In my design, I try to make the classes as general as possible. Like Tank and MyPlane are both ControllableWeapon, meaning they can be controlled by the players. ControllableWeapon handles KeyboradEvent and CollisionEvent. So the only differences Tank and MyPlane have are the different sprites, different Munitions, different enemy, different effect if being hit and so on. In this sense, in any combating game project, ControllableWeapon can be reused to create objects controlled by players. I try to accomplish this in all aspects of my design. Some minor details I didn’t cover, like different kinds of Munitions, Rockets and BouncingBombs. But they can be easily created by extending Munitions.

Below are a table of class diagram and detailed explanation.

|  |
| --- |
| Superclass |
| Subclass |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| BasicObject | | | | | |
| Island | Wall | Munitions | | Weapon | |
|  | TemporaryWall | EnemyBullet  EnemyAim | MyMunitions | EnemyShoot  EnemyTarget  EndBoss | ControllableWeapon |
|  |  |  | MyBullet Shell  Rocket  BouncingBomb |  | MyPlane  Tank |

|  |
| --- |
| GameEvent |
| CollisionEvent, KeyboardEvent, CreationEvent, DestructionEvent |

Other classes:

Controller, KeyboardDetector, CollisionDetector

BasicObject

is an abstract class, which is the superclass for almost all of the classes that I am going to create for this project. It can also be reused in other projects. It represents the most general functionality of objects in game, like position, image, health and so on. collide() is an abstract method for reacting to collision.

|  |
| --- |
| *BasicObject* implements Observer |
| x : int // x position  y: int // y position  speed: int  direction: int // direction controls the change of x and y  health: int // if the object are permanent, their health are always positive  image: Image  explosion: Explosion  music: Music  CreationEvent  DestructionEvent |
| BasicObject(including Image image) // image is passed in by the controlled  draw()  update() // check whether any events happened.  *collide*(BasicObject obj) // private method, only called by update()  intersects(BasicObject obj): boolean // check whether two sprites intersect  *explode()* // called by destroy()  destroy() // notify the controller  accessor and mutator method  increase and decrease health methods |

The constructor and destructor needs to change the data of CreationEvent and DestructionEvent inorder to notify the Controller about the creation and destruction of the instance. (I plan to make these two static data field) Since all the other visible objects in the game are derived from this class, their creation and destruction also notify the Controller. This enables the Controller to use CollisionDetecter to detect collision and notify the observers.

update() checks for events. If CollisionEvent, call collide().

collide() is called when there is a collsion. If the object is free of collision, like Island, just leave the method body empty. It first checks which object it collides with and react correspondingly.

intersects() is called by CollisionEvent to check whether the two objects collide. To check whether two objects intersect, check whether the rectangle of their image intersects. It can be implemented by using Rectangle.intersect(Rectangle r).

If the object has no health issue, like Island, just leave explode() empty.

In my implementation, I separate detecting collision from handling collision. CollisionDetector is responsible for detecting collision and changing data of the CollisionEvent(extends Observable). The two objects where collision happens will be notified. Any BaiscObject in the game shouldn’t and don’t know the existence of other objects. So they should not worry about checking collision.

Subclasses:

Island, Wall, Munitions, Weapon

**Island**

No direction and speed change. Free of collision and explosion.

If move out of window, meaning collide with the window (CollisionEvent), change its position and reappear again.

**Wall**

Still object. Explosion free.

Subclass: TemporaryWall

**TemporaryWall**

is a Wall that can be temporarily removed and will rebuilt afterwards. When the war is removed, it is not destroyed. Just its position is changed so the player cannot see. Because it needs to be rebuilt, its start position needs to be remembered. It has two timers. One controls how long to disappear; Another one controls if the start position is occupied, how long to wait (this should be checking in a loop).

|  |
| --- |
| TemporaryWall |
| int disappear // position which outside the window  disappearTimer  WaitOccupyTimer |
| disappear()  appear() // loop calling wait() if the start position is occupied  wait()  collide() // if collide with Munitions, disappear() and then appear(). |

**Munitions**

class represents anything that can be used in combat that includes bullets, shells, bombs and rockets. Because Munitions is active to attack, so it has the explosion and sound. Munitions will explode as long as collision happens.

|  |
| --- |
| Munitions extends BasicObject |
| explosion  sound |
| update() // override. check collision  collide() // override. If collision, destroy(). |

If collide() with owner’s enemy, call owner.hitEnemy().

Subclasses:

In Wingman –EnemyBullet, EnemyAim

**EnemyBullet**

Owner – EnemyShoot

If EnemyBullet hits Plane, since EnemyPlane does not care about its score, the hit will not change anything about the EnemyPlane.

**EnemyAim**

Owner - EnemyTarget

|  |
| --- |
| EnemyAim extends Munitions |
| BasicObject enemy |
| aim() // aim() adjusts the direction |

**MyMunitions**

represents Munitions that are fired by ControlledWeapon. MyMunitions needs to know its owner. When MyMunitions collide with any objects, it calls owner.isEnemy(). If so, the owner’s score should be increased.

|  |
| --- |
| MyMunitions extends Munitions |
| ControlledWeapon owner |

Subclasses:

MyBullet, Shell, Rocket, BouncingBomb

Can add more subclasses

**MyBullet**

Owner – MyPlane

**Shell** – very similar to MyBullet

Owner – Tank

**Weapon**

represents objects that can damage other objects by firing munitions. In this project, tank and plane are ControlledWeapon because players control when to fire; while EnemyPlane and EndBoss control when to fire by themselves.

Weapon needs to be able to create Munitions and Weapon. It also needs to know the storage and usage of Munitions. Weapon may have different kinds of munitions, and even other Weapon. EnemyPlane can fire EnemyBullet, while EndBoss can fire EnemyPlane. Weapon also needs to know who are the enemy so that when its hit enemy, its score will be increased.

|  |
| --- |
| Weapon extends BasicObject |
| ArrayList<MunitionUsage> munitionList  ArrayList<String> EnemyType |
| update() // override. Check whether CollisionEvent  -fire() // create Munitions  remove() MunitionUsage from munitionList  collide() // override |

InnerClass **MunitionUsage**

stores the storage, frequency of usage relates to one object. Timer controls the frequency of using munitions.

|  |
| --- |
| MunitionUsage |
| BasicObject  int storage  timer |
| Increase and decrease storage methods  Getter and setters |

Subclasses:  
Plane, EnemyShoot, EnemyTarget, EndBoss

**EnemyShoot**

Is able to fire EnemyBullet. Once it hits with MyBullet or MyPlane, it explode().

|  |
| --- |
| EnemyShoot extends Weapon |
| enemyType – MyPlane |

Change direction of EnemyShoot enables creating enemy plane coming from different directions.

**EnemyTarget**

is a subclass of EnemyShoot. It can fire EnemyAim.

**EndBoss**

Is a subclass of EnemyShoot. It is very powerful since it can destroy MyPlane

|  |
| --- |
| EndBoss extends Weapon |
| munitionsList includes EnemyBullet and EnemyTarget |

**ControllableWeapon**

represents the Weapon controlled by players. In this project, MyPlane and Tank are ControllableWeapon. Players can control its movement and ask to fire. Here I put fire(), move() and rotate() methods private since they are only called by updata() inside the class.

|  |
| --- |
| ControllableWeapon extends Weapon |
| int score  int fireKey  int forwardKey  int backwardKey  int rotateClockwiseKey  int rotateCounterClockwiseKey  boolean isRotatable // shows whether the direction can change. |
| update() // call updateMovement()  -fire()  add() and remove() of munitionList  -updateMovement() // check GameEvent and call corresponding move and rotate methods  -moveForward()  -moveBackward()  -rotateClockwise()  -rotatecounterClockwise()  isEnemy(BaiscObject)  hitEnemy()  increase() and decrease() of score |

Subclasses:

MyPlane, Tank

**MyPlane**

is controlled by player through keyboard, like move and fire bullet. MyPlane may collide with other planes and bullets. If it collide(EnemyBullet/EnemyShoot), its health decreases; if it collide(EndBoss), it explode(). It can fire MyBullet periodically. If its bullet hit enemy, its score increases.

|  |
| --- |
| MyPlane extends ControllableWeapon |
| fire () // override to fire MyBullet |

Instances: MyPlane1 and MyPlane2

**Tank**

can fire shell, rocket and bouncingbomb. Tank also have a special weapon – shield. If tank hits Wall, its speed reverses. Tank can rotate to change directions.

|  |
| --- |
| Tank extends ControllableWeapon |
| munitionList: shell, rocket, bouncingbomb  int shield  shieldTimer // shield can only exist for a certain time |
| fire() // override to fire shell, rocket and bouncingbomb |

**CollisionDetector**

is responsible to check whether any collision happens. It is created by Controller and has access to the ArrayList which stores all the existing instances. Note that any newly created and destroyed instances are reflected here.

|  |
| --- |
| CollisionDetector |
| check(BasicObject o1, BasicObject o2) |

The comparing mechanism is to call the comparing object’s intersects(). If it returns true, set the data of CollisionEvent. The CollisionEvent will notify the proper observers.

**KeyboardDetector**

is a subclass of KeyAdapter. It is used to listen to key events.

|  |
| --- |
| KeyboardDetector extends KeyAdapter |
| keyPressed(Keyevent 3) // call setKey() in KeyboardEvent |

KeyboardDetector as a KeyAdapter class is created and added to the controller.

**GameEvent**

is the superclass of all the game events that I will create. It extends Observable.

Here the concept is only creating one GameEvent for each kind of GameEvent. So in this homework, the Controller needs to create one CollisionEvent and one KeyboardEvent. These two GameEvent need to add proper observers according to the situation. Like KeyboardEvent needs to add Plane and Tank as observers, while CollisionEvent needs to add multiple observers. setValue() is called when a GameEvent happens and notify observers.

|  |
| --- |
| *GameEvent* extends Observable *//abstract class* |
| *setValue() //abstract method* |

setValue() needs to be implemented by subclasses.

**CreationEvent**

is created to enable every time a BasicObject created, its data field obj is reset and its observer – Controller will be notified.

|  |
| --- |
| CreationEvent extends GameEvent |
| BasicObject obj |
| setValue() |

**DestructionEvent**

is created to enable every time a BasicObject destroyed, its data field obj is reset and its observer – Controller will be notified.

|  |
| --- |
| DestructionEvent extends GameEvent |
| BasicObject obj |
| setValue() |

**CollisionEvent**

is a subclass of GameEvent. It will notify the observers for collision event happening.

|  |
| --- |
| CollisionEvent extends GameEvent |
| BasicObject o1  BasicObject o2 |
| CollisionEvent()  setValue(BasicObject o1, BasicObject o2) |

setValue() is called in CollisionDetector to set the collision objects to the CollisionEvent.

**KeyboardEvent**

is an Observable of key event. It will notify the observers for keyboard event happening.

|  |
| --- |
| KeyboardEvent extends GameEvent |
| String key |
| setValue() // set key. It is called by KeyboardDetector  getKey() |

**Controller**

is the core class of each game project. It sets up and coordinates the interaction between different objects. Init() sets the starting point of the game. It draws the background. Every time a BasicObject is created and destroyed, Controller changes exisingObjects. Controller knows the resources the game has, like sprites, explosion and sound. It has several hashmap to map the resource with corresponding class.

|  |
| --- |
| Controller extends JApplet implements Observer |
| ArrayList<BasicObject> existingObjects  HashMap<objectType, spriteName>  CreationEvent  DestructionEvent  CollisionEvent  KeyboardEvent  CollisionDetector  KeyDetector |
| addObject() // if new object created, add it to the arraylist  removeObject() // is an existing object destroyed, remove it from the arraylist  update() // call add and remove  getSprite() // map spriteName according to the class type and pass it to create the object  init() |

Subclasses: PlaneController, TankController