Wendy Yin
CPS-108 - 13 Apr 2012
Vooga - design proposal for fighter-style game

My design challenge contains two main parts: the organization of sprites inside the game object, so that less code needs to be written by the developer to group sprites, and more flexible sprite creation (and their reactions to collisions) than golden T's subclassing of rather bloated sprite classes. As an overview, this will be done with sprites that mimick multiple inheritance, though the use of interfaces, and a sprite ID mapping to each unique sprite that can be accessed for filtering sprites. The interfaces will also be used to create the sprite IDs.

A standard use case of the features being implemented would be creating two player "fighter" sprites, with a platform. Inside the game method .initResources, the game reads in the properties of each fighter and the platform from a file and creates them using a LevelObjectsFactory class. Each fighter also accepts collisionEvent objects, which will be produced in another factory and passed into the fighter during creation. The game developer would create this file through the character editing gui, or perhaps writing directly on the xml file, or even explicitly coding the sprites (although this is not advisable):

```
BodySprite torso=new BodySprite(image, xcoord,ycoord);
// create other limbs
FighterBody f=new FighterBody(image, groupID_PLAYER, xcoord,ycoord);
f.addChildren(torso, leg, arm, ...);
PlatformBlock p=new PlatformBlock(image, groupID_PLATFORM, xcoord,ycoord, width, height);
```

Note that all the limbs of the fighter inherit the "groupID". This implications of this will be covered later.

After this automated process of sprite creation, and throwing them all into a big collection of some sort, initResources calls a function called "groupSprites", which takes all the sprites, sorts them according to groupID, and each groupID ends up mapping to a SpriteGroup, a group that contains all the sprites with that particular groupID. This mapping of groupID to SpriteGroup is passed back into the game, which saves it.

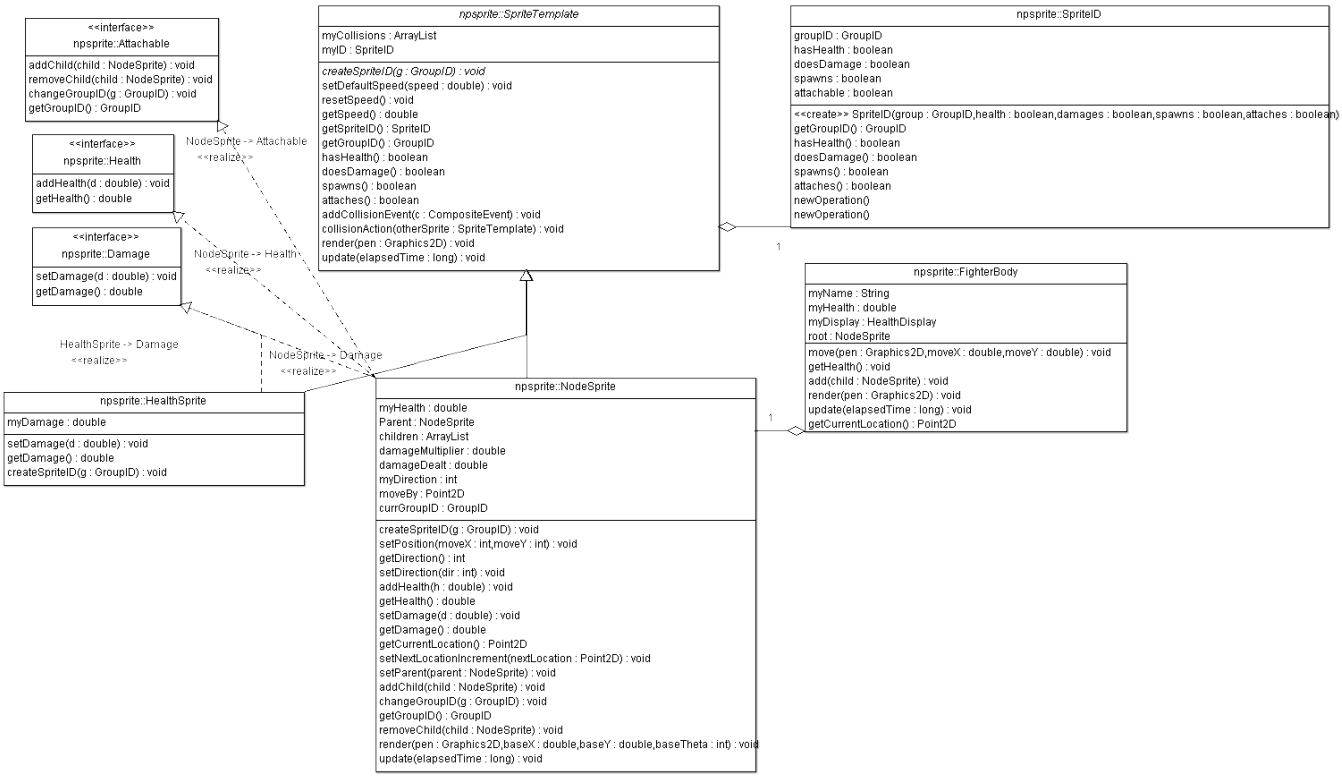
Inside render and update, the code only needs to call "renderAllGroups" and "updateAllGroups" and the physics engine will access the maps of spritegroups and check them for movement and collisions.

In summary, the the standard use case, the developer just has to pick images off a gui to create the standard predefined sprites, specify what happens to that sprite when it hits other sprites (takes away health, bounces other sprite, spawns new sprites, etc), and what the name of the sprite group will be, and the game will load everything from there. On the off chance the developer wants a sprite, say, the platform, to have a health bar, all they would have to do is subclass platform, implement the "health" interface, and add this into the gui.

A difficult case would be a screen filled with sprites (although that is more of an issue for the physics engine/collision detection), or in a specific hypothetical case a new sprite has a timer built into it, where if a player (and only a player, not an AI) runs over the sprite, it disappears, creates a new "black hole" sprite that pulls all players towards itself and prevents them from blocking themselves from attacks/projectiles until the timer runs out, then the sprite disappears

entirely and the players are dropped back onto the ground to resume normal gameplay. This can be implemented with the current design by creating a sprite that implements a Volatile (for sprites that appear and disappear) and Attachable (for sprites that can own other sprites) interface. First, we have to cover the design to understand how this case becomes easier to implement than in golden T.

The design for the sprites is not very different from the basic subclassing found in the base Golden T framework. The difference lies mostly in the concept of multiple inheritance and Strategy pattern.



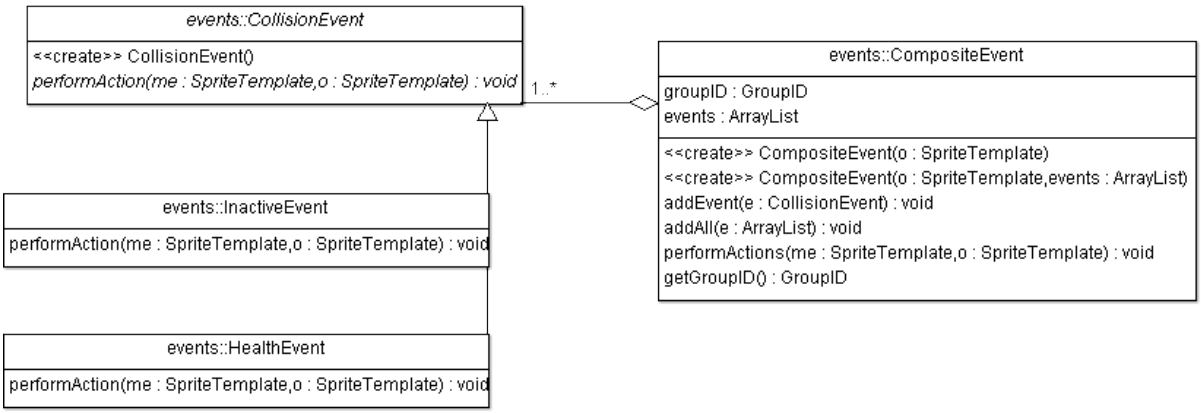
As can be seen in the pared down UML diagram, there is a main SpriteTemplate class which extends Sprite (which is itself a simplified version of the Golden T Sprite, to prevent developers from circumventing the redefined methods and fields in SpriteTemplate - in particular, the original gtge sprite had dataID and ID fields, which are unnecessary given the new SpriteID class). All sprites will extend this class to give basic functionality - movement, rendering, storing a SpriteID, storing and getting collisionEvents. From now on, any reference to a "sprite" assumes a subclass of SpriteTemplate.

There are also multiple interfaces, which specify certain methods (the Health interface add "addHealth" and "getHealth" methods) that their implementations would like to have, along with a boolean flag set to true. Like a decorator, the interfaces allow the sprites to combine multiple behaviors in flexible ways; unlike decorators, this cannot be done at run-time, however given the amount of overhead in creating a decorator-subclass of sprite and the vastly different method names between behaviors, the decorator pattern was not used. During the instantiation of any sprite, the constructor calls the method "createSpriteID", which calls the constructor of the SpriteID class. SpriteID is a data class with only getter methods (which are wrapped by the

spriteTemplate for ease of access) - this was chosen to group all the possible traits a class might have, and want to be filtered by, under one easily accessible class, and prevent any events within a sprite from changing the traits (a fighter sprite cannot suddenly decide it doesn't have a health field, although it can decide what values its health should be - think of the ID as a workaround to using reflection, the flags set to true are passed into the constructor and the program won't have to do any heavy checking other than a call to the spriteID to confirm that a sprite uses a particular interface). This SpriteID class also comes in handy during collisionEvent checking, which will be addressed later. These benefits are balanced against the linkage between interfaces and the spriteID - as more interfaces are added, more fields inside spriteID (or subclasses of spriteID) might be needed if the developer wants to check if a sprite has these properties. Directly calling final fields inside the individual sprite achieves the same purpose, but the same issue comes up of synchronizing the changes inside sprite and interfaces, except more spread out and requiring shotgun surgery - if you add an interface called "isBlue", calling the boolean sprite.isBlue on old sprites makes no sense, and so you'd have to go back to each sprite and add a boolean isBlue that is set to false. This can't be abstracted to the SpriteTemplate.The other areas of the game don't know and don't really care about all the intricate differences between isBlue sprites and not isBlue sprites, and won't keep the subtypes intact while passing through many methods.

The two sprites shown demonstrate a health pack and a fighter (player) sprite. Player sprites are slightly different from other sprites in that they have a "FighterBody", which is essentially a pointer to the top of the tree of sprites that represents the fighter, and allows for the the player, through the input handler, to control the fighter. Each sprite inherits the same groupID (which makes sense, they all belong to the same player). For more information on the layout and recursive function calls, see Helena's paper on fighter sprite animation. What is important for this problem is that other sprites may also want to have a dynamic tree-like structure without being keyboard-controlled, and so they have the option of implementing the Attachable interface.

Now looking at the collisionEvents, which roughly resembles the Actions passed around by the input handler and a Strategy pattern. CompositeEvents are added to sprites during the process of initResources, and map a particular groupID to a particular set of reactions.



For example, a player and a platform collide. The physics engine calls the methods player.collisionAction(platform) and platform.collisionAction(player). Inside collisionAction, the sprite determines what set of actions to do on itself given the other sprite's groupID.

The benefit of having the collision reactions and sprites separated is that different sprites can have different reactions even if they are in the same group, a feature not available in gtge.

Also, collision reactions stored inside the sprite already force some sprite filtering (a sprite with no health ability isn't going to store a decreaseHealthEvent, although currently this certainly could happen with an inept and inattentive developer editing directly from code), so that even more filtering based on other fields in SpriteID (like damages, spawns) is only a short step away. Strategy pattern is a natural fit for this swapping of different algorithms. An issue with this is that collisionEvents may take up too much memory space and slow the program down (most of the memory should be devoted to sprite storage and physics calculations) - this can be avoided by turning the collisionEvents into Singletons with a single static method, and making sure nothing goes bad if two sprites call the event simultaneously.

<Insert stuff about automatic grouping, and how attachable allows for sprites to jump between groups (making for slightly different collision events happening) during the game - very flexible>

Knowing all this design, the case of a black hole sprite looks manageable. A sprite with the interface Volatile (appearing and reappearing), maybe a new interface Spawns (the original sprite spawns the black hole) and Timer, and the black hole sprite interfacing Attachable. When the player collides with the original sprite, it spawns the black hole sprite. The black hole queries the game for all the spritegroups with a groupID referencing "player", and attaches all these groups as children, changing their groupID's to the black hole's, and sets their velocities to point towards the center of the hole. This way, the physics engine no longer checks for collision between the black hole and the players, so they can overlap each other. The black hole can now send all sorts of signals to its player children, such as not allowing them to block, until the timer runs out, the black hole sets itself to inactive, and the players, no longer having a parent sprite, revert back to their old groupIDs and go back to the way they were before.