Wendy Yin
CPS-108 - 6 Apr 2012
Vooga - design proposal

Design challenge: Efficient non-playing sprite creation and organization.

The main problem embedded in the non-playing character issue is that it is such a large topic, and so I'd have to code as open as possible, providing many hooks for new functionality. Developers would want to be able to create many different types of NPCs, with many different properties, without sacrificing performance (ie the game shouldn't become jerky and hard to control just because there's 100 sprites on-screen). Possible implementations would include simple platforms, power-ups, local physics-altering objects (bombs, black holes), projectiles, or even temporarily attachable weapons for the player sprites (the player picks up a box and throws it at another). This general area overlaps with the design problems of efficient collision detection, specific collision reactions, fighter sprite organization (how will the body parts become related to the other sprites? They are organized in a tree structure, and contain aspects of fighter sprites (health) and non-playing sprites (not directly connected to input handler)) and AI (the non-player sprites might have to be extendable to change their properties given the game statistics/conditions, eg the randomly appearing power-ups near a player become stronger as the player becomes weaker).

Possible solutions:

I originally thought a decorator pattern might be part of the solution, as it allows you to add functionality at run-time and without me having to foresee all possible combinations. However, after trying to implement an "attachable" decorator that allows the wrapped object to attach to a fighter sprite, I realized that implementing a decorator doesn't work well with the Golden T framework because there would be a great deal of overhead to deal with (already defined functions within the sprite class that are useless for the decorators to have and could even mess up the organizational aspect of the sprites, ie decorators ending up with different coordinates, speeds, IDs than the wrapped sprite). Additionally, there could be completely new functionality, strange behaviors that would not quite be universal enough to be included in a virtual function in the sprite class but still desireable to have. These can be saved as new sprite subclasses, but once again, there is a great deal of memory usage involved.

It seems that there are two main ways to go:

1. The sprite contains references to the new functionality. The issue with this is when you want to call those functions through the sprite - how does the sprite know to treat those functions inside another class as if it were its own?

pseudo-code:

```
public NonPlayerSprite{
    public void addFunction(FunctionClass c);
    public void doSpecialFunctions(){ for all FunctionClass, FunctionClass.doStuff(); }
}
public FunctionClass{
    private NonPlayerSprite parent;
    public void doStuff() { //stuff; parent.updateWithStuff() }
}
```

If I implement the pseudocode above, this allows for perhaps too much clumping of methods

together - one added functionality may have to do with how the sprite moves sideways, while the other may deal with grouping the sprite with another, and they would be under the same virtual function call. The calls to FunctionClass may be split up among many different function calls, but that may lead to smelly code and the same problem of overhead (subclasses getting methods they never use or want).

2. Keeping the additional functionality in a completely separate wrapping class. Not quite decorator pattern.

```
public NonPlayerSprite{ //stuff, does not have a function defined as "doStuff" }
public FunctionClass extends NonPlayerSprite{
        private NonPlayerSprite parent;
        public void doStuff() { //stuff; parent.updateWithStuff() }
}
public OtherFunctionClass{
        private NonPlayerSprite parent;
        public void doOtherStuff() { //stuff; parent.updateWithStuff() }
}
```

This is what is currently implemented. The issue with this is the declaration, which looks like:

```
NonPlayerSprite p=new NonPlayerSprite();
FunctionClass o=new FunctionClass(p);
OtherFunctionClass n = new OtherFunctionClass(o);
```

This would get rather tedious to write out, and taking out the FunctionClass functionality would mean that the initialization of OtherFunctionClass now needs to change to take the parameter p, not o. This is not flexible and would require a lot of coding on the model-view end.

As for different collision actions according to what the sprite collided with, right now there is a method called collisionAction(int otherGroupID) which all sprites must inherit. Presumably each sprite takes a different action (reduce health, bounce backwards, do nothing) depending on what it collided with (maybe decided though switch statements, it's up to the person who implements specific subclasses of sprites). This is more flexible than the Golden T's collision checker, since every sprite can specify its own behavior despite whichever group it's in, while before each spritegroup was stuck with a defined action versus another spritegroup. This also simplifies grouping, as sprites that are children of other sprites (like body limbs for the fightersprite) can automatically adopt the ID of their parent object, as seen in the method:

```
public void addWeapon(NonPlayerSprite child) {
    myWeapons.add(child);
    child.setID(this.getID());
}
```