inheritance is an “is-a” relationship
  - What are the defining features of an object’s existence?
  - Allows for deep and complex taxonomy of objects
- Also possible to treat objects as a collection of other objects
  - Creates a “has-a” relationship
  - What is the functionality of the objects that an object possesses?
  - Allows for the deep and complex composition of objects
COMPOSITION

- Object contains subobjects that implement desired functionality
  - Composition: object can own the subobject (i.e. subobjects share main object’s life cycle)
  - Aggregation: object contains the subobject (i.e. subobject does not share main object’s life cycle)
- High level principle of how and when to split functionality
  - Can be implemented using interfaces, mix-ins, delegates, etc
One “hub” object contains pointers to instances of various service class instances as needed (e.g. composition).

Note: Filled diamond indicates composition; unfilled diamond indicates aggregation
USING COMPOSITION

- "Hub" class owns its components and manages their lifetimes (i.e. creates and destroys them)

- Naive component creation:
  - 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 list of generic components
- Derive specific components from the component base class
- Allows arbitrary number of instances and types of components
EXAMPLE: UE4 AND UACTORCOMPONENTS

Creates new subobject associated with BP

Subobject inherited from C++ parent class
THINKING ABOUT OOP, COMPONENTS, AND INHERITANCE

- Consider the principles of OOP we discussed last time
  - Encapsulation
  - Abstraction
  - Inheritance
  - Polymorphism

- How useful are these in practice?

- What are the trade offs in large systems like a game engine?

- How well do the ideas of inheritance and components help or hinder these concepts?

- Are there other concepts we should be considering in game development?