Object Oriented Software Engineering

CS134

Chris Pollett

Sep.8, 2004

Outline

- Aspects of Object Orientation
- Object-oriented analysis
- Encapsulation, inheritance, and polymorphism
- Composition and Delegation
- OO Design Principles
- The Code Interface

Aspects of Object Orientation

- As projects get larger need some way to organize the program to maintain comprehensibility.
- Successfully higher level language have been developed to simplify this process.
- For CS134 we are using OO to accomplish this task. Three stages: OO Analysis (OOA), OO Design (OOD), and OO Programming (OOP).

Object-oriented analysis

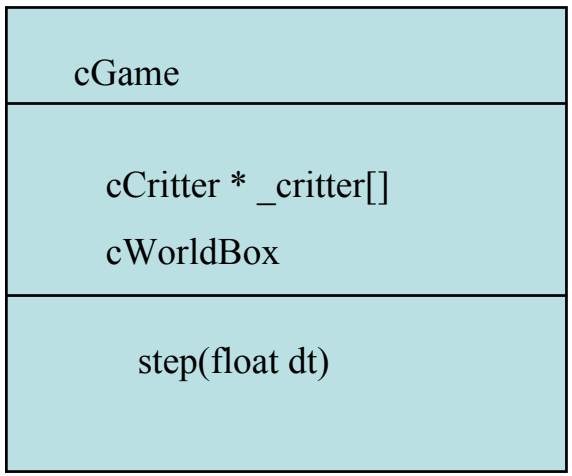
- Goal is to figure out which classes you should use.
- To do this one might:
  - Circle nouns and noun phrases from the requirement description in blue. The circle the verbs in red.
  - The nouns will be potential classes. The verbs potential methods.

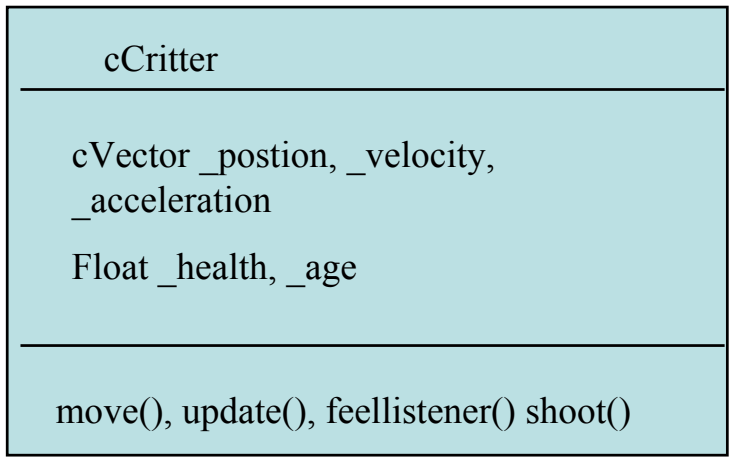
Pop Example

- Requirement: a framework for computer games with moving critters. The critters are draws as polygons, bitmaps, or animated loops of bitmaps. The world includes forces like gravity and friction. The critters listen to mouse-keyboard controls…
- Nouns: game, critter, polygon, bitmap, …
- Actions: draw, listen …

Pop Example cont'd

- Might come up with classes like:





In OOA you should

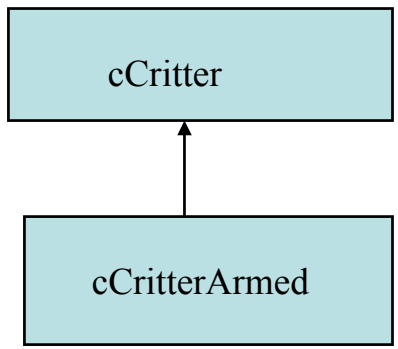
- Use simple diagrams
- Revise often, until everything is organized
- Use multiple diagram rather try to squeeze everything on one diagram
- Test out diagram under different use cases.

Encapsulation, inheritance, and polymorphism

- As said before putting data within object is called *encapsulation.*
- Two other related concepts are *inheritance* and *polymorphism*.

Inheritance

- The idea of inheritance is that if you have already have a class that is close to what you need it is often a good idea to subclass to get the class you want.



More Inheritance

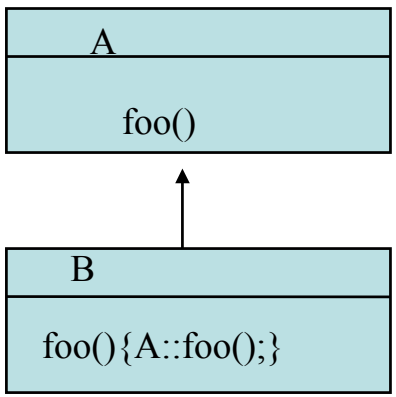
- Where inherit a class from another you might want to *override* some existing function.
- For example, in cCritter might override the update() function which changes the critter's state according to the current game world…
- This might be used in cGame where an array of cCritter's is kept. By calling biota[i]->update() can invoke whichever kind of critter's update function.
- Note need pointers to exhibit polymorphic behavior.

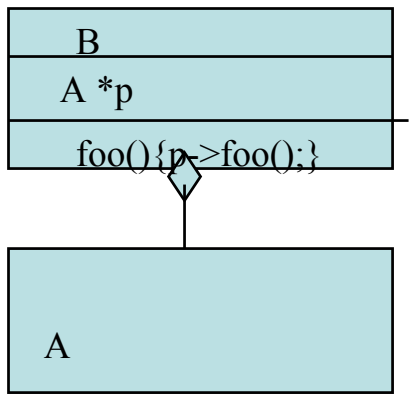
Some OO terminology

- For us, sometimes call public methods of a class the *interface* of the class.
- Member functions which return internal variable values called *accessors*.
- Functions which changed internal variable values called *mutators*.
- If a method has not been implemented called *abstract*

Composition and Delegation

- ClassA is *composed* with ClassB if it has ClassB or ClassB* as a member.
- Can replace inheritance by composition.





More Composition and Delegation

- Passing method calls to a composed object is called *delegation.*
- We used member pointers rather than objects to permit polymorphism.
- Composition sometimes slightly better than inheritance because using multiple inheritances can sometimes be harder to maintain. In have two inheritances could make one an inheritance and for the other use compostion.
- Compostion allows one to lock in behavior at runtime.
- Still is blackbox code reuse.

Examples of Delegation

- cCritter use of cSprite's, cListeners, and cForce. Note also using polymorphism when subclass.

OO Design Principles

- OOA: An object is an organism-- it should own all the methods it needs to do things
- OOA: Have more than one class.
- OOA: Don’t make your classes do too much.
- OOA: reuse classes
- OOA: prefer composition to class inheritance
- OOD: Think like an object to determine class methods.
- OOD: Use pointer methods rather than instance methods
- OOD: program to an interface not an implementation

More principles

- OOP: Don’t store same thing in two places.
- OOP: Don’t write the same code twice
- OOP: Don’t ask object their runtime type.
- OOP: Don’t break encapsulation.

The Code Interface

- C++ features for interfaces:
  - #define switched for #ifdef's
  - Typedef's for renaming types
  - Static variables
  - Static methods