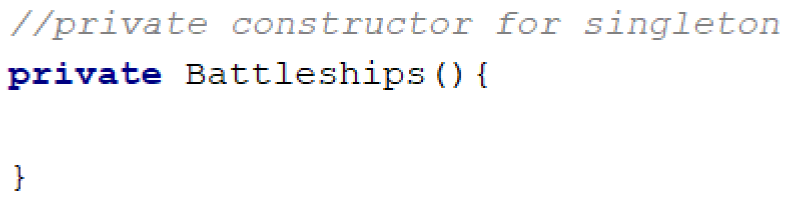
Battleships

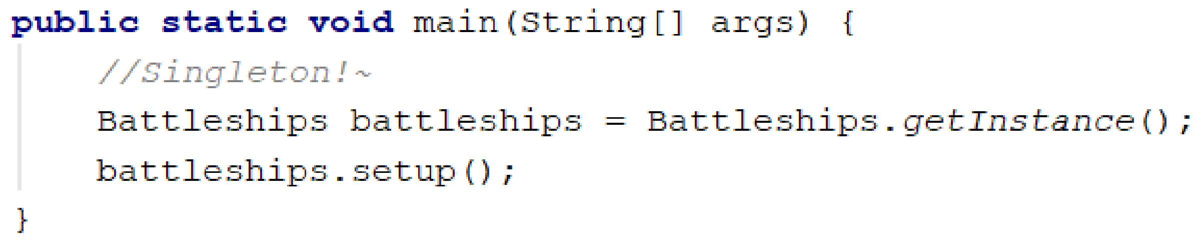
Using Design Patterns

Singleton

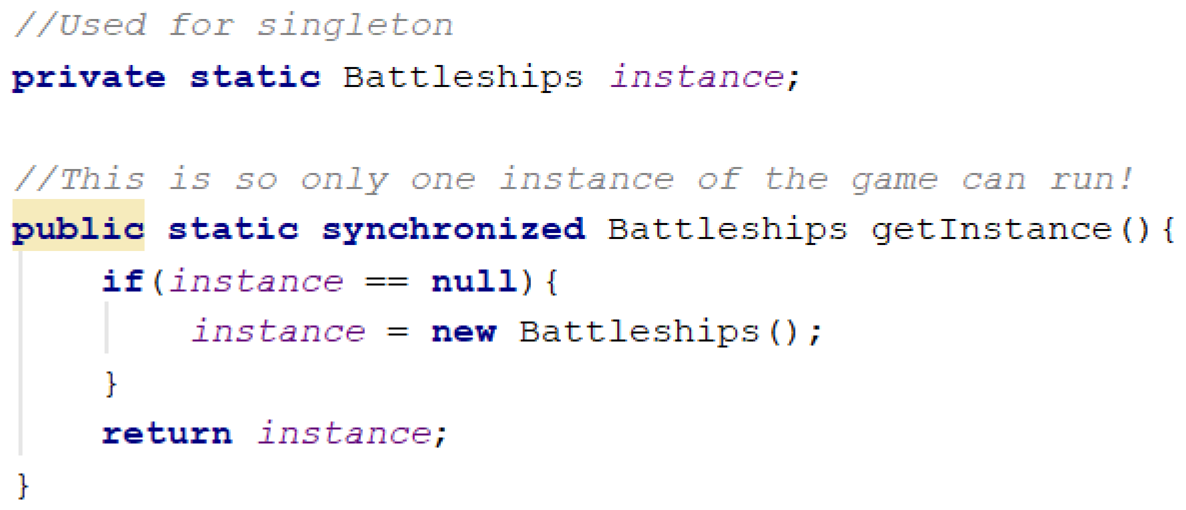
I used the singleton design pattern to ensure that only one instance of the game can be run at each time.



```
//private constructor for singleton
private Battleships(){

}
```



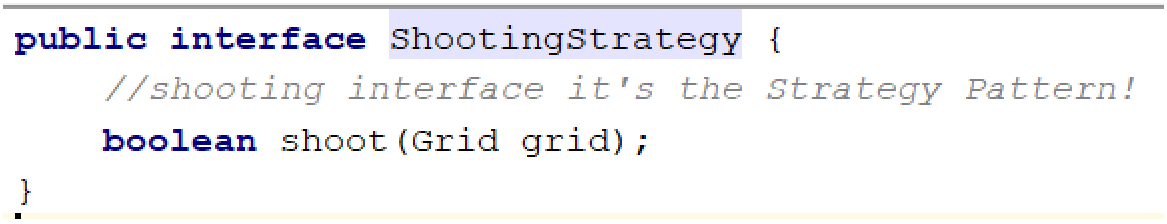
```
public static void main(String[] args) {
    //Singleton!~
    Battleships battleships = Battleships.getInstance();
    battleships.setup();
}
```



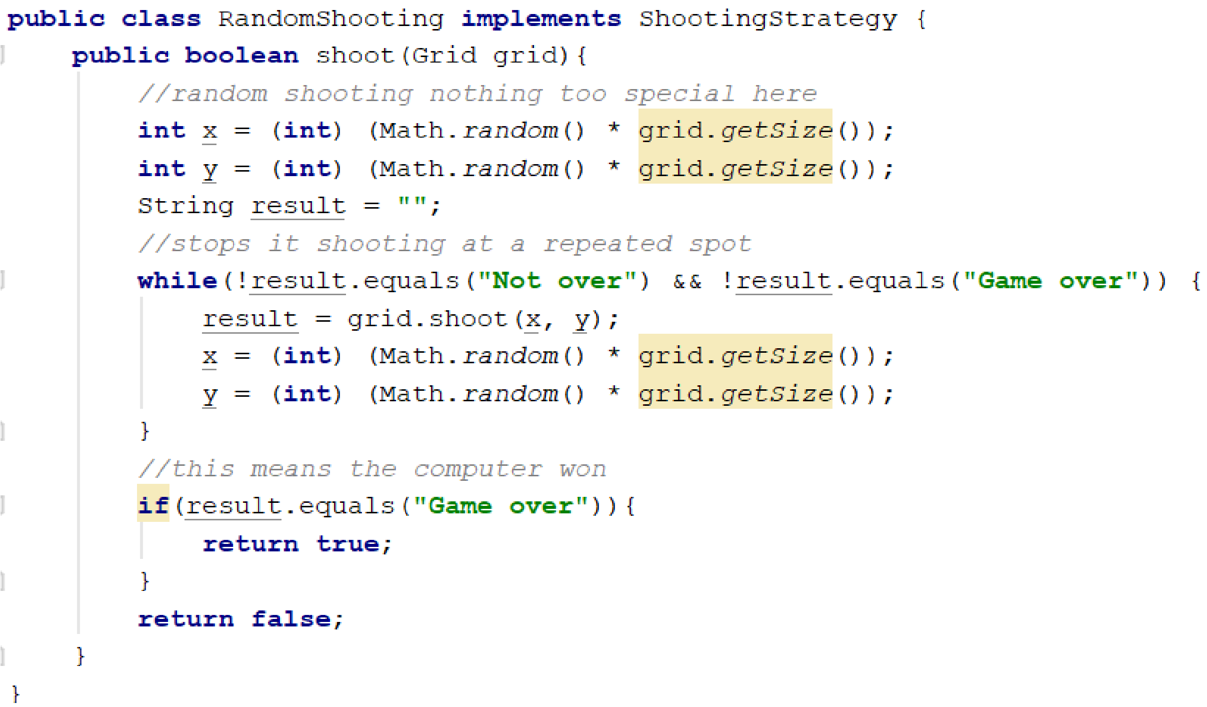
```
//Used for singleton
private static Battleships instance;

//This is so only one instance of the game can run!
public static synchronized Battleships getInstance(){
    if(instance == null){
        instance = new Battleships();
    }
    return instance;
}
```

Strategy

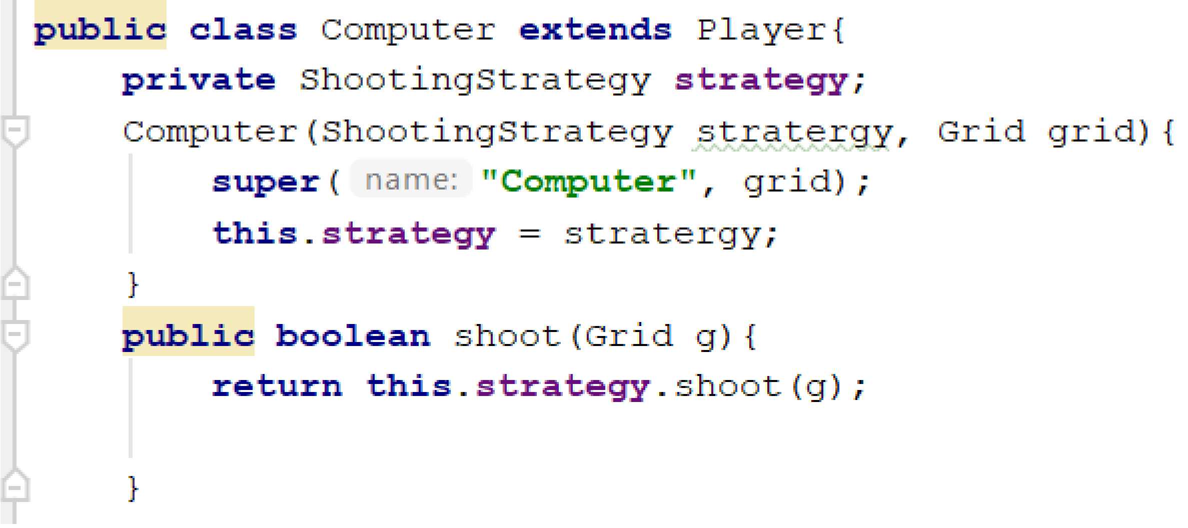
I used the Strategy design pattern to allow the choice of different shooting methods for the computer player. This is useful in case there are different difficulty levels in the future currently I only implemented random shooting.



```
public interface ShootingStrategy {
    //shooting interface it's the Strategy Pattern!
    boolean shoot(Grid grid);
}
```



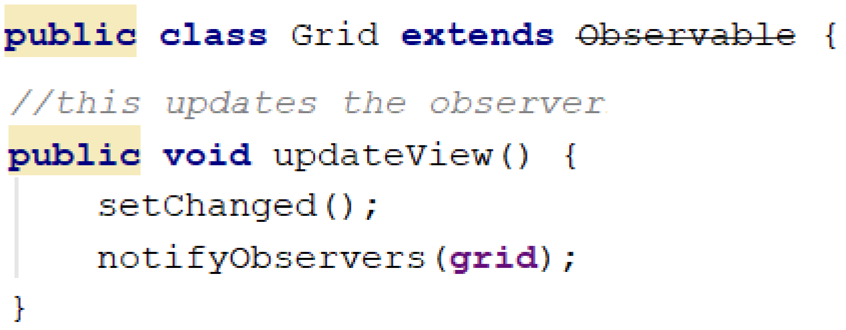
```
public class RandomShooting implements ShootingStrategy {
    public boolean shoot(Grid grid){
        //random shooting nothing too special here
        int x = (int) (Math.random() * grid.getSize());
        int y = (int) (Math.random() * grid.getSize());
        String result = "";
        //stops it shooting at a repeated spot
        while(!result.equals("Not over") && !result.equals("Game over")) {
            result = grid.shoot(x, y);
            x = (int) (Math.random() * grid.getSize());
            y = (int) (Math.random() * grid.getSize());
        }
        //this means the computer won
        if(result.equals("Game over")){
            return true;
        }
        return false;
    }
}
```



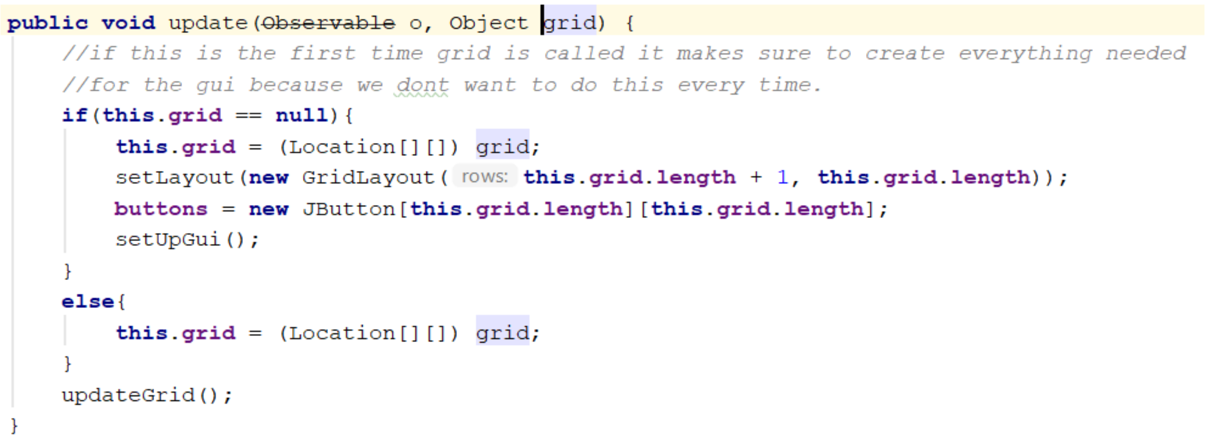
```
public class Computer extends Player{
    private ShootingStrategy strategy;
    Computer(ShootingStrategy stratergy, Grid grid){
        super( name: "Computer", grid);
        this.strategy = stratergy;
    }
    public boolean shoot(Grid g){
        return this.strategy.shoot(g);

    }
```

Observer

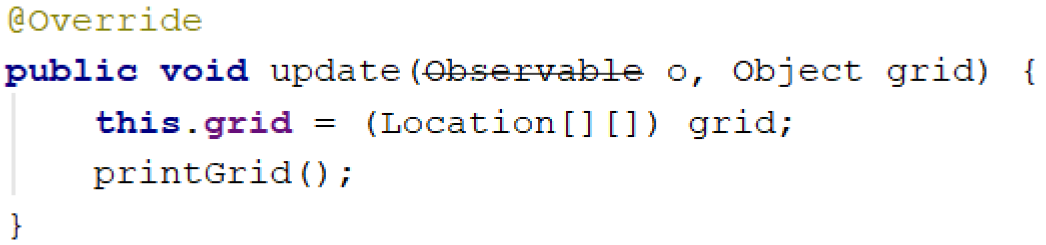
I used the observer design pattern to allow different views of the game either on command line view using text characters or also a graphical view in another window. More views could also be easily added as this pattern is implemented.



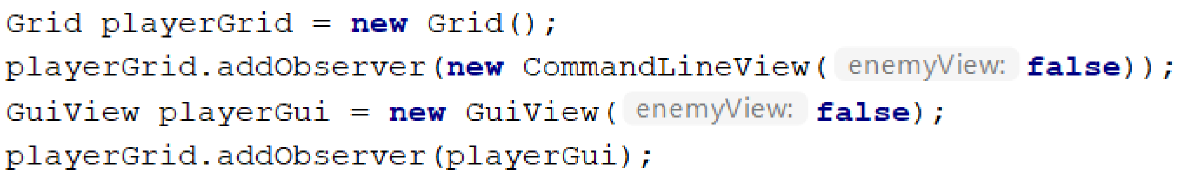
```
public class Grid extends ~~Observable~~ {
//this updates the observer
public void updateView() {
    setChanged();
    notifyObservers(grid);
}
```



```
public void update(~~Observable~~ o, Object grid) {
    //if this is the first time grid is called it makes sure to create everything needed
    //for the gui because we dont want to do this every time.
    if(this.grid == null){
        this.grid = (Location[][]) grid;
        setLayout(new GridLayout( rows: this.grid.length + 1, this.grid.length));
        buttons = new JButton[this.grid.length][this.grid.length];
        setUpGui();
    }
    else{
        this.grid = (Location[][]) grid;
    }
    updateGrid();
}
```

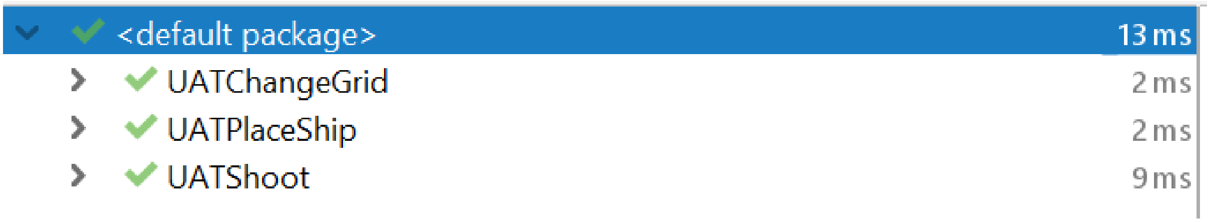


```
@Override
public void update(~~Observable~~ o, Object grid) {
    this.grid = (Location[][]) grid;
    printGrid();
}
```



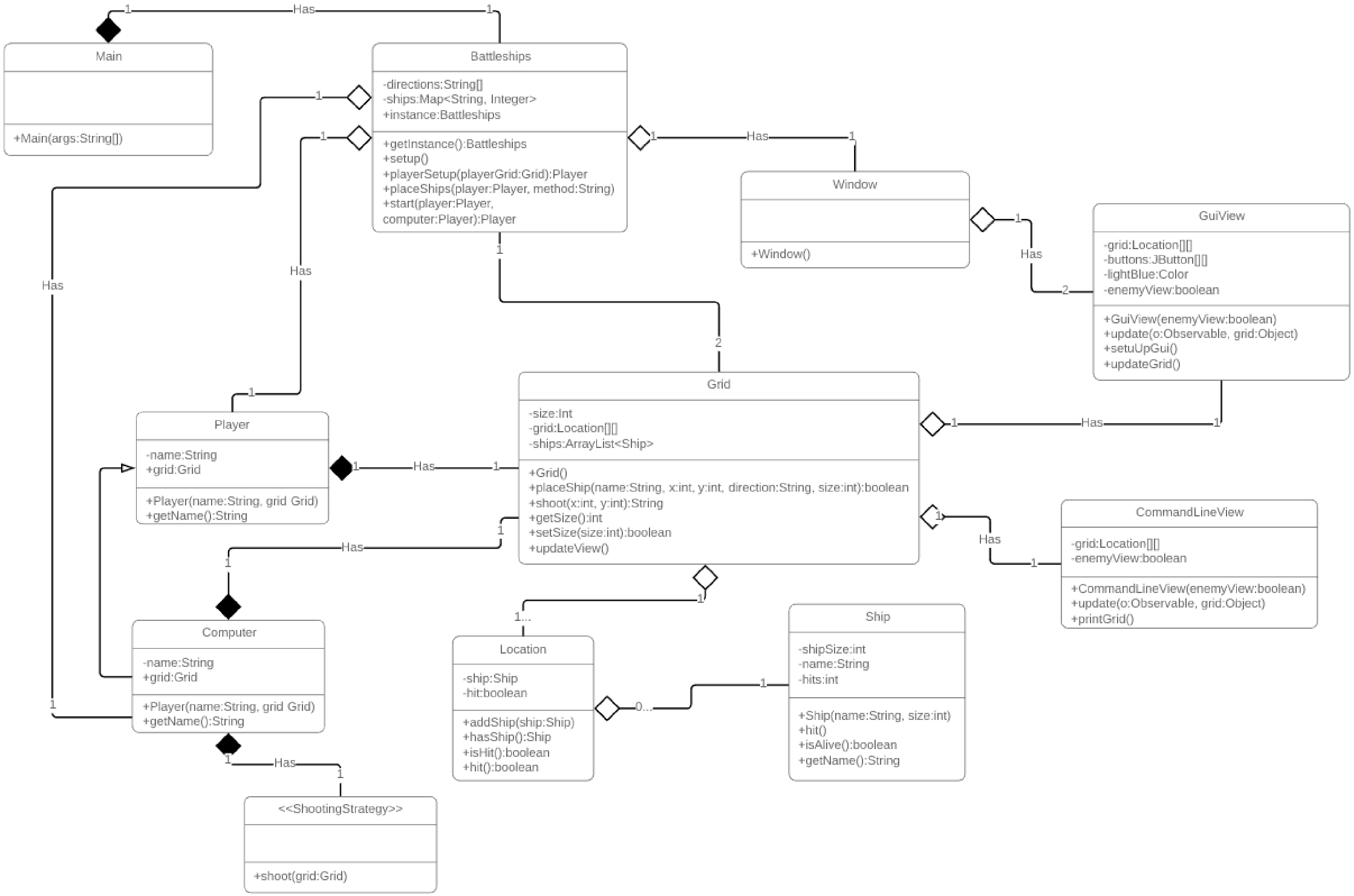
```
Grid playerGrid = new Grid();
playerGrid.addObserver(new CommandLineView( enemyView: false));
GuiView playerGui = new GuiView( enemyView: false);
playerGrid.addObserver(playerGui);
```

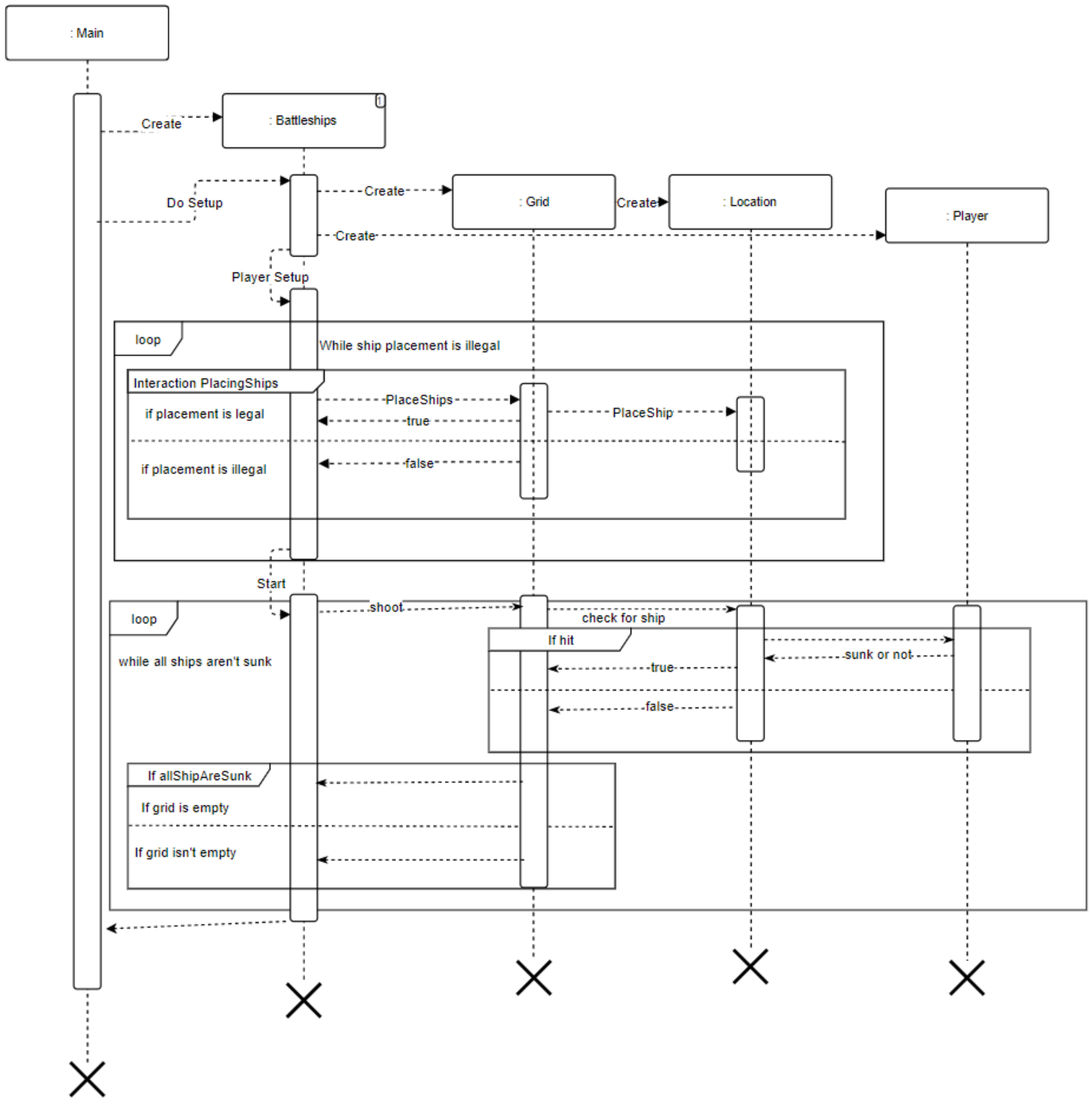
Tests



```
✓ <default package>    13 ms
  ✓ UATChangeGrid       2 ms
  ✓ UATPlaceShip        2 ms
  ✓ UATShoot            9 ms
```

You can see my code passes all the tests in the feature files I created.


Main
+Main(args:String[])
Battleships
-directions:String[]
-ships:Map<String, Integer>
+instance:Battleships
+getInstance():Battleships
+setup()
+playerSetup(playerGrid:Grid):Player
+placeShips(player:Player, method:String)
+start(player:Player,
computer:Player):Player
Window
+Window()
GuiView
-grid:Location[][]
-buttons:JButton[][]
-lightBlue:Color
-enemyView:boolean
+GuiView(enemyView:boolean)
+update(o:Observable, grid:Object)
+setuUpGui()
+updateGrid()
Player
-name:String
+grid:Grid
+Player(name:String, grid Grid)
+getName():String
Grid
-size:Int
-grid:Location[][]
-ships:ArrayList<Ship>
+Grid()
+placeShip(name:String, x:int, y:int, direction:String, size:int):boolean
+shoot(x:int, y:int):String
+getSize():int
+setSize(size:int):boolean
+updateView()
CommandLineView
-grid:Location[][]
-enemyView:boolean
+CommandLineView(enemyView:boolean)
+update(o:Observable, grid:Object)
+printGrid()
Computer
-name:String
+grid:Grid
+Player(name:String, grid Grid)
+getName():String
Location
-ship:Ship
-hit:boolean
+addShip(ship:Ship)
+hasShip():Ship
+isHit():boolean
+hit():boolean
Ship
-shipSize:int
-name:String
-hits:int
+Ship(name:String, size:int)
+hit()
+isAlive():boolean
+getName():String
<<ShootingStrategy>>
+shoot(grid:Grid)
Has
1
2
1...
0...


: Main
: Battleships
Create
Do Setup
Create
: Grid
Create
: Location
Create
: Player
Player Setup
loop
While ship placement is illegal
Interaction PlacingShips
if placement is legal
PlaceShips
true
PlaceShip
if placement is illegal
false
Start
loop
while all ships aren't sunk
shoot
check for ship
If hit
sunk or not
true
false
If allShipAreSunk
If grid is empty
If grid isn't empty