Wendy Yin
CPS-108 - 27 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 in conjunction with the gui, 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. The rules are that the fighters can each punch each other, causing damage reflected in the health bars, jump on top of platforms (but with gravity acting on the fighters), and cannot walk over each other or the platforms. The bottom of the screen is the floor.

To construct this simple case, 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 and PropertyObjects, which will be produced in another method inside the factory and passed into the fighter during creation. CollisionEvent is a single method wrapped by a class, which takes specific properties of the two sprites passed in, and does actions on the sprite it belongs to. It is called inside the method collisionAction inside the sprite, which in turn is called by the physics engine when it detects a collision. PropertyObject is similar to collisionEvent except each property is defined by a single field for the statistic (health, damage, etc), at least one 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):

```
NodeSprite torso=new NodeSprite("torso",image, groupID.PLAYER_1,xcoord,ycoord,damage);
// create other limbs, attach them to the torso
//attach torso to the fighter body
FighterBody f=new FighterBody(torso, "player 1", groupID.PLAYER_1, HealthDisplay);

PlatformBlock p=new PlatformBlock(image, groupID_PLATFORM);
p.setLocation(xcoord,ycoord);
```

Note that all the limbs of the fighter inherit the "groupID". This is so the physics engine allows limbs to overlap each other. Also, FighterBody has no image, and thus getWidth and getHeight from the Sprite class must be overridden to return the sum of the width and height of limbs.

After this automated process of sprite creation, the sprites are returned in arraylists of sprites. The game saves all the arraylists in the SpriteGroupTemplate class (an arraylist of arraylists) and passes the SpriteGroupTemplate into the physics/collision engine, which 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 but in different arraylists (allowing multiple

layers of grouping other than the explicit groupID-enforced one).

In summary, in 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/ID of the sprite will be, and the game will load everything from there.

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 button sprite appears in a random location, and if a player (and only a player, not an AI) runs over the button, 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 SpawnsObject, and collisionEvents. First, we have to cover the design to understand how this case becomes easier to implement than in golden T.

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 slightly 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 (which were needed once spriteGroups and input/physics/collision engines were changed), storing a GroupID, storing and getting collisionEvents, storing and getting propertyObjects. 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(int h), 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 LevelObjectsFactory class would also have to be changed, to check for a specific tag to create that subclass by name, perhaps through factories for sprites. 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). Unfortunately, the issue of changing LevelObjectsFactory to account for new tags still remains, but seems unavoidable. The benefit of isolating properties and collisions, though, is that the constructors end up looking much more similar to each other, and can be refactored into a more generalized constructor than the widely varying sprites that may have occurred with concrete implementations of interfaces. Additionally, properties can have their own effects, such as the TimerObject which uses the update loop of the sprite to release a new collisionEvent into the sprite (thus changing its behavior during the game) after a certain amount of time.

The main issue with this propertyObject concept is illustrated below, in an excerpt from the collisionEvent:

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

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, hard to read, and suggests the code could be better. The other issue is that the collisions and properties are dissociated, so it is entirely possible that someone passes in a property that is never invoked in collisions, or a collision that affects properties in bizarre illogical ways or trying to access properties that the sprite doesn't have. A possible solution is double dispatch.

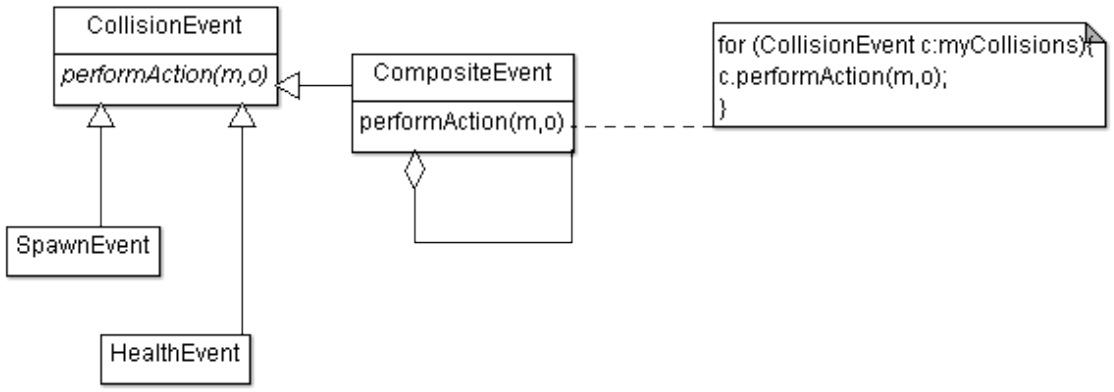
```
//inside HealthEvent
    @Override
    public void performAction(SpriteTemplate me, SpriteTemplate o) {
        o.getProperty("damage").doAction(me.getProperty("health"));
    }

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

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

This was not implemented in this version of the code, because double dispatch will make the code inside propertyObjects very repetitive and crowded when the number of properties are scaled up by developers. Additionally, not all properties may have the need for a doAction method - for example, the direction property is used more as a data class for fighters, one that is only changed by the input handler. Lastly, having the static final field NAME in each property (and collisionEvent) may not be the best design, but requiring its use provides another reminder to the developer to include that field, for easier construction of the game files by the characterEditor gui which needs standardized tags that stay consistent across creation and loading. Still, double dispatch is a good alternative design, as it enforces a certain connection between properties and collisions without combining the two (it is advisable to keep the properties and collisions separated because they operate at different levels of abstraction - collisions have access to the entirety of both sprites, and can include events that don't involve changes in intrinsic properties, properties can only return things to the containing object but are not limited to being called by the physics engine).

The uml of sprites demonstrates 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 composite layout and recursive function calls, see Helena's paper on fighter sprite animation. What is important for this problem is that this area of the code is a combination of three different projects and is not as flexible as the rest of the code. Most sprites can be created by merely creating a SpriteTemplate and passing in properties/collisions as needed.

Now look at the collisionEvents, which roughly resembles the Actions passed around by the input handler and a Strategy pattern. CollisionEvents are added to sprites during the process of .addCollisionEvent(), and map a particular groupID to a particular set of reactions.



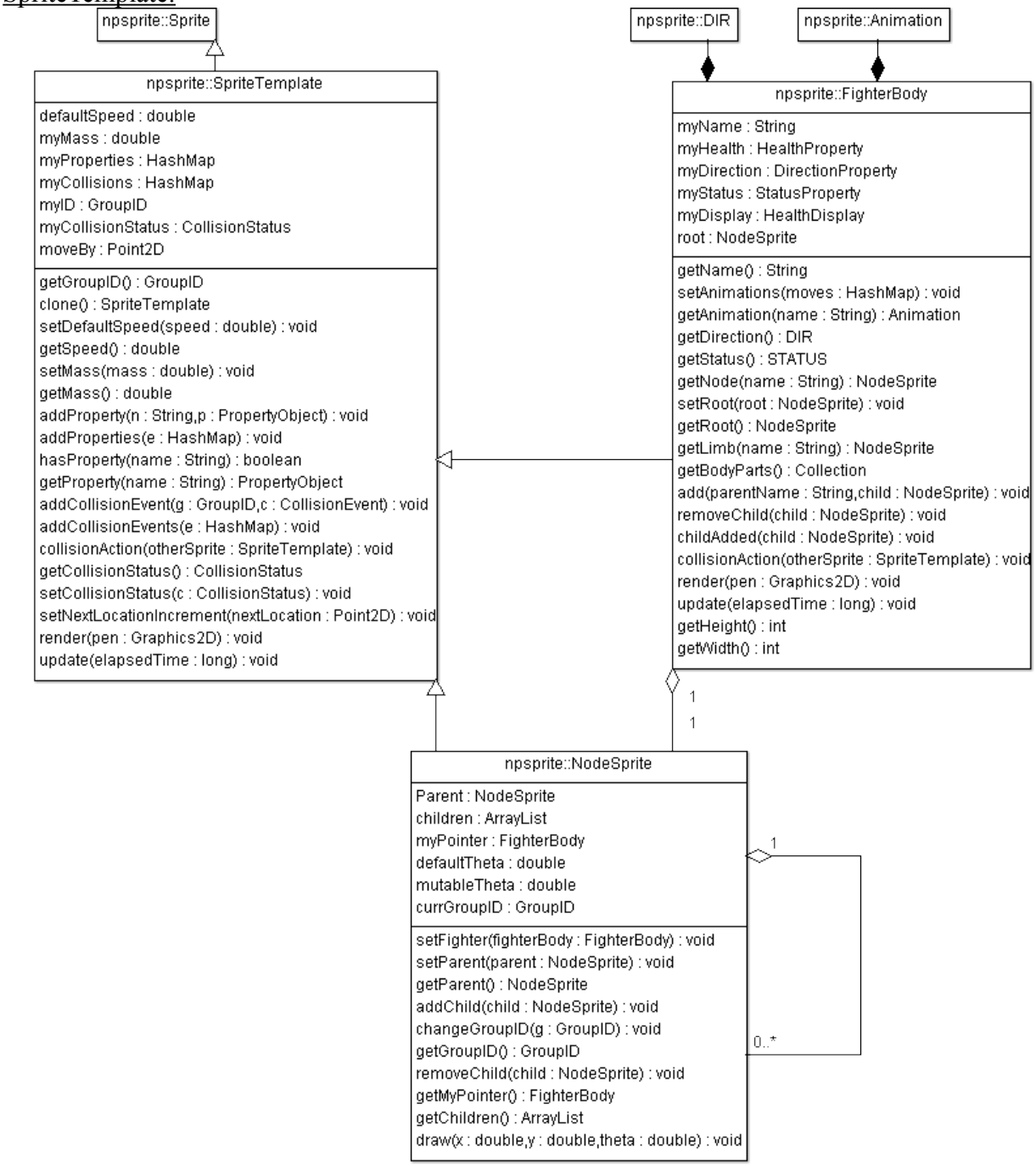
For example, a player and a platform collide. The physics engine calls the methods player.collisionAction(platform) and platform.collisionAction(player). Inside collisionAction, the sprite determines what set of actions to do on itself given the other sprite's groupID by doing a hashmap lookup. CollisionAction also includes a composite pattern, literally called CompositeEvent.

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). 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 was avoided by turning the collisionEvents into Singletons with a single getInstanceOf() static method, and the performAction method taking in two sprite instances.

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, which is saved as an inactive template inside the property object), and the black hole sprite subclassing NodeSprite, with a Timer property. The black hole's Timer property returns an InactiveEvent (inactivates the black hole). When the player collides with the original sprite, it spawns a set number of black hole sprites at set locations. The black hole queries the groupID class for all the sprites with a groupID referencing "player", and attaches all these nodeSprites as children (all player sprites must be nodeSprites), 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 developer may have to write a function that checks for that collisionKind, for more info reference the physics engine proposal by Donghe). The black hole can now send all sorts of recursive signals to its player children, such as not allowing them to block, until the timer runs out, the timer returns the event that tells the black hole to set itself to inactive and to drop its reference to the torso nodeSprite, and the players, no longer having a parent sprite, revert back to their old groupIDs and go back to the way they were before, having a FighterBody as the ultimate parent node.

SpriteTemplate:



Sprite properties: