Wendy Yin
CPS-108 - 20 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 ID mapping to each unique sprite that can be accessed for filtering sprites and maps of "property" and "collision" objects that contain details of statistics and their interactions (since interactions of sprites mainly involve alterations in their statistics).

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, and propertyObject which is similar to collisionEvent except each property is defined by a single field for the statistic (health, damage, etc), a single function that modifies the statistic when being called by a collisionEvent, and various getter methods. 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. The physics/collision engine is able to access all these groups to check for movement and collisions, with the opportunity for detecting collisions between sprites with the same groupID (allowing multiple layers of grouping other than the explicit groupID-enforced one).

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 subclassing NodeSprite (a sprite that is part of the composite pattern in fighter sprites) that has properties TimerObject and Spawns. 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 at the end of this paper, 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 GroupID). All sprites will extend this class to give basic functionality - movement, rendering, storing a GroupID, storing and getting collisionEvents. From now on, any reference to a "sprite" assumes a subclass of SpriteTemplate.

Originally the concept of propertyObjects was instead each new sprite implementing a variety of interfaces that contained the methods specific to the interface, like addHealth, thereby achieving various properties at compile time. The problem with this was that it was created at compile time, so if a developer wanted a sprite with new properties, they had to create a new interface, a new sprite that implemented that interface, and a way of checking each sprite (including the old ones) if they had implemented that interface. The current method of passing propertyObjects into a map allows for changes at run-time, essentially eliminating the need to create new subclasses of sprite, and easier checking of if the sprite had a property (myProperties.contains(property)) through a single method call. This is slightly closer to the original idea of decorators for sprites, as it allows for changing properties at run-time but without decorator's amount of overhead (sprites have a lot of methods, and decorators, being subclasses of sprites, would inherit a lot of unnecessary methods).

The main issue with this propertyObject concept is illustrated below:

```java
@Override
public void performAction(SpriteTemplate me, SpriteTemplate o) {
    if (o.hasProperty(DamageProperty.getName())) {
        ((HealthProperty) me.getProperty(HealthProperty.getName()))
                .addHealth(((DamageProperty) o.getProperty(DamageProperty
                        .getName())).getDamage());
    }
}
```

Checking and extensive casting is still needed, and although in terms of safety this isn't so bad (since each propertyObject has been checked to ensure it is safe to cast to the subclass), it's very messy and suggests the code could be better. A possible solution is the strategy pattern with double dispatch.

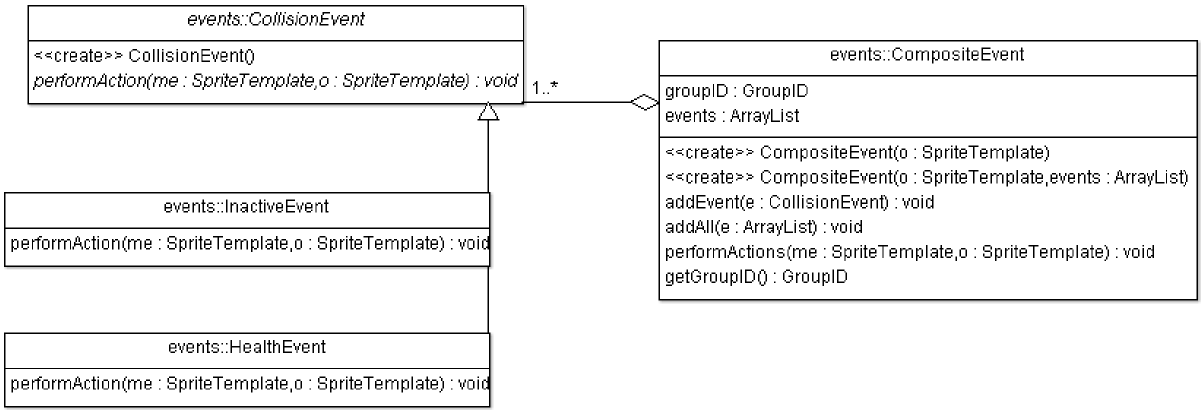
```
//inside HealthEvent
    @Override
    public void performAction(SpriteTemplate me, SpriteTemplate o) {
        o.getProperty("damage").doAction(me.getProperty("health"));
    }

//inside DamageObject
    void doAction(PropertyObject p){
        p.doAction(this);
    }

//inside HealthObject
    void doAction(DamageObject d){
        //do stuff
    }
```

The two sprites shown (in uml at bottom) 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 the actual tree composite structure is isolated in the NodeSprite class.

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.

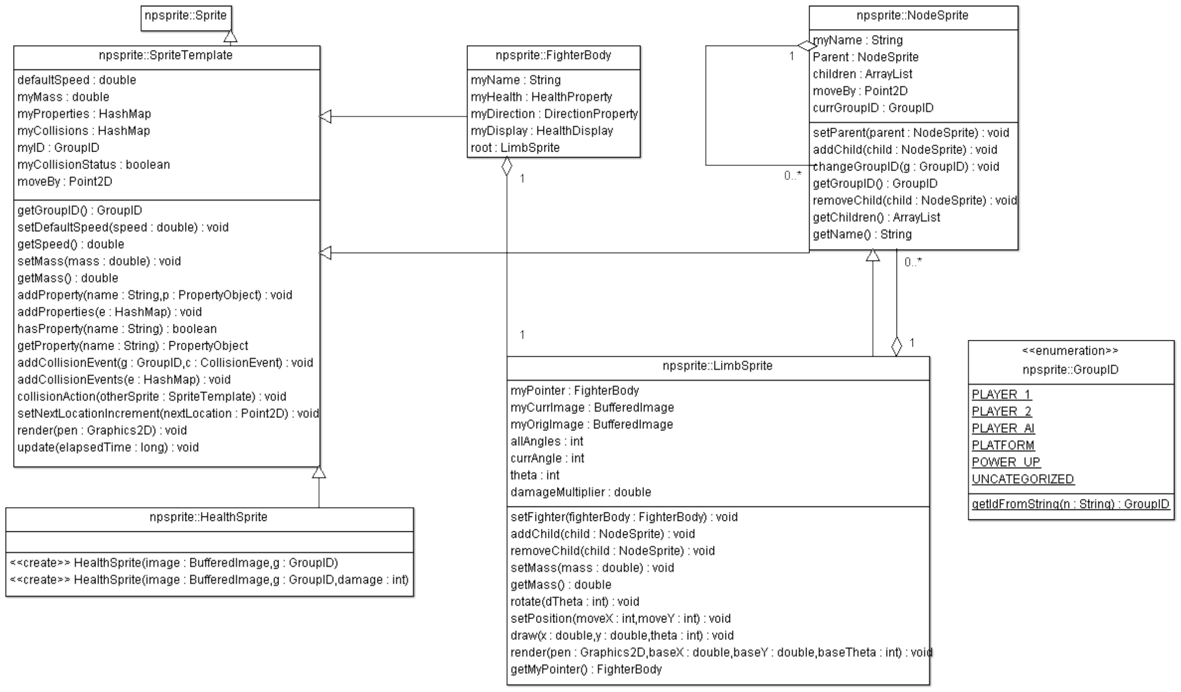


events::CollisionEvent
<<create>> CollisionEvent()
performAction(me : SpriteTemplate,o : SpriteTemplate) : void
1..*
events::CompositeEvent
groupID : GroupID
events : ArrayList
<<create>> CompositeEvent(o : SpriteTemplate)
<<create>> CompositeEvent(o : SpriteTemplate,events : ArrayList)
addEvent(e : CollisionEvent) : void
addAll(e : ArrayList) : void
performActions(me : SpriteTemplate,o : SpriteTemplate) : void
getGroupID() : GroupID
events::InactiveEvent
performAction(me : SpriteTemplate,o : SpriteTemplate) : void
events::HealthEvent
performAction(me : SpriteTemplate,o : SpriteTemplate) : void

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.

Knowing all this design, the case of a black hole sprite looks manageable. A sprite with the properties Spawns (original sprite spawns the black hole) and Timer, and the black hole sprite subclassing NodeSprite. 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.



npsprite::Sprite
npsprite::SpriteTemplate
defaultSpeed : double
myMass : double
myProperties : HashMap
myCollisions : HashMap
myID : GroupID
myCollisionStatus : boolean
moveBy : Point2D
getGroupID() : GroupID
setDefaultSpeed(speed : double) : void
getSpeed() : double
setMass(mass : double) : void
getMass() : double
addProperty(name : String,p : PropertyObject) : void
addProperties(e : HashMap) : void
hasProperty(name : String) : boolean
getProperty(name : String) : PropertyObject
addCollisionEvent(g : GroupID,c : CollisionEvent) : void
addCollisionEvents(e : HashMap) : void
collisionAction(otherSprite : SpriteTemplate) : void
setNextLocationIncrement(nextLocation : Point2D) : void
render(pen : Graphics2D) : void
update(elapsedTime : long) : void
npsprite::HealthSprite
<<create>> HealthSprite(image : BufferedImage,g : GroupID)
<<create>> HealthSprite(image : BufferedImage,g : GroupID,damage : int)
npsprite::FighterBody
myName : String
myHealth : HealthProperty
myDirection : DirectionProperty
myDisplay : HealthDisplay
root : LimbSprite
1
1
npsprite::NodeSprite
myName : String
Parent : NodeSprite
children : ArrayList
moveBy : Point2D
currGroupID : GroupID
setParent(parent : NodeSprite) : void
addChild(child : NodeSprite) : void
changeGroupID(g : GroupID) : void
getGroupID() : GroupID
removeChild(child : NodeSprite) : void
getChildren() : ArrayList
getName() : String
1
0..*
0..*
1
npsprite::LimbSprite
myPointer : FighterBody
myCurrImage : BufferedImage
myOrigImage : BufferedImage
allAngles : int
currAngle : int
theta : int
damageMultiplier : double
setFighter(fighterBody : FighterBody) : void
addChild(child : NodeSprite) : void
removeChild(child : NodeSprite) : void
setMass(mass : double) : void
getMass() : double
rotate(dTheta : int) : void
setPosition(moveX : int,moveY : int) : void
draw(x : double,y : double,theta : int) : void
render(pen : Graphics2D,baseX : double,baseY : double,baseTheta : int) : void
getMyPointer() : FighterBody
<<enumeration>>
npsprite::GroupID
PLAYER_1
PLAYER_2
PLAYER_AI
PLATFORM
POWER_UP
UNCATEGORIZED
getIdFromString(n : String) : GroupID