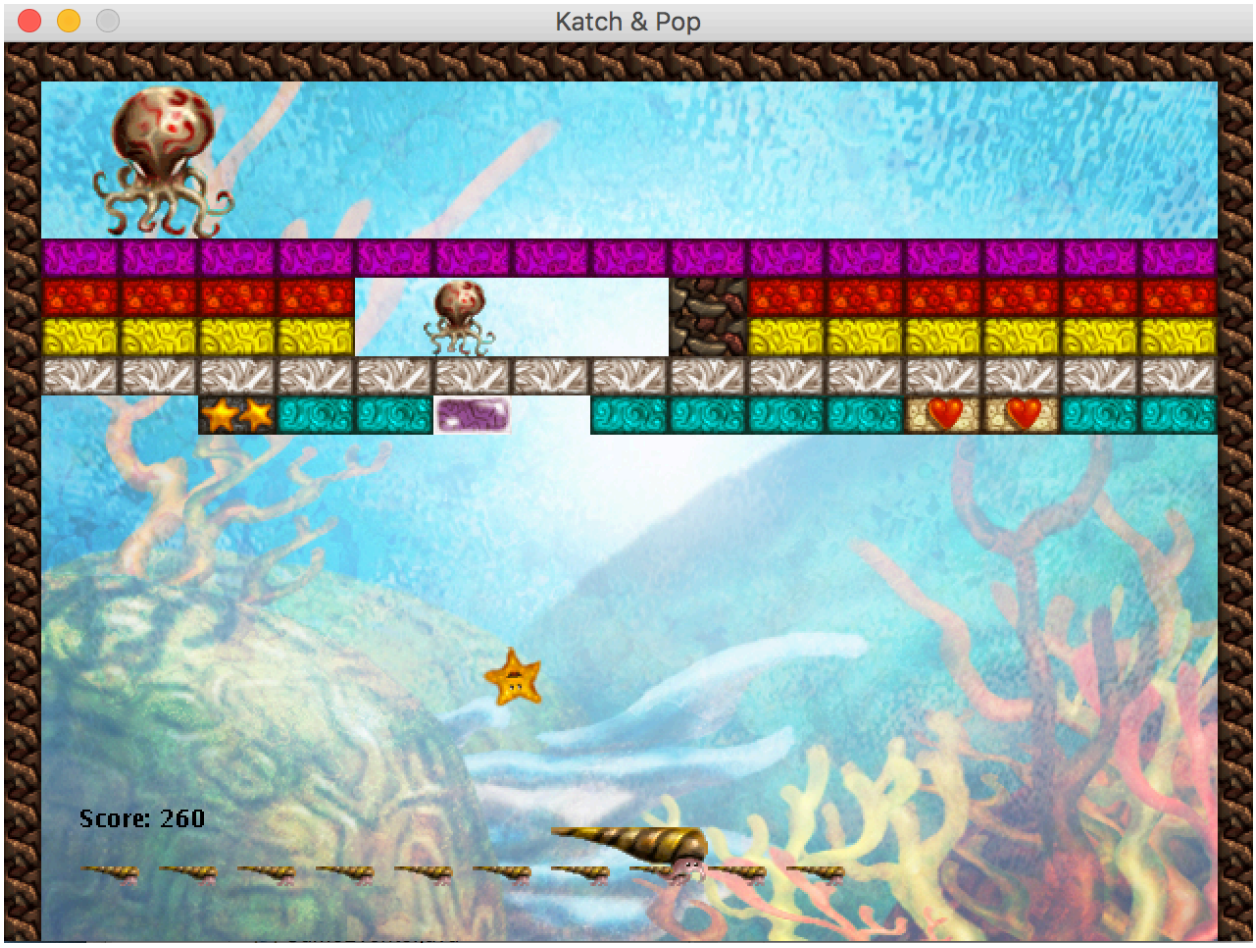
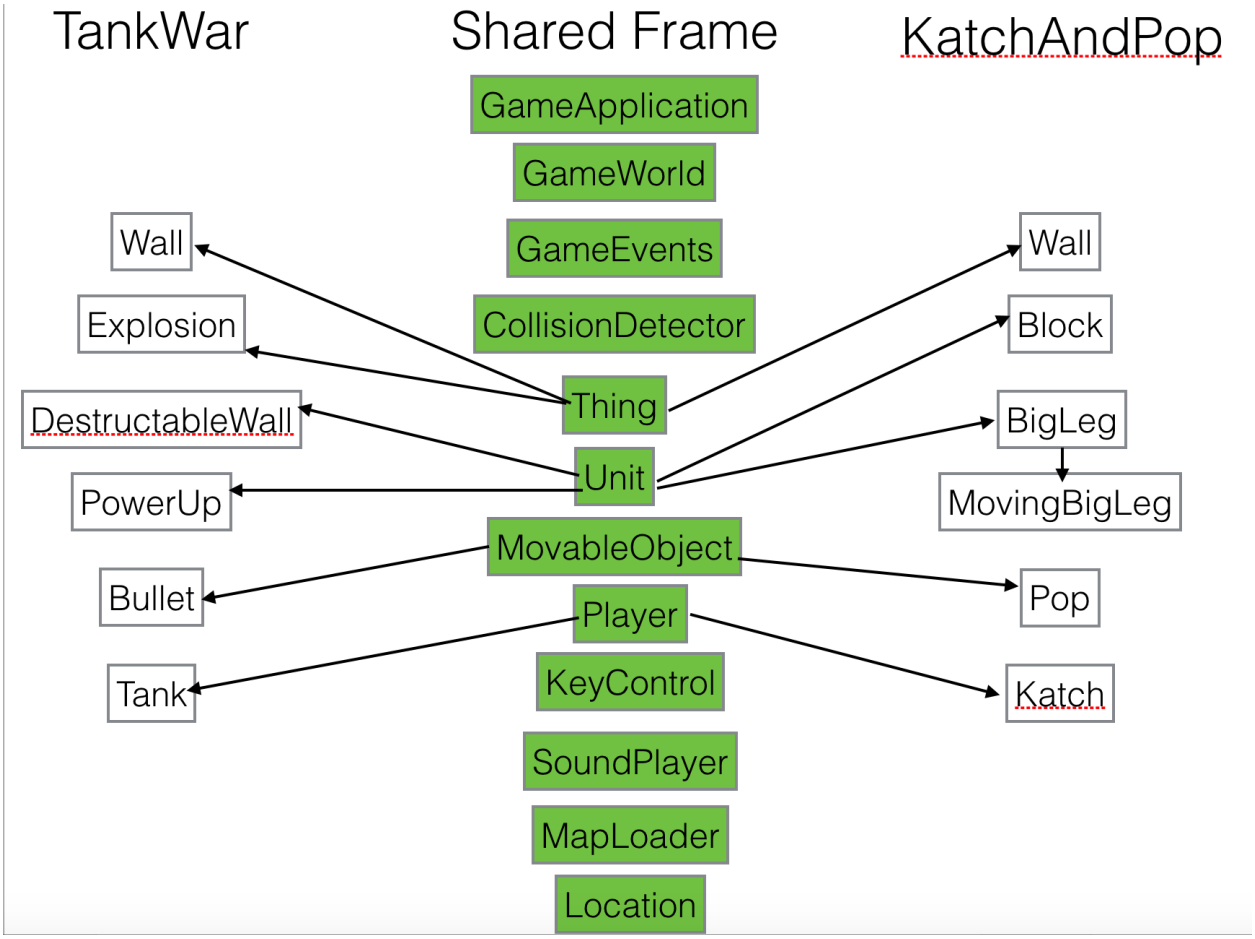
Katch & Pop



The Structure of the two game:



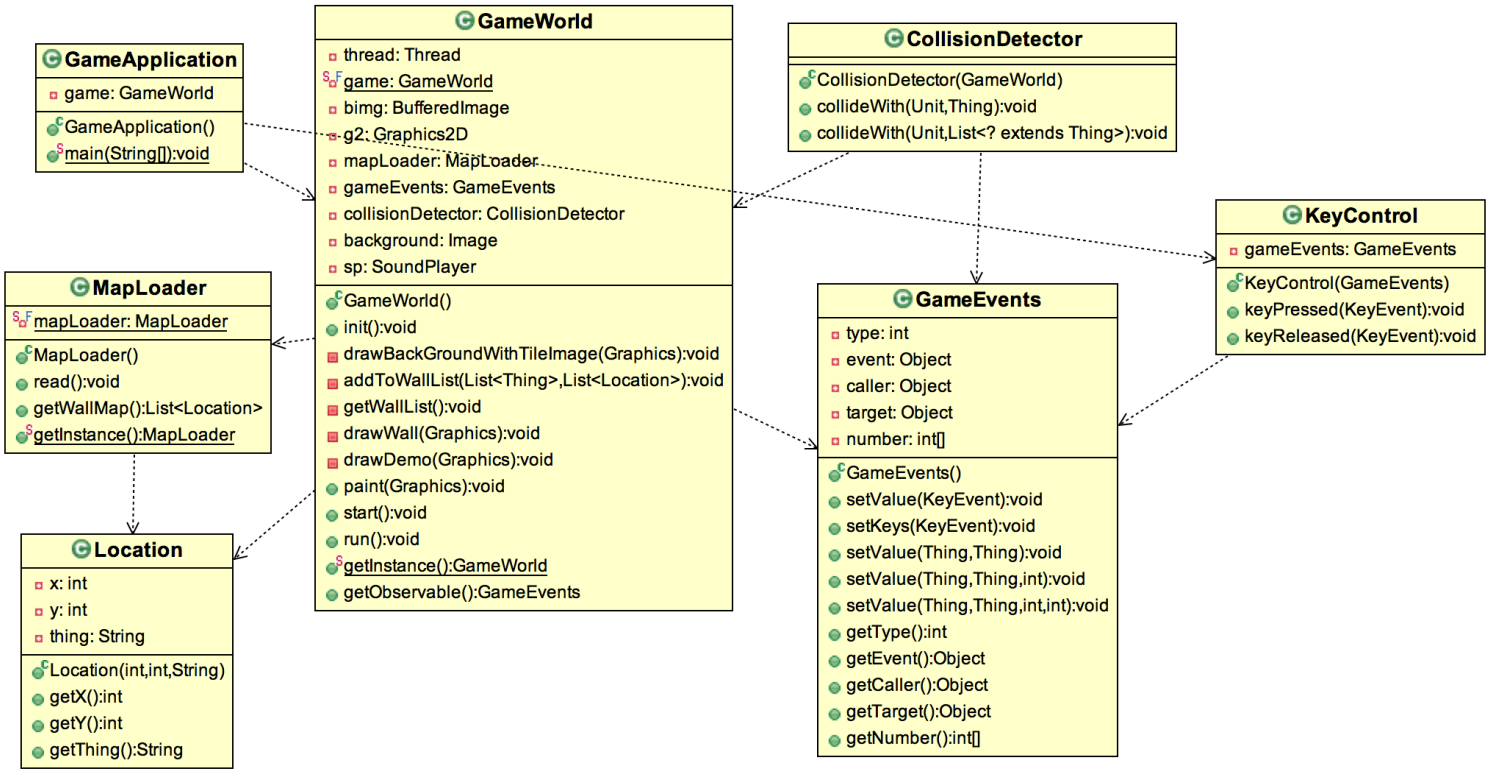
In the middle is the main part of the both games.

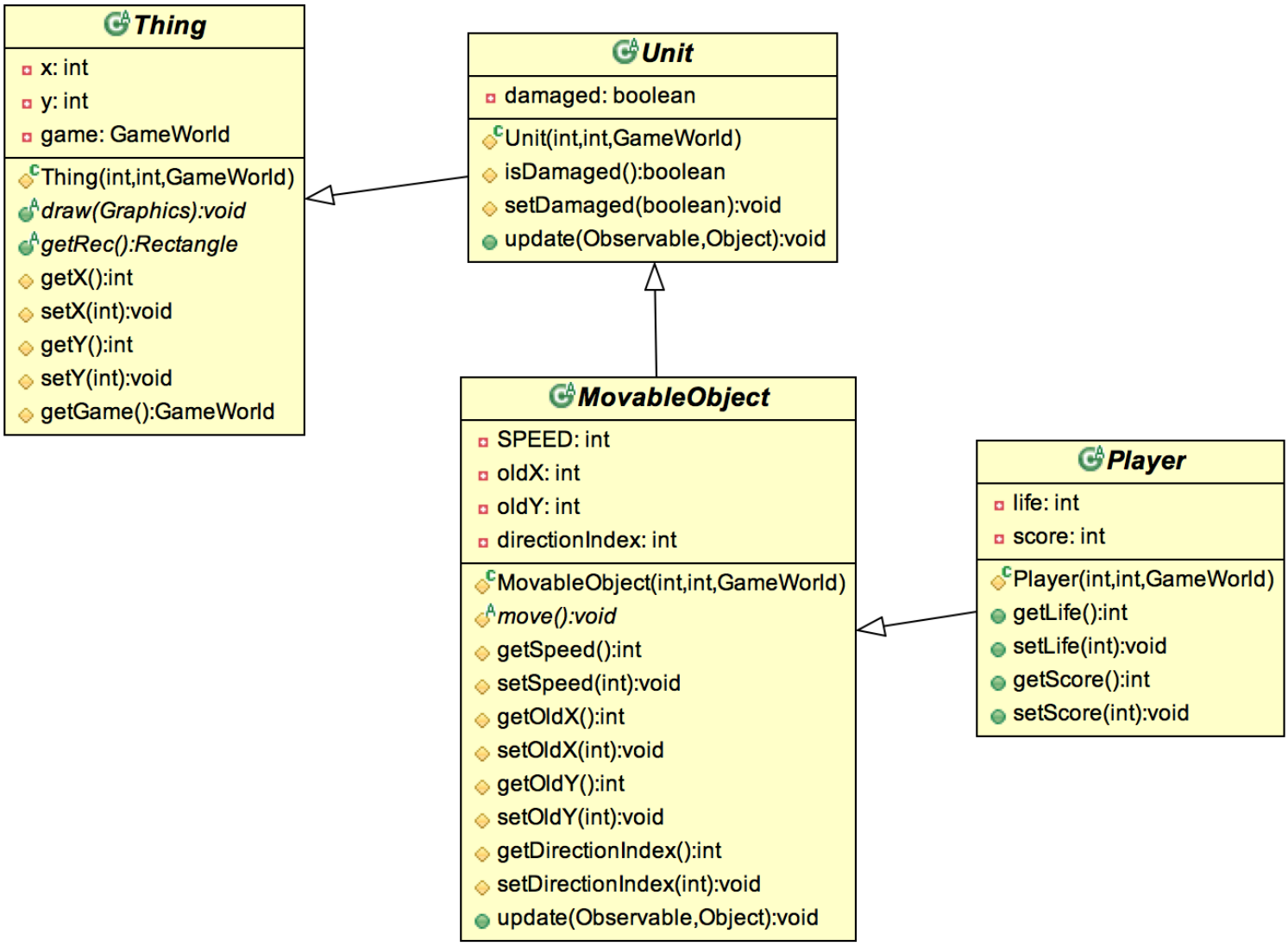
The shared part could be divided into 3 categories. The first 4 (GameApplication class, GameWorld class, GameEvents class and CollisionDetector class) are the main engine for the two games.

Thing, Unit, MovableObject and Player are the inherited abstract classes used by the two games.

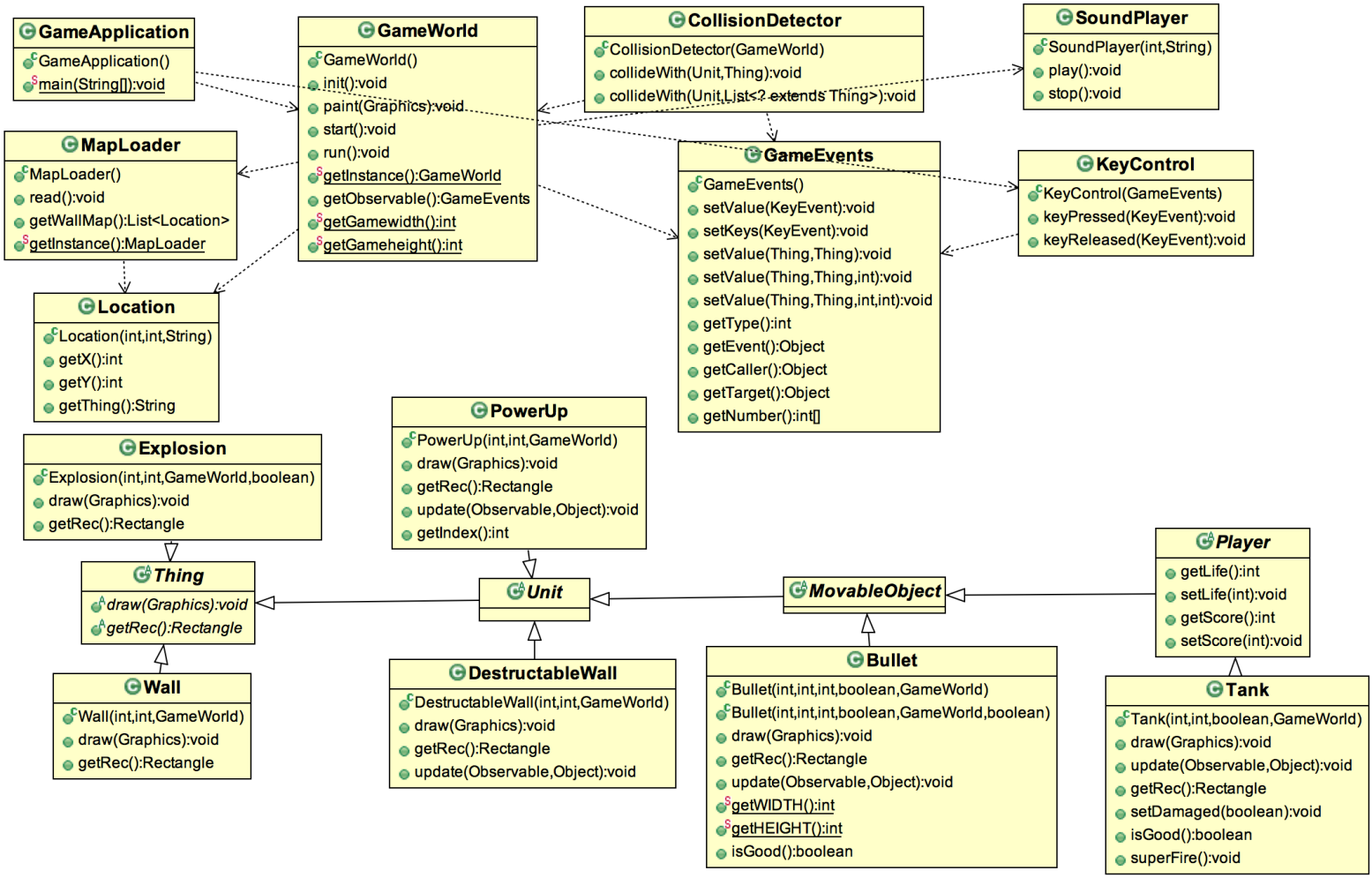
KeyControl, SoundPlayer, MapLoader and Location are the helper classes for both games.

Shared Class Diagram:

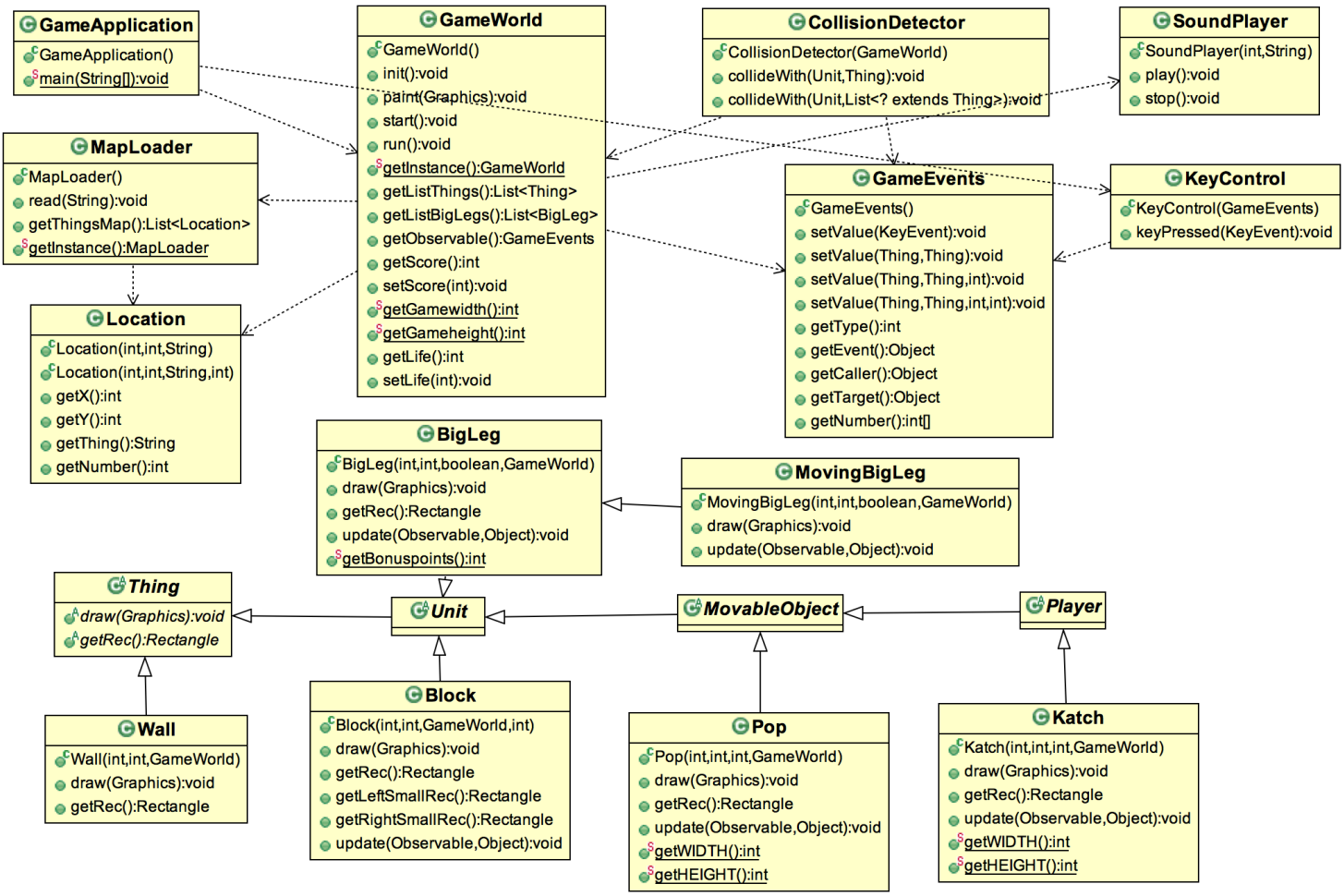



Thing
x: int
y: int
game: GameWorld
Thing(int,int,GameWorld)
draw(Graphics):void
getRec():Rectangle
getX():int
setX(int):void
getY():int
setY(int):void
getGame():GameWorld
Unit
damaged: boolean
Unit(int,int,GameWorld)
isDamaged():boolean
setDamaged(boolean):void
update(Observable,Object):void
MovableObject
SPEED: int
oldX: int
oldY: int
directionIndex: int
MovableObject(int,int,GameWorld)
move():void
getSpeed():int
setSpeed(int):void
getOldX():int
setOldX(int):void
getOldY():int
setOldY(int):void
getDirectionIndex():int
setDirectionIndex(int):void
update(Observable,Object):void
Player
life: int
score: int
Player(int,int,GameWorld)
getLife():int
setLife(int):void
getScore():int
setScore(int):void

Tank War Class Diagram:



Rainbow Reef Class Diagram:



In the shared part, GameApplication is the class which contains the main function.

GameWorld is the key part of the the both games. It extends JPannel and implements Runnable interface.

GameEvents is the observable class.

CollictionDetector is the one to detect if there is a collision. If there is a collision, it will pass all the information to the gameEvents to enable gameEvents to notify all of the observers.

Thing is the base abstract class, it could be represent stable things and those that don't need to update after a collision, such as wall in both games.

Unit extends Thing, and implements Observer interface. It need to update after a collision, such as destructible wall in the TankWar or block in the Rainbow Reef.

MovableObject extends Unit, it could move, such as Pop extends it.

Player extends MovableObject, which includes additional data fields.

Implementation Details:

1. Using Collection Detector

In this project, I use Collection Detector to help me separate different classes. Observer Pattern could reduce coupling to minimize the effect of change.

For example, some day if I modify the Katch class or even delete that class, pop class would not be changed any more.

CollisionDetector

```
public void collideWith(Unit caller, List<? extends Thing> things) {

}

public void collideWith(Unit caller, Thing target) {
    if (caller.getRec().intersects(target.getRec())) {
        …
        gameEvents.setValue(caller, target, katch.getX(), …);
    }
}
```

Observer

```
@Override
public void update(Observable obj, Object arg) {
    GameEvents ge = (GameEvents) arg;
     if (2 == ge.getType() && this == ge.getTarget()) {
        int number = ge.getNumber();  // get the information from gameEvents,
                                      // not interact with other classes
                …
        super.setDamaged(true);  // do it by itself
         …
     }
}
```

2. Polymorphism

use collideWith(Thing, thing), not collideWithTank()

3. override and overload:

collideWith(List<Thing> things) and collideWith(Thing thing)

4. Iterator:

If use iterator in one place, don't use index method to modify the same collection in another place. Otherwise, it would show up ConcurrrentModificationException

5. A bullet shoot 2 blocks:

Need to test if damaged in the collision detector

6. Jar file did not work:

How to get the path name, how to read from a URL, pay attention to the upper case and lower case, e.g: Bigleg_strip24.png vs BigLeg_strip24.png

7. Control the speed:

Should control the speed of the bullet in TankWar and that of pop in Rainbow Reef. Otherwise, it may show some bugs.

8. Inheritance:

parent class's static field / method would be modified if I change the subclass.

9. Downcast:

```
if (bigLeg instance MovingBigLeg) {
        MovingBigLeg movingBigLeg = (MovingBigLeg) bigLeg;
        …
}
```

10. Inheritance @ Override

```
        public void draw(Graphics g) {
                super.draw(g);
                update();
        }
```

11. Iterator for loop:

```
Iterator<BigLeg> iterator = bigLegs.iterator();
while(iterator.hasNext()) {
        BigLeg bigLeg = iterator.next();
        if (bigLeg.isDamaged()) {
                iterator.remove();
        } else {
                …
        }
```

12. add StartPannel:

use JPannel's method: paint(g) or paintComponent(g)
and add a component:

```
        myPanel.add(button);
        button.setBounds(x, y, width, height);
```

13. KeyPressed is not working after adding ActionListener to JButton.

How to solve:

button.setFocusable(false);

(keyAdaptor has lost focus)

14. Block does not appear:

Need to clear MapLoader

15. inner class:

call the owner class' parent class' method: Tank.super.getX()

16. BufferedReader read from URL

17. ImageIO.read(new File(…))

ImageIO.read(GameWorld.class.getResource(…));