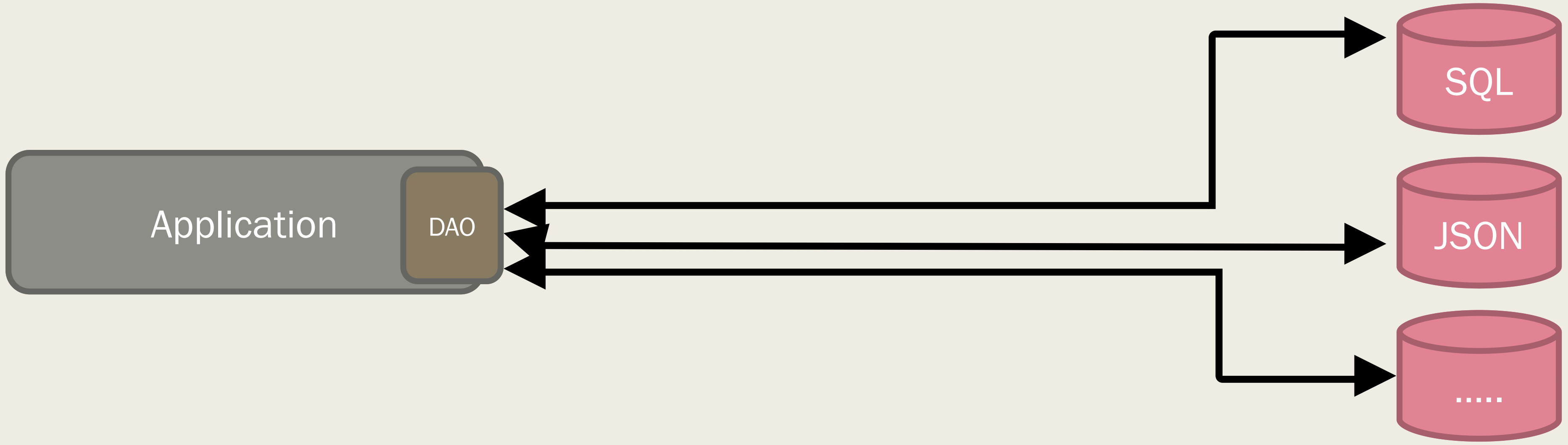
DESIGN PATTERNS

Piranha Team

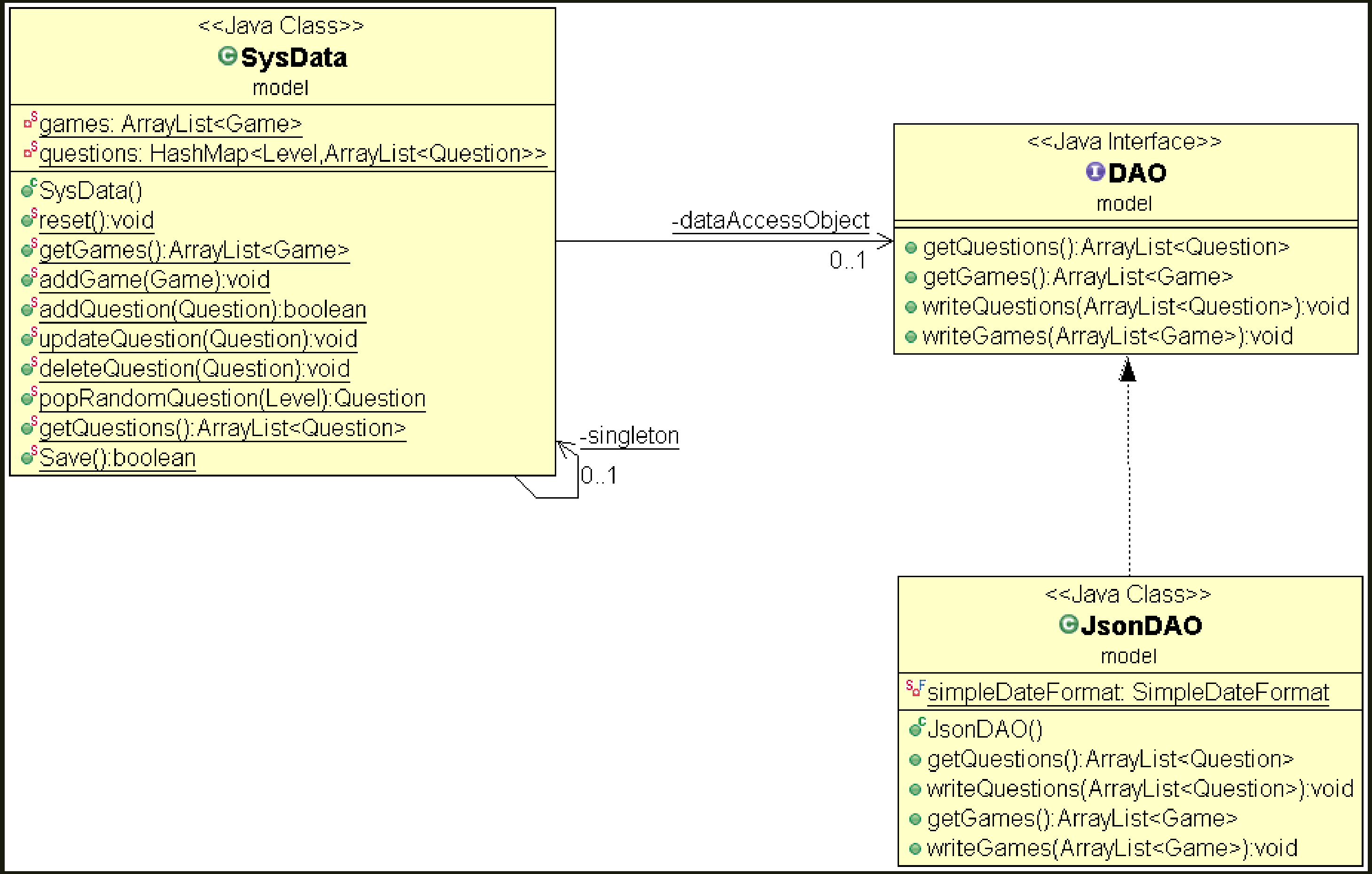
DAO-Data Access object

DAO stands for Data Access Object. DAO Design Pattern is used to separate the data persistence logic . This way, the service remains completely in dark about how the low-level operations to access the database is done. This is known as the principle of Separation of Logic. For example, if you shift from .json mechanism to SQL Database, your change will be limited to data access object and won't impact controller layer or model Objects and all you have to do is to add a new SqlDAO and implement the interface methods



Advantages

- DAO design pattern keeps coupling low between different parts of an application.
- DAO design pattern allows JUnit test to run faster as it allows to create Mock and avoid connecting to database to run tests. It improves testing because it's easy to write test with Mock objects, rather than an Integration test with the database. In the case of any issue, while running Unit test, you only need to check code and not database. Also shields with database connectivity and environment issues.
- Since DAO pattern is based on interface, it also promotes Object oriented design principle "programming for interface than implementation" which results in flexible and quality code.

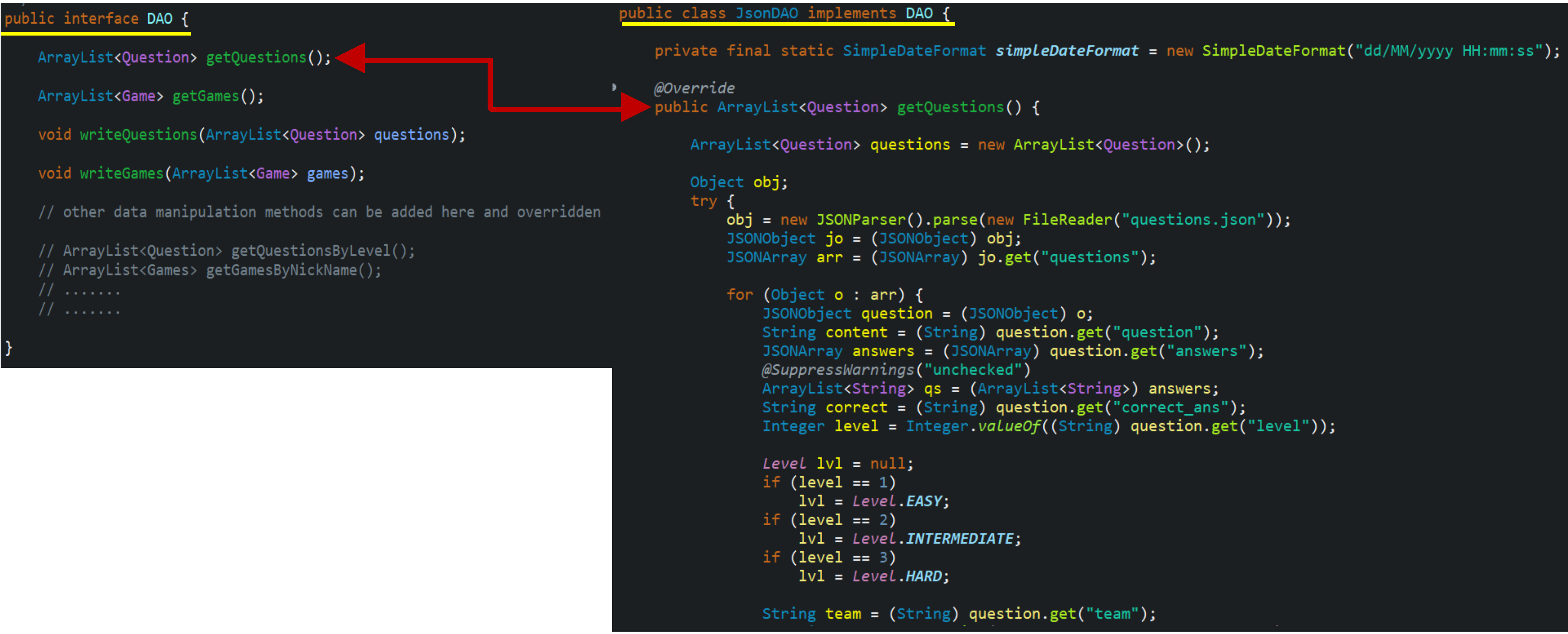


<<Java Class>>
SysData
model
games: ArrayList<Game>
questions: HashMap<Level,ArrayList<Question>>
SysData()
reset():void
getGames():ArrayList<Game>
addGame(Game):void
addQuestion(Question):boolean
updateQuestion(Question):void
deleteQuestion(Question):void
popRandomQuestion(Level):Question
getQuestions():ArrayList<Question>
Save():boolean
-singleton
0..1
-dataAccessObject
0..1
<<Java Interface>>
DAO
model
getQuestions():ArrayList<Question>
getGames():ArrayList<Game>
writeQuestions(ArrayList<Question>):void
writeGames(ArrayList<Game>):void
<<Java Class>>
JsonDAO
model
simpleDateFormat: SimpleDateFormat
JsonDAO()
getQuestions():ArrayList<Question>
writeQuestions(ArrayList<Question>):void
getGames():ArrayList<Game>
writeGames(ArrayList<Game>):void

UML

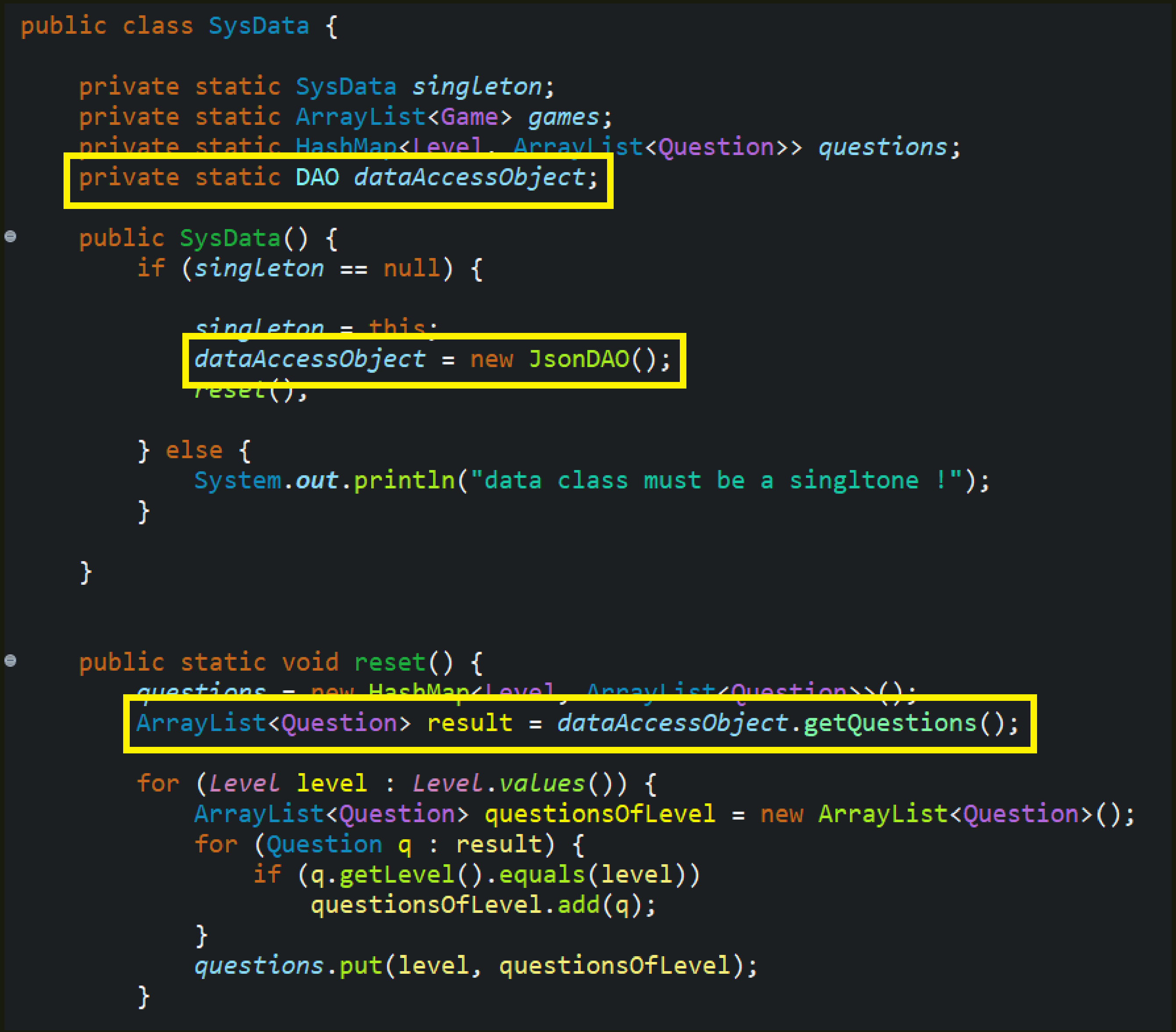
SysData class has no idea how the implimintation of the low level data processing is done

It only knows that it can use the DAO methods to read and write the application data



```java
public interface DAO {

    ArrayList<Question> getQuestions();

    ArrayList<Game> getGames();

    void writeQuestions(ArrayList<Question> questions);

    void writeGames(ArrayList<Game> games);

    // other data manipulation methods can be added here and overridden

    // ArrayList<Question> getQuestionsByLevel();
    // ArrayList<Games> getGamesByNickName();
    // .......
    // .......

}
```

```java
public class JsonDAO implements DAO {

    private final static SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

    @Override
    public ArrayList<Question> getQuestions() {

        ArrayList<Question> questions = new ArrayList<Question>();

        Object obj;
        try {
            obj = new JSONParser().parse(new FileReader("questions.json"));
            JSONObject jo = (JSONObject) obj;
            JSONArray arr = (JSONArray) jo.get("questions");

            for (Object o : arr) {
                JSONObject question = (JSONObject) o;
                String content = (String) question.get("question");
                JSONArray answers = (JSONArray) question.get("answers");
                @SuppressWarnings("unchecked")
                ArrayList<String> qs = (ArrayList<String>) answers;
                String correct = (String) question.get("correct_ans");
                Integer level = Integer.valueOf((String) question.get("level"));

                Level lvl = null;
                if (level == 1)
                    lvl = Level.EASY;
                if (level == 2)
                    lvl = Level.INTERMEDIATE;
                if (level == 3)
                    lvl = Level.HARD;

                String team = (String) question.get("team");
```



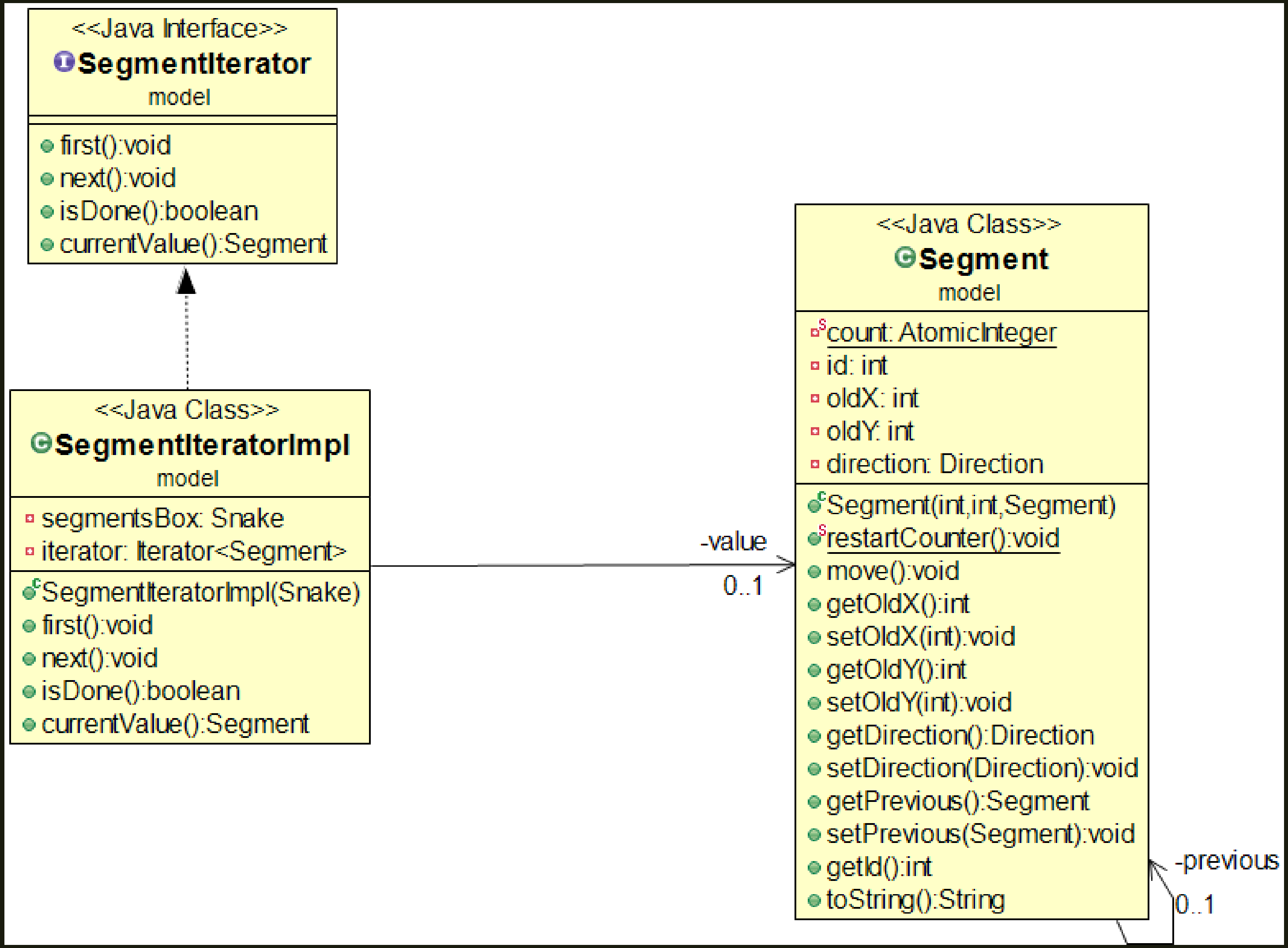
```java
public class SysData {

    private static SysData singleton;
    private static ArrayList<Game> games;
    private static HashMap<Level, ArrayList<Question>> questions;
    private static DAO dataAccessObject;

    public SysData() {
        if (singleton == null) {

            singleton = this;
            dataAccessObject = new JsonDAO();
            reset();

        } else {
            System.out.println("data class must be a singltone !");
        }

    }


    public static void reset() {
        questions = new HashMap<Level, ArrayList<Question>>();
        ArrayList<Question> result = dataAccessObject.getQuestions();

        for (Level level : Level.values()) {
            ArrayList<Question> questionsOfLevel = new ArrayList<Question>();
            for (Question q : result) {
                if (q.getLevel().equals(level))
                    questionsOfLevel.add(q);
            }
            questions.put(level, questionsOfLevel);
        }
```

USAGE

Iterator

- Iterator design pattern in one of the behavioral pattern. Iterator pattern is used to provide a standard way to traverse through a group of Objects.
- We used the Iterator to provide us a ways to access elements of an aggregate object (Snake Body) sequentially without exposing the underlying structure of the object.
- Iterator pattern is not only about traversing through a collection, we can provide different kind of iterators based on our requirements.
- Iterator design pattern hides the actual implementation of traversal through the collection and client programs just use iterator methods.
- in case we changed the data structure that holds the segments we will not need to change the iteration technique in all the places where we used the set , we can still iterate over it the same way with iterator .

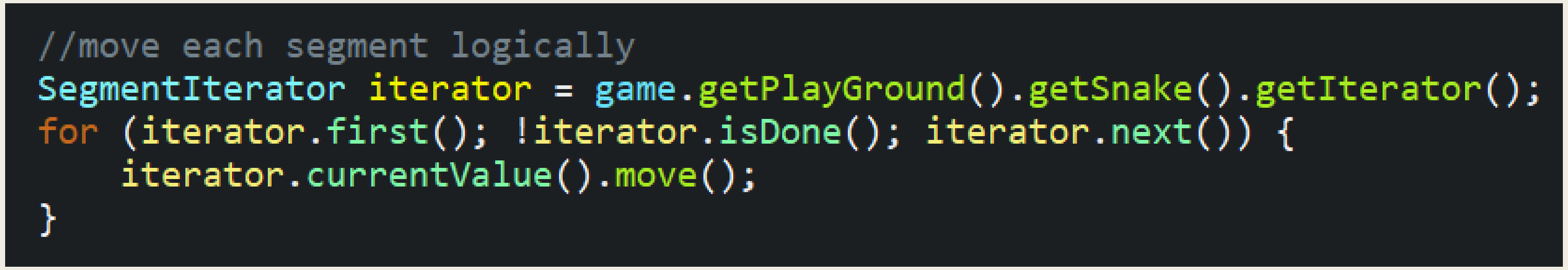
UML



<<Java Interface>>
SegmentIterator
model
first():void
next():void
isDone():boolean
currentValue():Segment
<<Java Class>>
SegmentIteratorImpl
model
segmentsBox: Snake
iterator: Iterator<Segment>
SegmentIteratorImpl(Snake)
first():void
next():void
isDone():boolean
currentValue():Segment
-value
0..1
<<Java Class>>
Segment
model
count: AtomicInteger
id: int
oldX: int
oldY: int
direction: Direction
Segment(int,int,Segment)
restartCounter():void
move():void
getOldX():int
setOldX(int):void
getOldY():int
setOldY(int):void
getDirection():Direction
setDirection(Direction):void
getPrevious():Segment
setPrevious(Segment):void
getId():int
toString():String
-previous
0..1

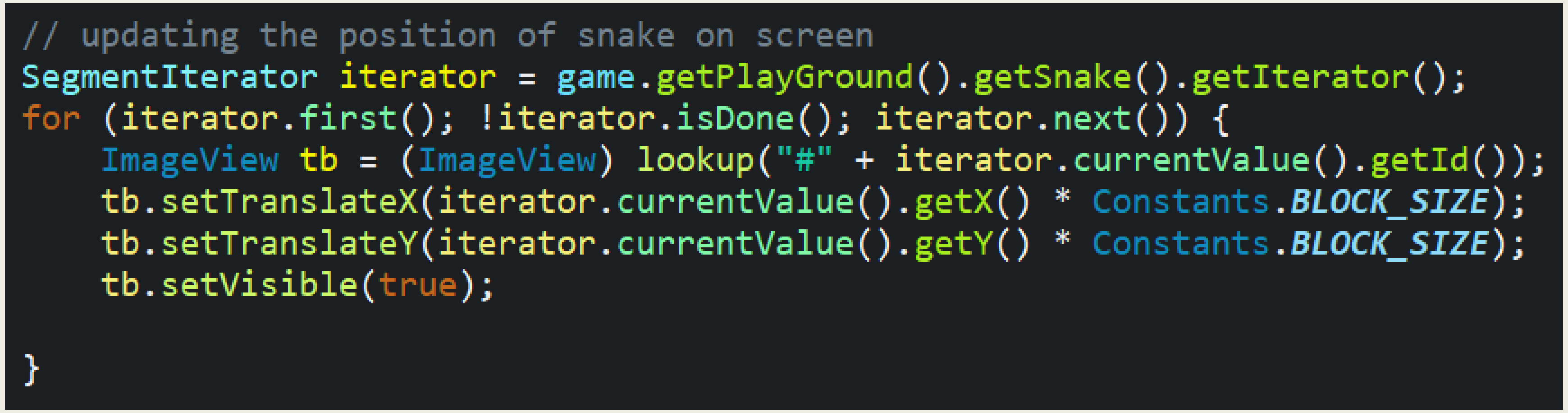
Usage

Controller



```java
//move each segment logically
SegmentIterator iterator = game.getPlayGround().getSnake().getIterator();
for (iterator.first(); !iterator.isDone(); iterator.next()) {
    iterator.currentValue().move();
}
```

View



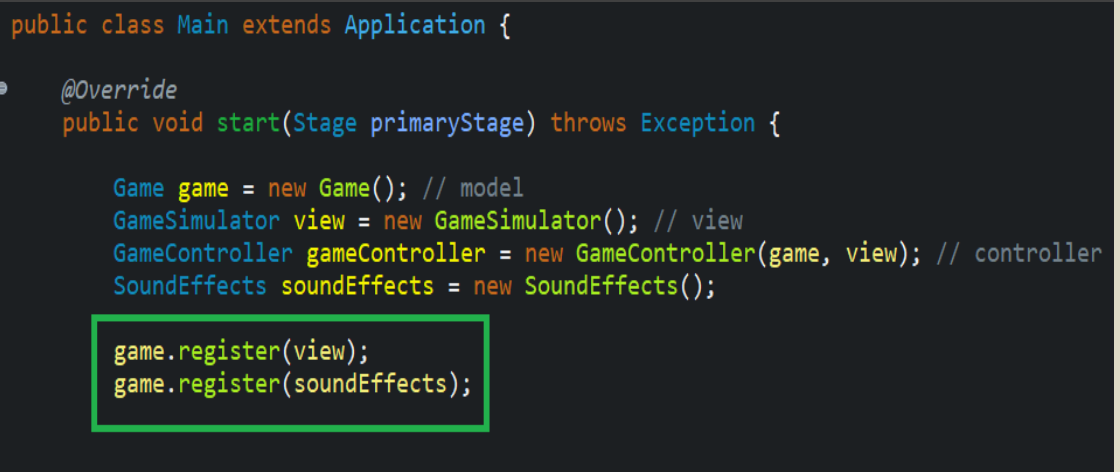
```java
// updating the position of snake on screen
SegmentIterator iterator = game.getPlayGround().getSnake().getIterator();
for (iterator.first(); !iterator.isDone(); iterator.next()) {
    ImageView tb = (ImageView) lookup("#" + iterator.currentValue().getId());
    tb.setTranslateX(iterator.currentValue().getX() * Constants.BLOCK_SIZE);
    tb.setTranslateY(iterator.currentValue().getY() * Constants.BLOCK_SIZE);
    tb.setVisible(true);

}
```

Observer

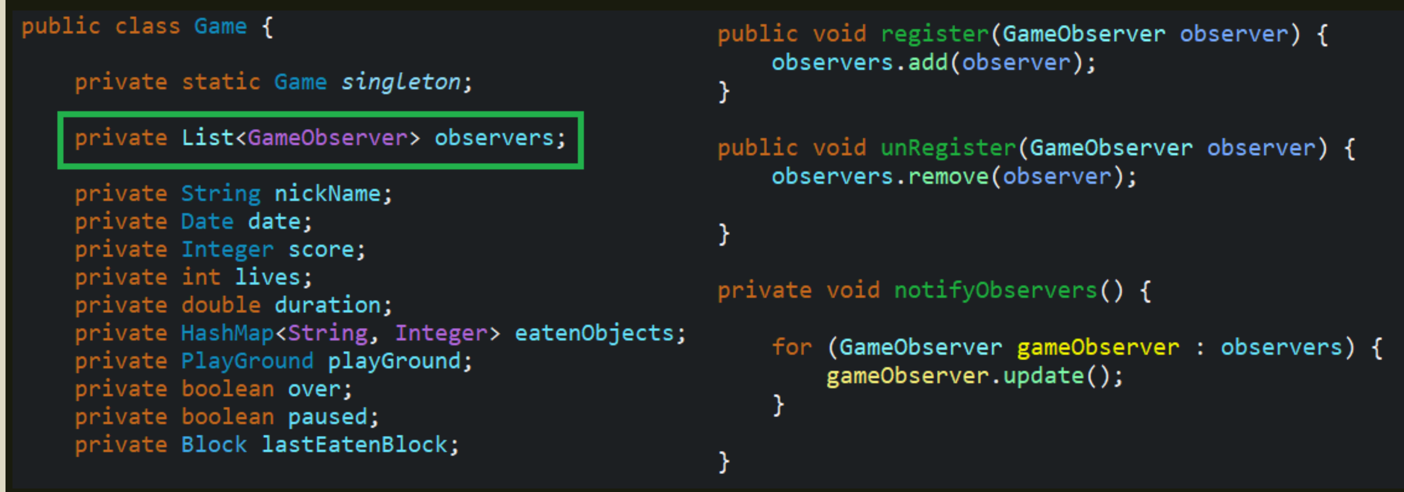
- **Observer Pattern** is one of the **behavioral** design pattern**.** Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change. In observer pattern, the object that watch on the state of another object are called **Observer** and the object that is being watched is called **Subject.**
- in our project , a Game should notify channels when it eats objects. eating objects is what changes the state of the game , and it causes the channels (GameSimulator , SoundEffects) to be notified and to do actions according to the game state.

Game.java

- **Subject** (or game in our case) contains a list of observers to notify of any change in it's state, so it should provide methods using which observers can register and unregister themselves.
- Subject also contain a method to notify all the observers of any change and either it can send the update while notifying the observer .

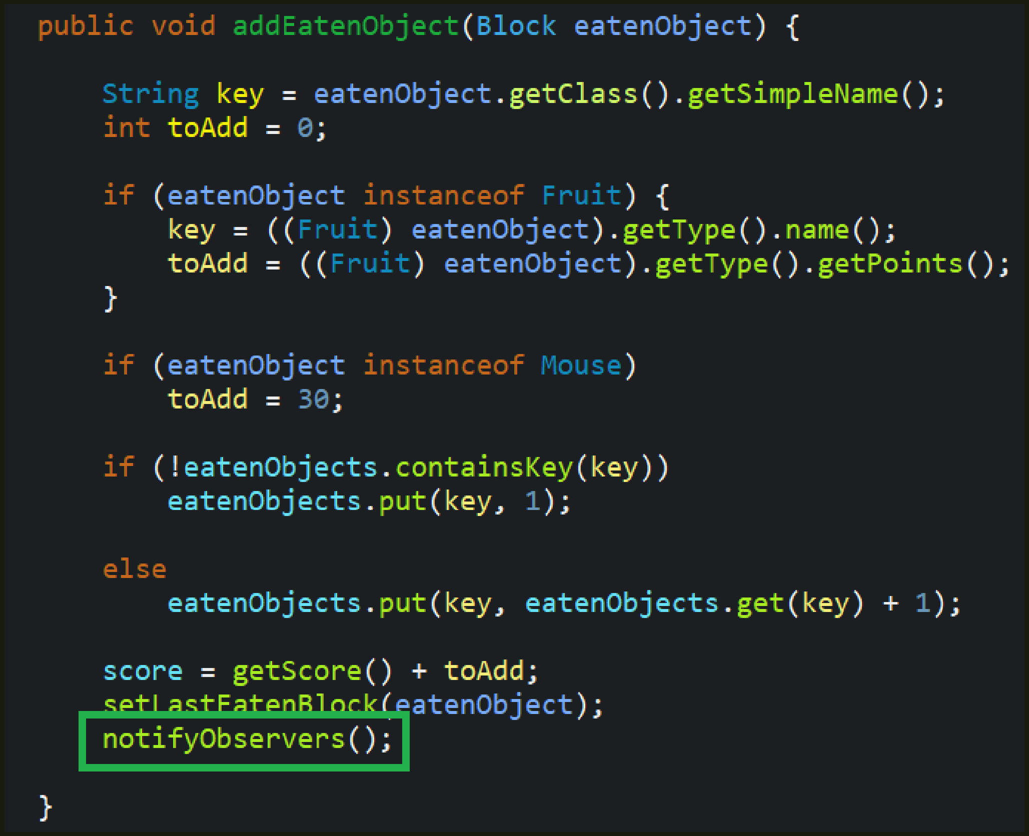


```java
public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {

        Game game = new Game(); // model
        GameSimulator view = new GameSimulator(); // view
        GameController gameController = new GameController(game, view); // controller
        SoundEffects soundEffects = new SoundEffects();

        game.register(view);
        game.register(soundEffects);
```



```java
public class Game {

    private static Game singleton;

    private List<GameObserver> observers;

    private String nickName;
    private Date date;
    private Integer score;
    private int lives;
    private double duration;
    private HashMap<String, Integer> eatenObjects;
    private PlayGround playGround;
    private boolean over;
    private boolean paused;
    private Block lastEatenBlock;
```

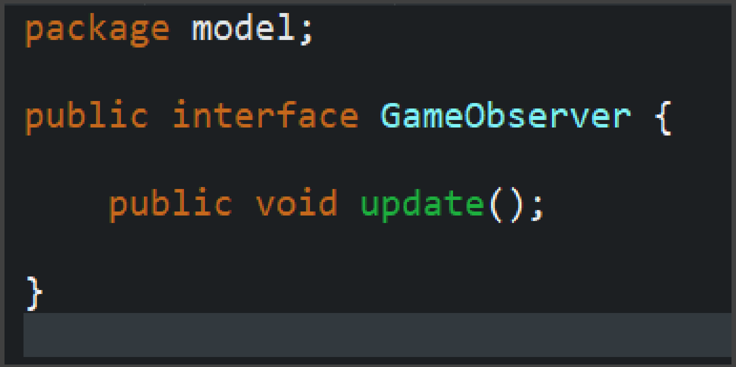
```java
public void register(GameObserver observer) {
    observers.add(observer);
}

public void unRegister(GameObserver observer) {
    observers.remove(observer);

}

private void notifyObservers() {

    for (GameObserver gameObserver : observers) {
        gameObserver.update();
    }

}
```



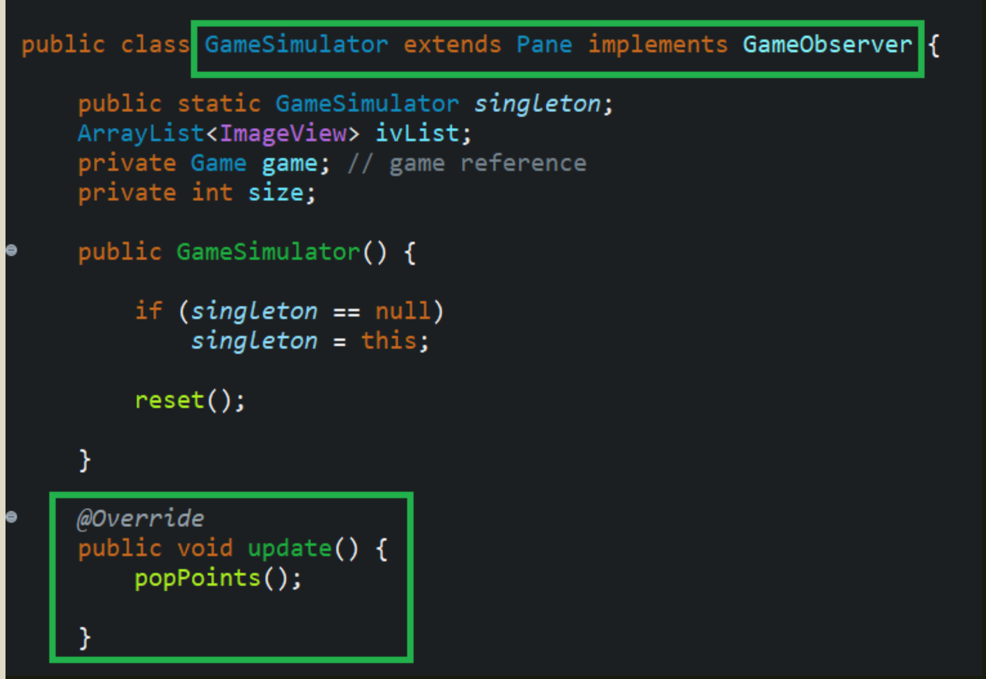
```java
public void addEatenObject(Block eatenObject) {

    String key = eatenObject.getClass().getSimpleName();
    int toAdd = 0;

    if (eatenObject instanceof Fruit) {
        key = ((Fruit) eatenObject).getType().name();
        toAdd = ((Fruit) eatenObject).getType().getPoints();
    }

    if (eatenObject instanceof Mouse)
        toAdd = 30;

    if (!eatenObjects.containsKey(key))
        eatenObjects.put(key, 1);

    else
        eatenObjects.put(key, eatenObjects.get(key) + 1);

    score = getScore() + toAdd;
    setLastEatenBlock(eatenObject);
    notifyObservers();

}
```

Observer

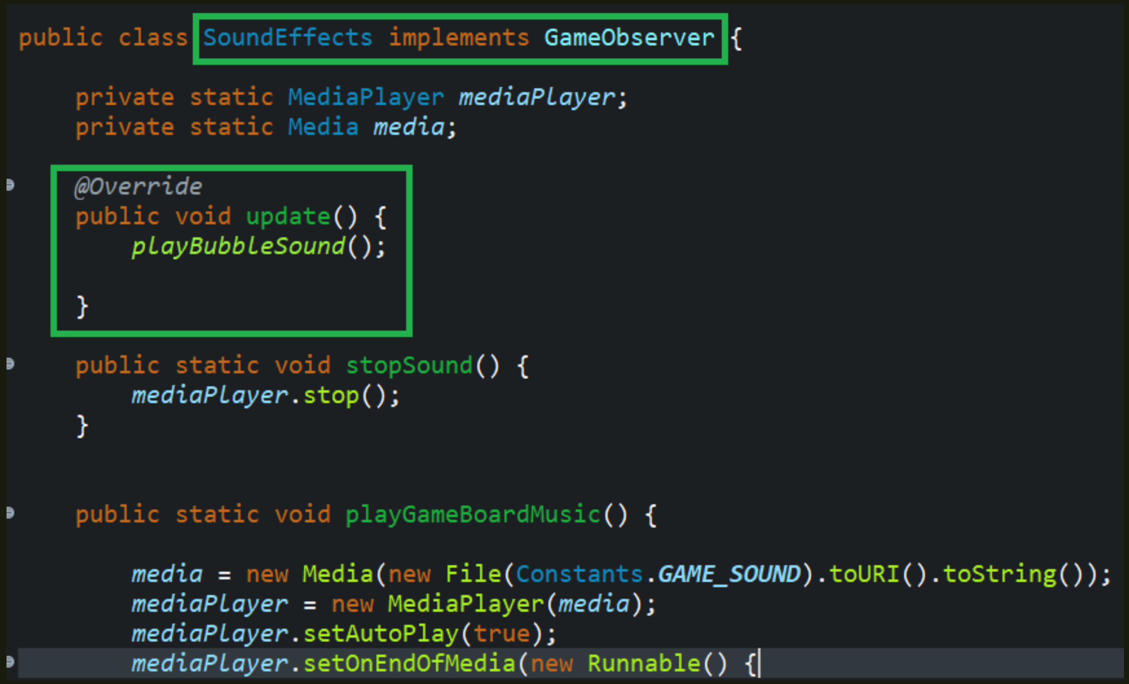
- Observer should have a method that will be used by Subject (Game) to notify them of any updates and another method to set the object to watch in our case we will not need it because the subject is singleton we can read it's state all the time .



```java
package model;

public interface GameObserver {

    public void update();

}
```



```java
public class GameSimulator extends Pane implements GameObserver {

    public static GameSimulator singleton;
    ArrayList<ImageView> ivList;
    private Game game; // game reference
    private int size;

    public GameSimulator() {

        if (singleton == null)
            singleton = this;

        reset();

    }

    @Override
    public void update() {
        popPoints();

    }
```



```java
public class SoundEffects implements GameObserver {

    private static MediaPlayer mediaPlayer;
    private static Media media;

    @Override
    public void update() {
        playBubbleSound();

    }

    public static void stopSound() {
        mediaPlayer.stop();
    }


    public static void playGameBoardMusic() {

        media = new Media(new File(Constants.GAME_SOUND).toURI().toString());
        mediaPlayer = new MediaPlayer(media);
        mediaPlayer.setAutoPlay(true);
        mediaPlayer.setOnEndOfMedia(new Runnable() {
```