Tower Defense

Second Build - Design Document

Amritansh Mishra	27288883
George Ekow-Daniels	40011883
Baitassov Ulan	40001592
Sajjad Ashraf	40012126

Table of Contents

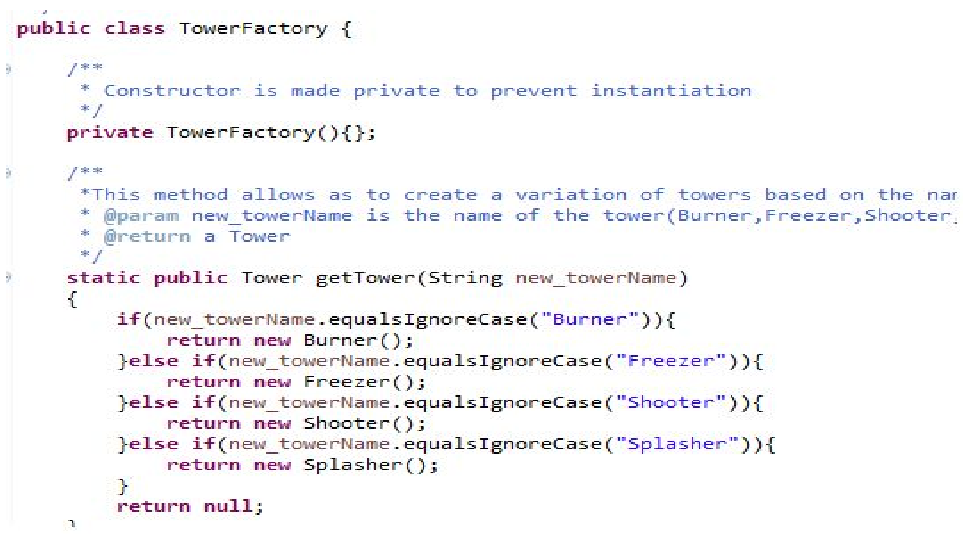
1.Introduction

Design document outlines the architectural and component design of the Tower Defense Game. It combines textual descriptions and UML class diagrams. The architectural design uses four different design patterns namely Observer pattern, Singleton pattern, Strategy and Factory pattern. In this document we will discuss the overall architecture of the game, the internal sub system design, code organization and testing strategies.

2.Architecture Design

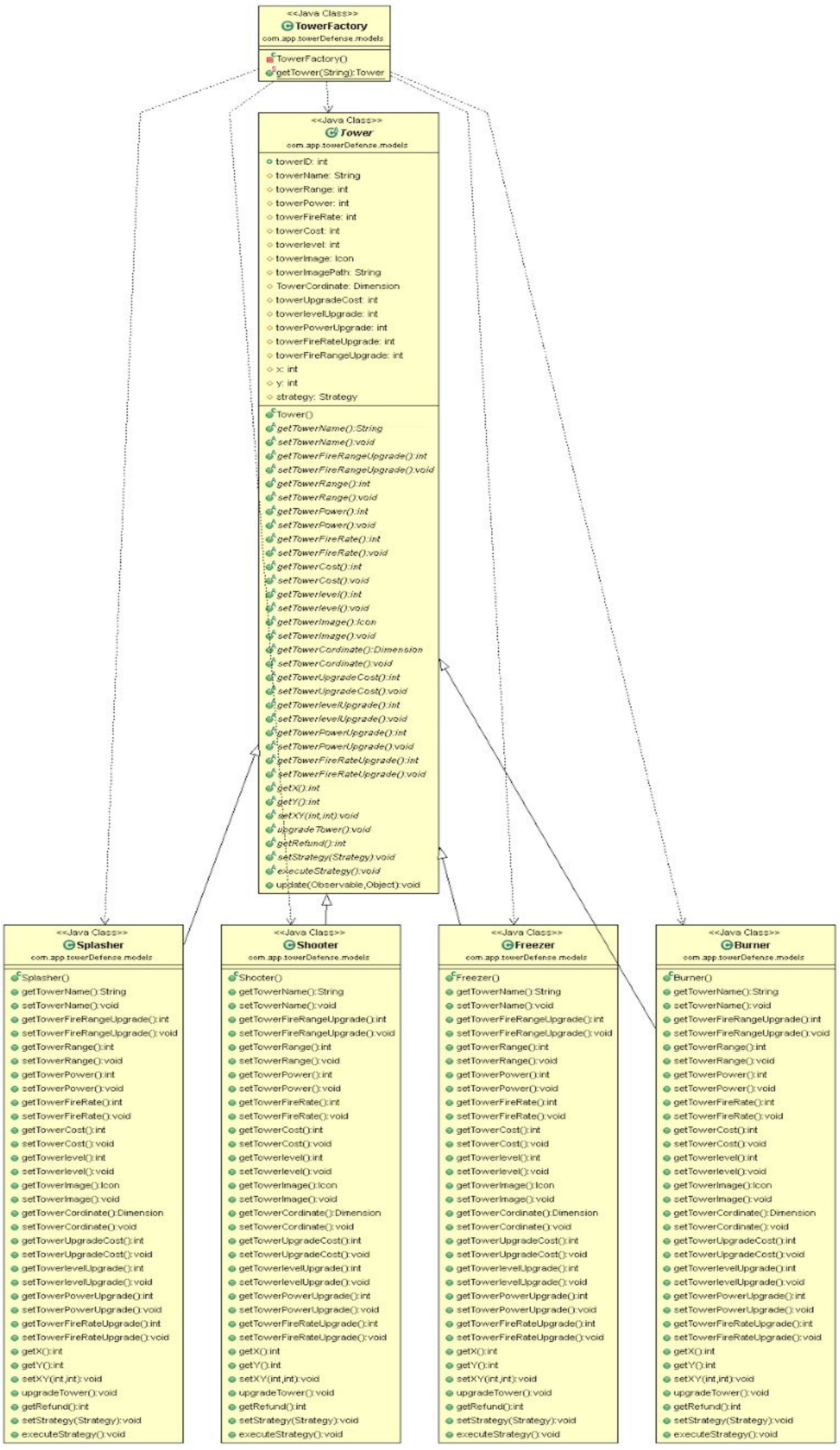
The architecture of our system adopts the widely used patterns :

1. **Factory pattern** : Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.The architecture of our system adopts the Factory design pattern for two of the most important components of the game that are tower and critters.

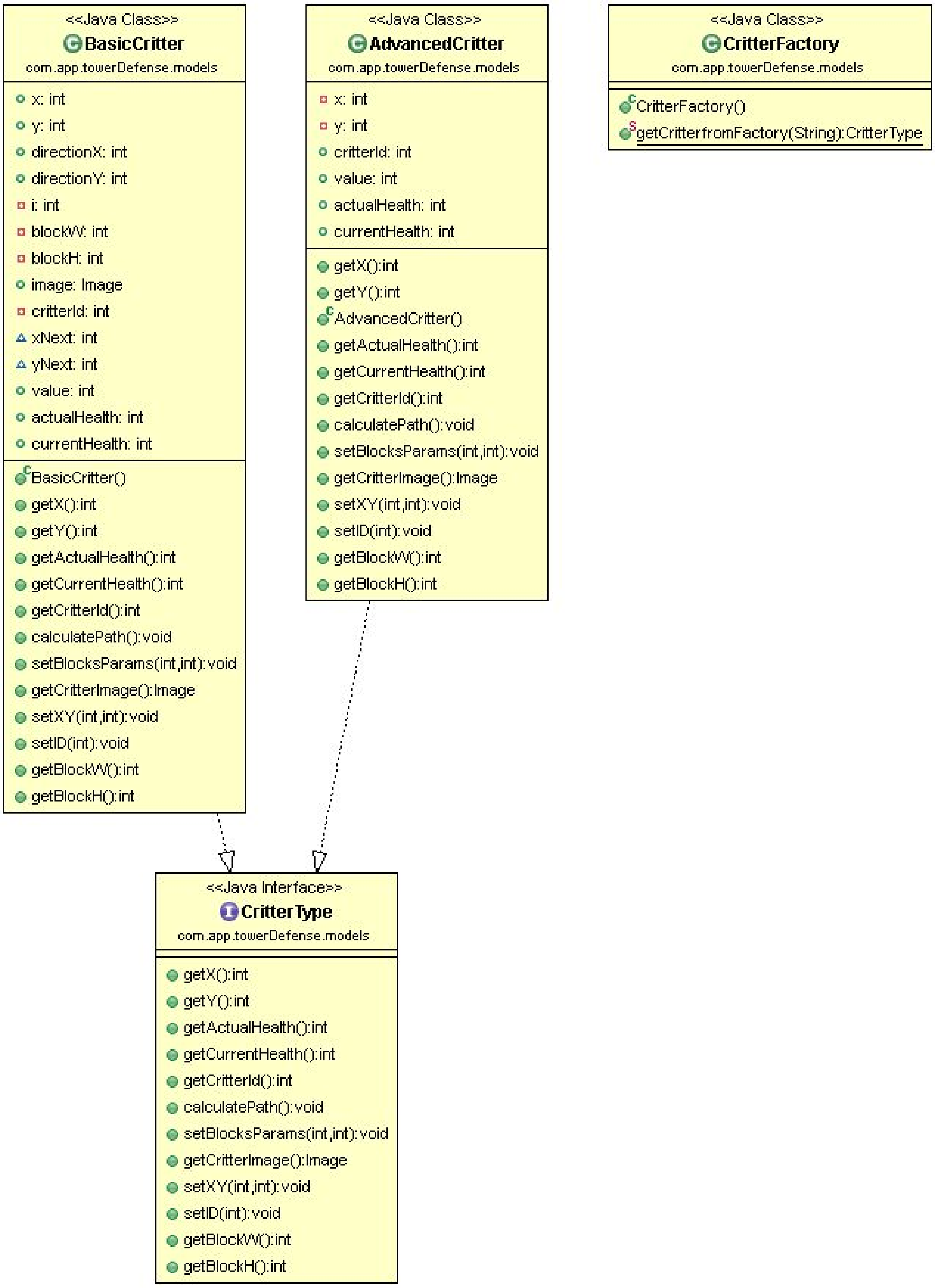


```java
public class TowerFactory {

    /**
     * Constructor is made private to prevent instantiation
     */
    private TowerFactory(){};

    /**
     *This method allows as to create a variation of towers based on the nar
     * @param new_towerName is the name of the tower(Burner,Freezer,Shooter,
     * @return a Tower
     */
    static public Tower getTower(String new_towerName)
    {
        if(new_towerName.equalsIgnoreCase("Burner")){
            return new Burner();
        }else if(new_towerName.equalsIgnoreCase("Freezer")){
            return new Freezer();
        }else if(new_towerName.equalsIgnoreCase("Shooter")){
            return new Shooter();
        }else if(new_towerName.equalsIgnoreCase("Splasher")){
            return new Splasher();
        }
        return null;
    }
```

Code Snippet of factory pattern in TowerFactory.java



<<Java Class>>
TowerFactory
com.app.towerDefense.models
TowerFactory()
getTower(String):Tower
<<Java Class>>
Tower
com.app.towerDefense.models
towerID: int
towerName: String
towerRange: int
towerPower: int
towerFireRate: int
towerCost: int
towerlevel: int
towerImage: Icon
towerImagePath: String
TowerCordinate: Dimension
towerUpgradeCost: int
towerlevelUpgrade: int
towerPowerUpgrade: int
towerFireRateUpgrade: int
towerFireRangeUpgrade: int
x: int
y: int
strategy: Strategy
Tower()
getTowerName():String
setTowerName():void
getTowerFireRangeUpgrade():int
setTowerFireRangeUpgrade():void
getTowerRange():int
setTowerRange():void
getTowerPower():int
setTowerPower():void
getTowerFireRate():int
setTowerFireRate():void
getTowerCost():int
setTowerCost():void
getTowerlevel():int
setTowerlevel():void
getTowerImage():Icon
setTowerImage():void
getTowerCordinate():Dimension
setTowerCordinate():void
getTowerUpgradeCost():int
setTowerUpgradeCost():void
getTowerlevelUpgrade():int
setTowerlevelUpgrade():void
getTowerPowerUpgrade():int
setTowerPowerUpgrade():void
getTowerFireRateUpgrade():int
setTowerFireRateUpgrade():void
getX():int
getY():int
setXY(int,int):void
upgradeTower():void
getRefund():int
setStrategy(Strategy):void
executeStrategy():void
update(Observable,Object):void
<<Java Class>>
Splasher
com.app.towerDefense.models
Splasher()
<<Java Class>>
Shooter
com.app.towerDefense.models
Shooter()
<<Java Class>>
Freezer
com.app.towerDefense.models
Freezer()
<<Java Class>>
Burner
com.app.towerDefense.models
Burner()
getTowerName():String
setTowerName():void
getTowerFireRangeUpgrade():int
setTowerFireRangeUpgrade():void
getTowerRange():int
setTowerRange():void
getTowerPower():int
setTowerPower():void
getTowerFireRate():int
setTowerFireRate():void
getTowerCost():int
setTowerCost():void
getTowerlevel():int
setTowerlevel():void
getTowerImage():Icon
setTowerImage():void
getTowerCordinate():Dimension
setTowerCordinate():void
getTowerUpgradeCost():int
setTowerUpgradeCost():void
getTowerlevelUpgrade():int
setTowerlevelUpgrade():void
getTowerPowerUpgrade():int
setTowerPowerUpgrade():void
getTowerFireRateUpgrade():int
setTowerFireRateUpgrade():void
getX():int
getY():int
setXY(int,int):void
upgradeTower():void
getRefund():int
setStrategy(Strategy):void
executeStrategy():void

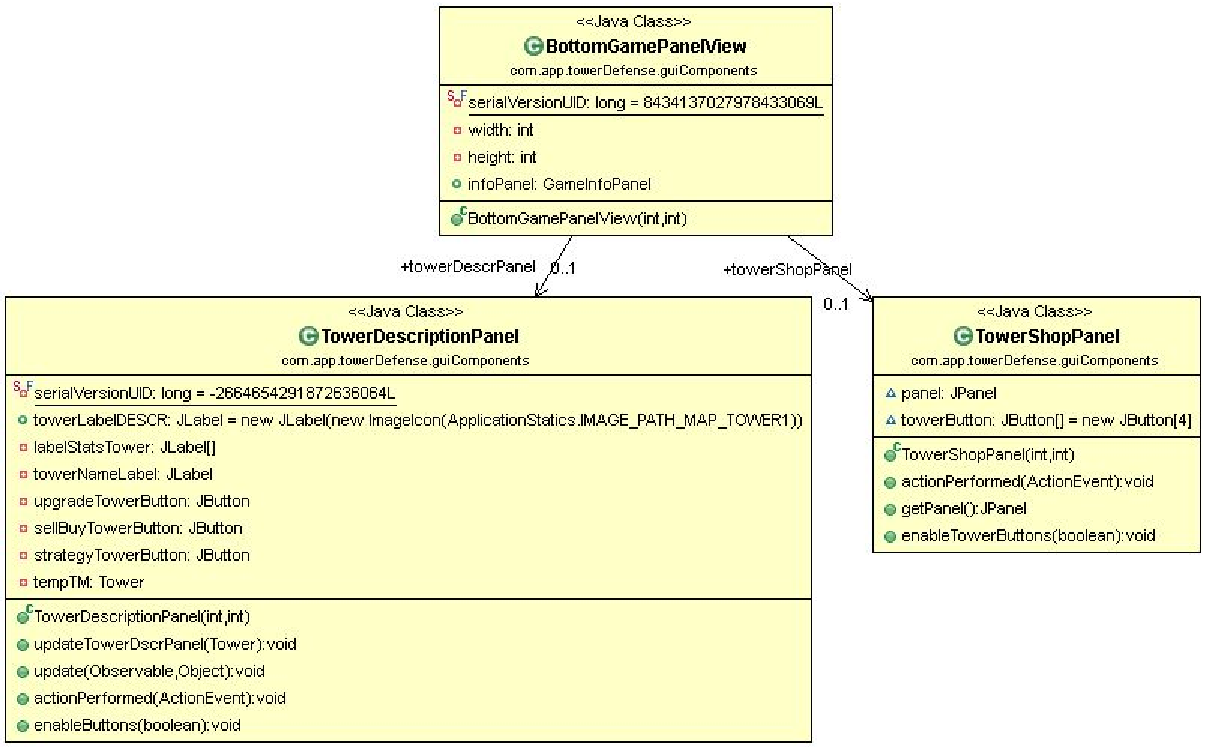


UML class diagrams for tower and critters

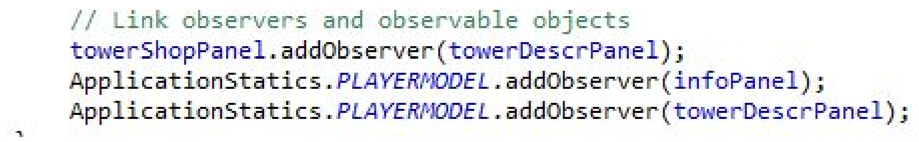
2. **Observer pattern :** Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern falls under behavioral pattern category. In our application we have the following structure for implementing observer pattern :

Class Name	**Role**
BottomGamePanel	Object/Subject
TowerShopPanel	Observable
TowerDescriptionPanel	Observer
.BottomGamePanel	Object/Subject
PlayerModel	Observable
GameInfoPanel	Observer
Tower	Observer
CritterType	Observable
AdvancedCritter	Observable
BasicCritter	Observable

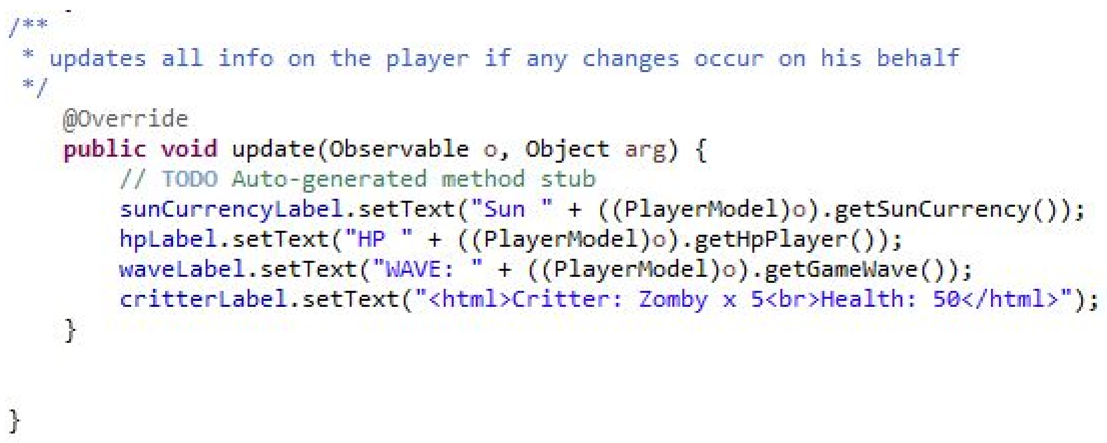
UML Class diagrams for observer pattern



<<Java Class>>
BottomGamePanelView
com.app.towerDefense.guiComponents
serialVersionUID: long = 8434137027978433069L
width: int
height: int
infoPanel: GameInfoPanel
BottomGamePanelView(int,int)
+towerDescrPanel
0..1
+towerShopPanel
0..1
<<Java Class>>
TowerDescriptionPanel
com.app.towerDefense.guiComponents
serialVersionUID: long = -2664654291872636064L
towerLabelDESCR: JLabel = new JLabel(new ImageIcon(ApplicationStatics.IMAGE_PATH_MAP_TOWER1))
labelStatsTower: JLabel[]
towerNameLabel: JLabel
upgradeTowerButton: JButton
sellBuyTowerButton: JButton
strategyTowerButton: JButton
tempTM: Tower
TowerDescriptionPanel(int,int)
updateTowerDscrPanel(Tower):void
update(Observable,Object):void
actionPerformed(ActionEvent):void
enableButtons(boolean):void
<<Java Class>>
TowerShopPanel
com.app.towerDefense.guiComponents
panel: JPanel
towerButton: JButton[] = new JButton[4]
TowerShopPanel(int,int)
actionPerformed(ActionEvent):void
getPanel():JPanel
enableTowerButtons(boolean):void

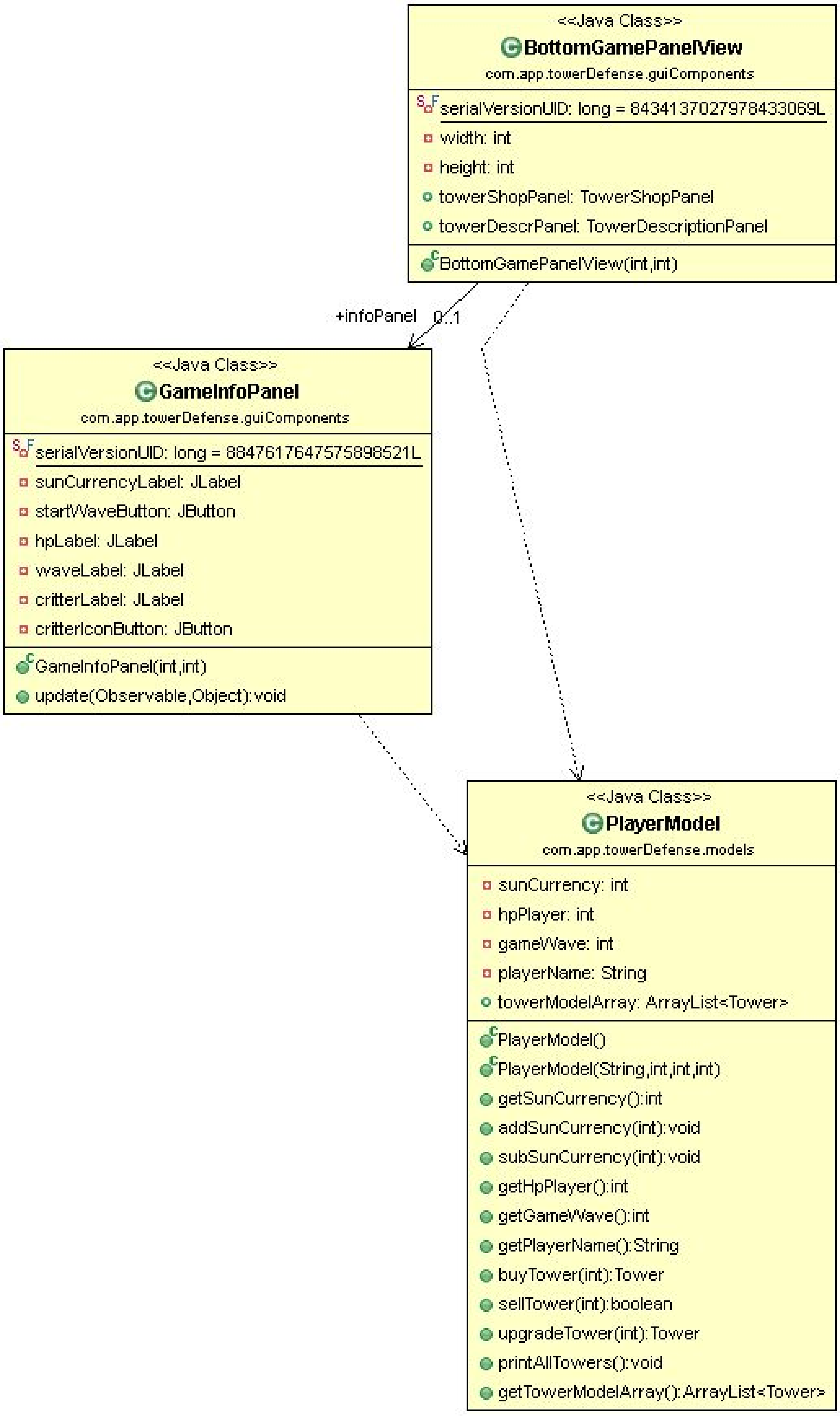


```
        // Link observers and observable objects
        towerShopPanel.addObserver(towerDescrPanel);
        ApplicationStatics.PLAYERMODEL.addObserver(infoPanel);
        ApplicationStatics.PLAYERMODEL.addObserver(towerDescrPanel);
```

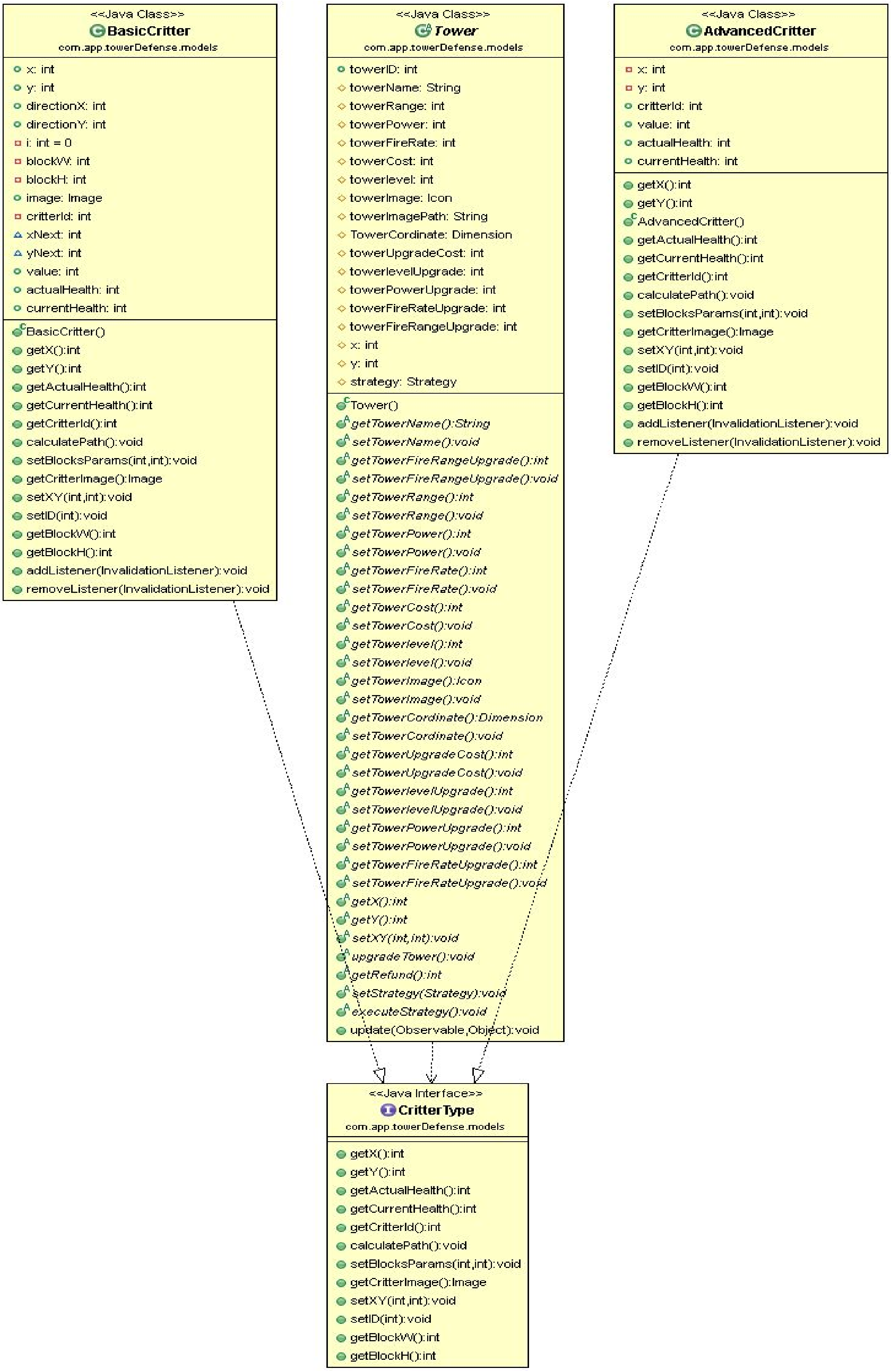


```
/**
 * updates all info on the player if any changes occur on his behalf
 */
    @Override
    public void update(Observable o, Object arg) {
        // TODO Auto-generated method stub
        sunCurrencyLabel.setText("Sun " + ((PlayerModel)o).getSunCurrency());
        hpLabel.setText("HP " + ((PlayerModel)o).getHpPlayer());
        waveLabel.setText("WAVE: " + ((PlayerModel)o).getGameWave());
        critterLabel.setText("<html>Critter: Zomby x 5<br>Health: 50</html>");
    }


}
```

Code snippet of observer pattern in BottomGamePanelView.java

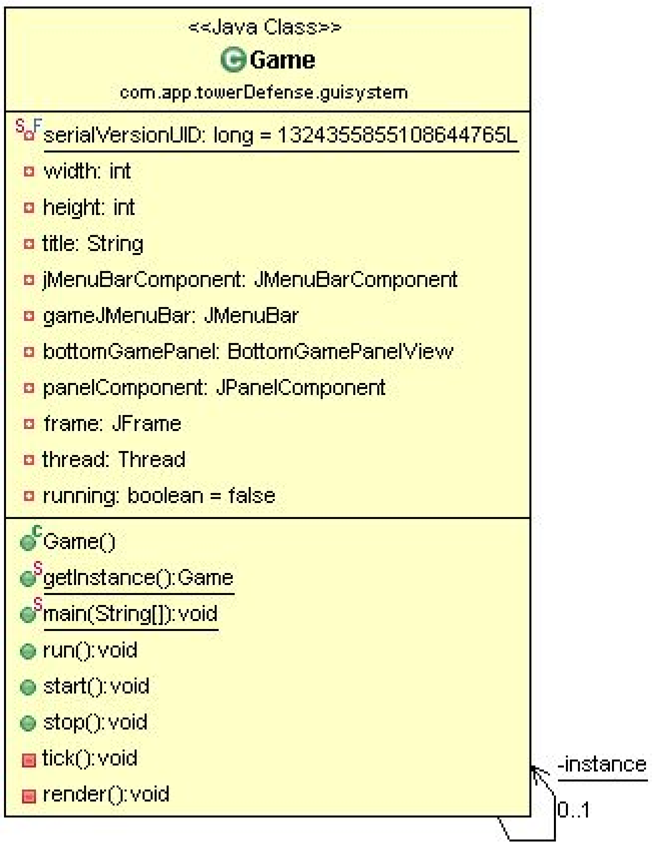


<<Java Class>>
BottomGamePanelView
com.app.towerDefense.guiComponents
serialVersionUID: long = 8434137027978433069L
width: int
height: int
towerShopPanel: TowerShopPanel
towerDescrPanel: TowerDescriptionPanel
BottomGamePanelView(int,int)
+infoPanel
0..1
<<Java Class>>
GameInfoPanel
com.app.towerDefense.guiComponents
serialVersionUID: long = 8847617647575898521L
sunCurrencyLabel: JLabel
startWaveButton: JButton
hpLabel: JLabel
waveLabel: JLabel
critterLabel: JLabel
critterIconButton: JButton
GameInfoPanel(int,int)
update(Observable,Object):void
<<Java Class>>
PlayerModel
com.app.towerDefense.models
sunCurrency: int
hpPlayer: int
gameWave: int
playerName: String
towerModelArray: ArrayList<Tower>
PlayerModel()
PlayerModel(String,int,int,int)
getSunCurrency():int
addSunCurrency(int):void
subSunCurrency(int):void
getHpPlayer():int
getGameWave():int
getPlayerName():String
buyTower(int):Tower
sellTower(int):boolean
upgradeTower(int):Tower
printAllTowers():void
getTowerModelArray():ArrayList<Tower>

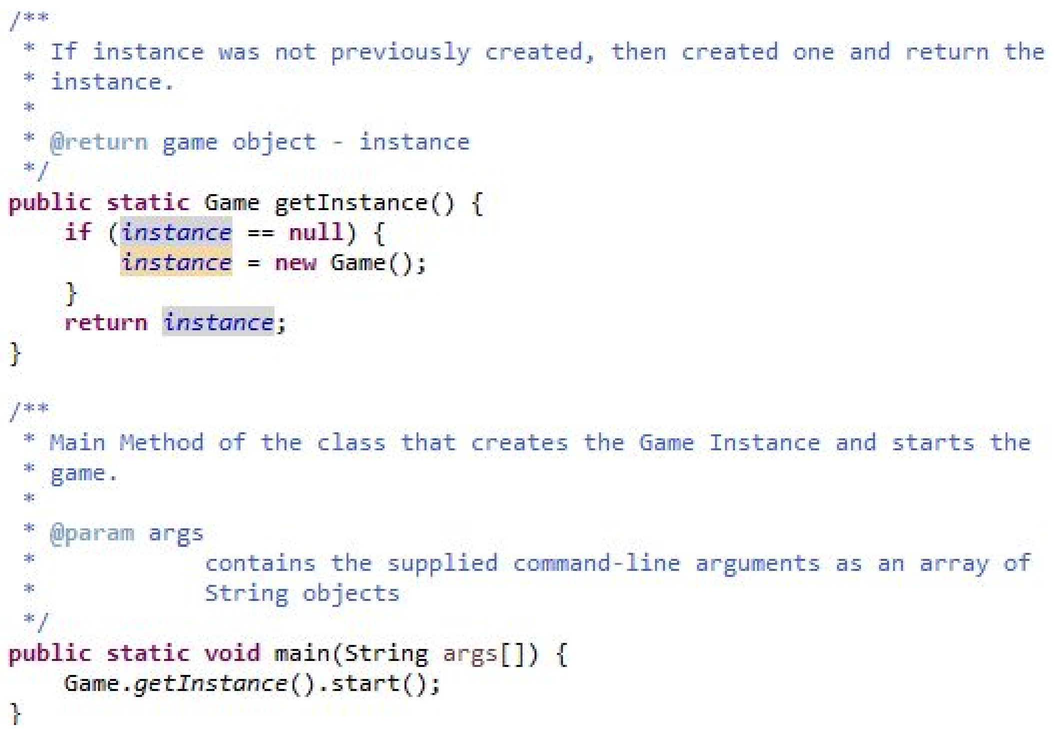


<<Java Class>>
BasicCritter
com.app.towerDefense.models
x: int
y: int
directionX: int
directionY: int
i: int = 0
blockW: int
blockH: int
image: Image
critterId: int
xNext: int
yNext: int
value: int
actualHealth: int
currentHealth: int
BasicCritter()
getX():int
getY():int
getActualHealth():int
getCurrentHealth():int
getCritterId():int
calculatePath():void
setBlocksParams(int,int):void
getCritterImage():Image
setXY(int,int):void
setID(int):void
getBlockW():int
getBlockH():int
addListener(InvalidationListener):void
removeListener(InvalidationListener):void
<<Java Class>>
Tower
com.app.towerDefense.models
towerID: int
towerName: String
towerRange: int
towerPower: int
towerFireRate: int
towerCost: int
towerlevel: int
towerImage: Icon
towerImagePath: String
TowerCordinate: Dimension
towerUpgradeCost: int
towerlevelUpgrade: int
towerPowerUpgrade: int
towerFireRateUpgrade: int
towerFireRangeUpgrade: int
x: int
y: int
strategy: Strategy
Tower()
getTowerName():String
setTowerName():void
getTowerFireRangeUpgrade():int
setTowerFireRangeUpgrade():void
getTowerRange():int
setTowerRange():void
getTowerPower():int
setTowerPower():void
getTowerFireRate():int
setTowerFireRate():void
getTowerCost():int
setTowerCost():void
getTowerlevel():int
setTowerlevel():void
getTowerImage():Icon
setTowerImage():void
getTowerCordinate():Dimension
setTowerCordinate():void
getTowerUpgradeCost():int
setTowerUpgradeCost():void
getTowerlevelUpgrade():int
setTowerlevelUpgrade():void
getTowerPowerUpgrade():int
setTowerPowerUpgrade():void
getTowerFireRateUpgrade():int
setTowerFireRateUpgrade():void
getX():int
getY():int
setXY(int,int):void
upgradeTower():void
getRefund():int
setStrategy(Strategy):void
executeStrategy():void
update(Observable,Object):void
<<Java Class>>
AdvancedCritter
com.app.towerDefense.models
x: int
y: int
critterId: int
value: int
actualHealth: int
currentHealth: int
getX():int
getY():int
AdvancedCritter()
getActualHealth():int
getCurrentHealth():int
getCritterId():int
calculatePath():void
setBlocksParams(int,int):void
getCritterImage():Image
setXY(int,int):void
setID(int):void
getBlockW():int
getBlockH():int
addListener(InvalidationListener):void
removeListener(InvalidationListener):void
<<Java Interface>>
CritterType
com.app.towerDefense.models
getX():int
getY():int
getActualHealth():int
getCurrentHealth():int
getCritterId():int
calculatePath():void
setBlocksParams(int,int):void
getCritterImage():Image
setXY(int,int):void
setID(int):void
getBlockW():int
getBlockH():int

3. **Singleton pattern** : Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create a single object. This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class. In this application we use the singleton pattern to ensure a class only has one instance, and provides a global point of access to it. This is useful when exactly one object is needed to coordinate actions across the system.



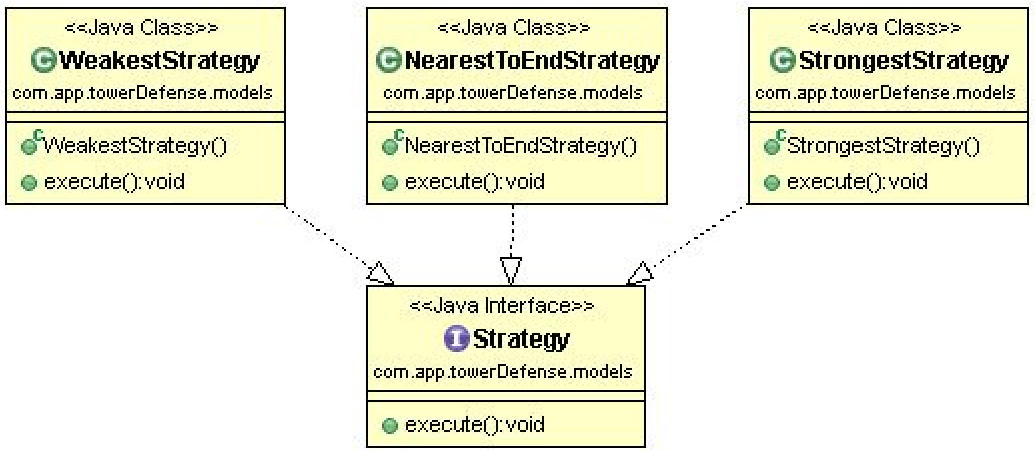
UML class diagram for Game.java



```java
/**
 * If instance was not previously created, then created one and return the
 * instance.
 *
 * @return game object - instance
 */
public static Game getInstance() {
    if (instance == null) {
        instance = new Game();
    }
    return instance;
}

/**
 * Main Method of the class that creates the Game Instance and starts the
 * game.
 *
 * @param args
 *            contains the supplied command-line arguments as an array of
 *            String objects
 */
public static void main(String args[]) {
    Game.getInstance().start();
}
```

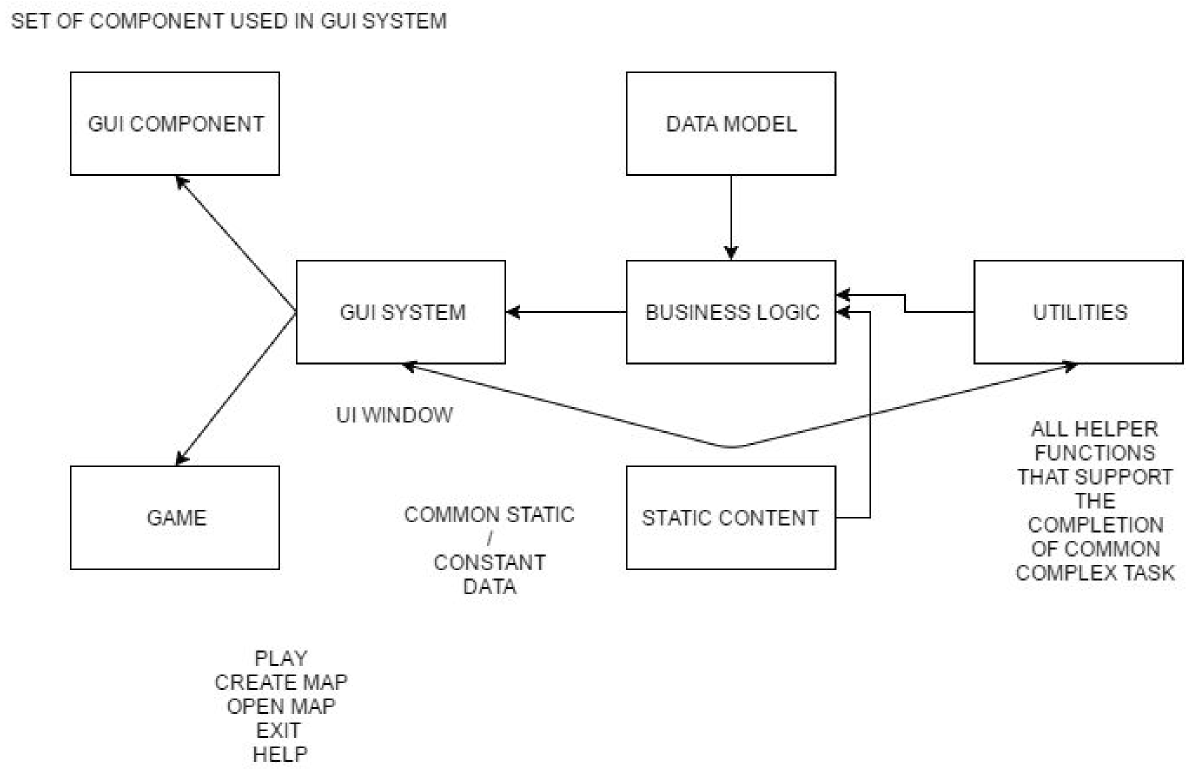
Code Snippet of singleton pattern in Game.java

4. **Strategy pattern :** In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern. In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object. In our application we use



UML Class diagram for strategy pattern

3.Code Organization


SET OF COMPONENT USED IN GUI SYSTEM
GUI COMPONENT
DATA MODEL
GUI SYSTEM
BUSINESS LOGIC
UTILITIES
UI WINDOW
ALL HELPER FUNCTIONS THAT SUPPORT THE COMPLETION OF COMMON COMPLEX TASK
GAME
COMMON STATIC / CONSTANT DATA
STATIC CONTENT
PLAY
CREATE MAP
OPEN MAP
EXIT
HELP

The code is organized into seven packages :

1. `com.app.towerDefense.guiSystem`
   This package contains :
   - the Game class which is the game entry point and is used to play the game
   - the Map Editor class which is used to edit maps.
2. `com.app.towerDefense.guiComponents`
   This package contains the UI components like buttons,panels, jmenubars, filechooser. It provides logic for creating the user interface for playing the game.

3. `com.app.towerDefense.gameLogic`
   This class contains game logic for map like creating, verifying and validating maps.
4. `com.app.towerDefense.models`

This class contains the data models for the different components of the game like tower, critter, map.

5. `com.app.towerDefense.staticContent`
   It contains all the static or constant data which are frequently used throughout the application. Declaring such data at one place makes it easy to maintain the code as we need to declare them once and we can call it in other classes. For example if we want to change the images in the game we just make changes under this package and it will reflect where those variable are being used.
6. `com.app.towerDefense.test`
   All unit test cases are defined under this package
7. `com.app.towerDefense.utilities`
   Under this package we define our helper functions which can be called throughout the application to read a file, save a file, serialize and deserialize a json object, base64 encoding and decoding.

4.Testing

Unit testing is used in our application to test important methods using JUnit Framework. Since its time consuming to test all the methods in the system as the large number of methods in our system is large, only selected sets of methods were tested and the method chosen test the most important aspects of the code.Among the classes that were tested are model classes for map, tower and critter. Important logic for map validation, critter path calculation were tested rigorously.

5.Coding Standard

To build enterprise Java applications, which are reliable, scalable and maintainable, it is important for development teams to adopt proven design techniques and good coding standards. The adoption of coding standards results in code consistency, which makes it easier to understand, develop and maintain the application. In addition by being aware of and following the right coding techniques at a granular level, the programmer can make the code more efficient and performance effective.

Commenting

Single Line Comments

```
/* Handle the condition. */
//if (bar > 1) {
```

Multiple Line Comments

```
/*
 * Here is a block comment.
 */
```

Documentation Comments

```
/**
 * The Example class provides ...
 */
```

Class Declaration

Java source are named as *. java while the compiled Java byte code is named as *.class file. Each Java source file contains a single public class or interface. Each class must be placed in a separate file. This also applies to non-public classes too.

Variable Declaration

Class Variables - All class variables are declared in camelcase.

Example :

```
public String mapSecret;
```

Local Variables- All local variables are declared in camelcase.

Example :

```
TowerModel tempTM = null;                    ;
```

Parameters - All parameter are prefixed with new_.

Example :

```
public TowerModel buyTower(int new_towerID){}
```

Constants - All constants are written in capital letters and the ones which need to be reused are declared in the file AppicationStatics.java

Example :

```
public static final int CHILD_POPUP_WINDOW_WIDTH = (int)
WINDOW_WIDTH - 100;
```

File Organization

All classes are declared as part of a package.

Example:
Models are organized under the package :
com.app.towerDefense.models

Method Names

- All method names are written in camelcase. All method names begin with a strong action verb.
  Example :
  public TowerModel buyTower(int new_towerID){}

- Use of the prefixes get and set for getter and setter methods.
  Example :
  public String getPlayerName() {}
- If the method returns a boolean value, use is or has as the prefix for the method name.
  Example :
  public boolean isEntryDone() {)

6.Reference

http://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm