Easy to use Simple 2D Framework

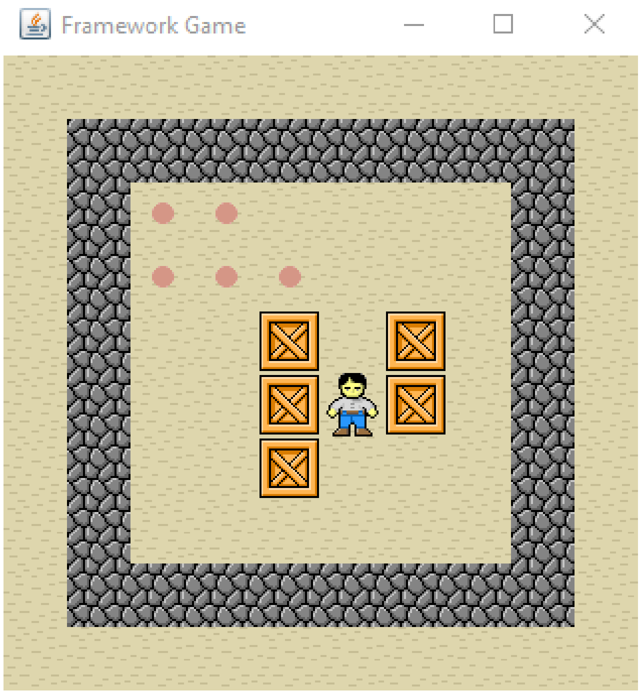
Mohammad Hammoud & Joakim Svensson

Introduction

The intended purpose of this framework is to provide ease of use for an application creator that wants to create a 2D Puzzle game. Some of the requirements for this Framework include, It should be able to create a grid map, display the grid together with images in a window. It should be able to handle inputs to alter the grid map, it needs to have a minimal application to demonstrate how to use it, have documentation, be created with OOP in mind.

Testing is also an inclusion that the framework needs, to be trustworthy and non-bug-prone to the person utilizing the framework. Debugging was used while it was being created, and to ensure the reliability of the code JUnit tests were created.

The choice was made to make the framework use polymorphism as it provides ease of use for the framework user. Another intent was to simplify the framework for someone who wouldn't be familiar with creating a simple 2D Puzzle game as that is the intended end-user for this framework.



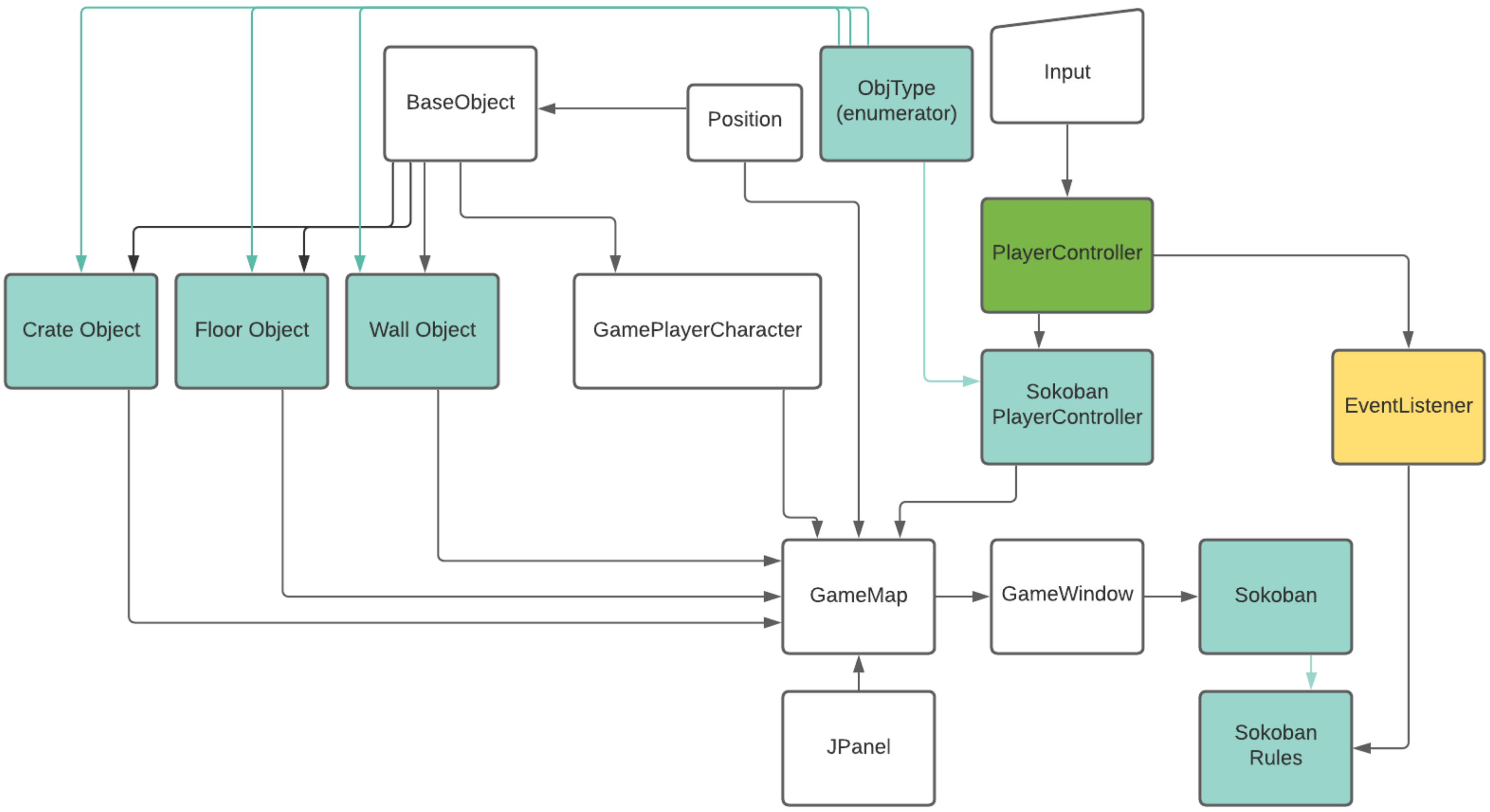
Design

The structure of the framework should provide you with enough logic out of the box to create a nice foundation for 2D games and to facilitate the end-user with the most ease to create an application using it. It should also provide some clear extension paths for the framework for example adding more objects, implementing your own movement logic. Once the user has learned the structure of the framework the intent is that it will be as frictionless as possible to use the framework. The framework also is tested to be sturdy and not prone to bugs otherwise, it would defeat its purpose.

The intention is to have the program have GameMap and GameWindow be what creates the game, PlayerController handling all the movements, Eventlistener to be there if additional requirements functionality is needed for the application built on top of the framework. BaseObject is intended to be the parent of all the Objects needed to create a game. There is also a base Enumerator included for extra convenience for the framework user. This is all done to have separate parts of the framework with intentional and clear meaning.

Setup

To use the framework for a game, certain parts will need to be created, these include Objects for your game, a PlayerController with movement logic overwritten in the movePlayer() method, a class that Implements the EventListener and the method eventTrigger() which is fired off each after each time any key is pressed, and after all the logic therein such as movePlayer(). To create the window you will want to first create an array of arrays of your BaseObjects (and children) which will be the game map, create a GameWindow object, GameMap object, createMap() in your GameMap object then add your JPanel GameMap and KeyListener PlayerController to the window. After that, you're all set and just create the objects you want and the movement logic that you want.



Teal is the application that is implemented, yellow and green are part of the Observer pattern (EventListener has an eventTrigger inside of PlayerController's KeyListener to allow the FrameworkUser to Override it and use it each time the player moves to do anything they like)

Intended use and Acknowledgements

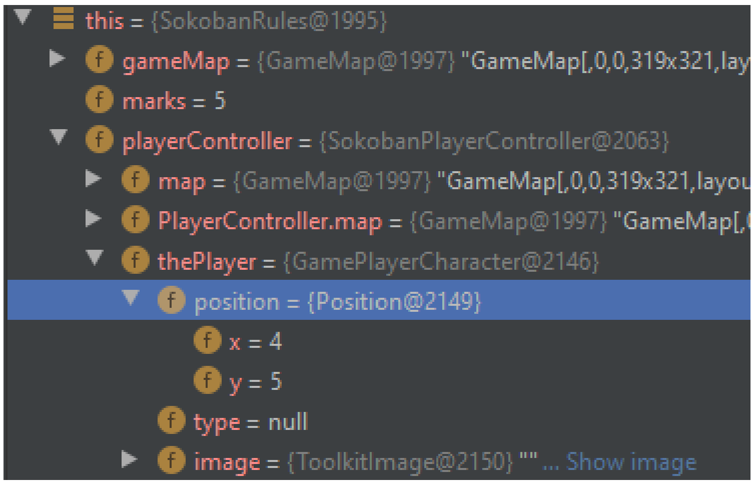
In the shown application a PlayerController child is used to handle the game between all the different types of Objects, this is suggested as a good way of reducing the dependency objects would get. All classes had their variables encapsulated, together with setters and getters.

The discussions about inheritance creating issues with encapsulation are also acknowledged, which could indicate that there are some issues with trying to do both in the framework and that it is possible that inheritance has been overused in this case as well "How Inheritance Weakens Encapsulation" (Weisfeld, 2005).

Testing

In order to have the framework created debugging was used as it was being created, after it had a working version the JUnit tests were created. The vast majority of the testing while the framework was being created was debugging using breakpoints and looking at variable values to see if they weren't working.

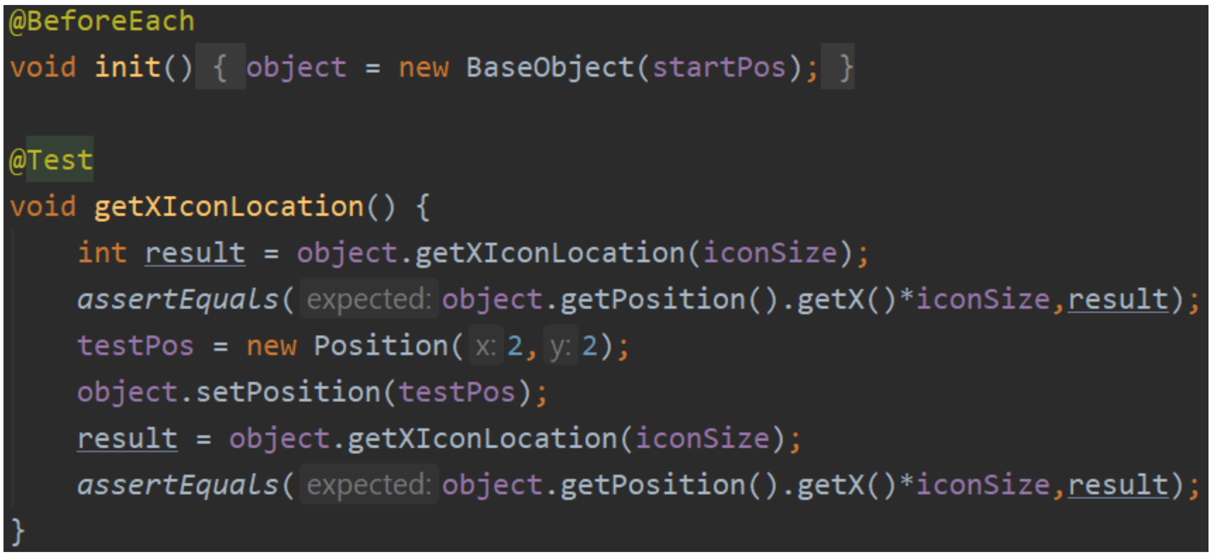
An example of one thing that happened is that PlayerController had a variable and also its child had a variable of the same type, the wrong one was being read, this was discovered fairly quickly using the debugging tool it was possible to see that directly in the debug window. It was fixed by making the child not have a player variable.



```
this = {SokobanRules@1995}
  gameMap = {GameMap@1997} "GameMap[,0,0,319x321,lay
  marks = 5
  playerController = {SokobanPlayerController@2063}
    map = {GameMap@1997} "GameMap[,0,0,319x321,layou
    PlayerController.map = {GameMap@1997} "GameMap[,
    thePlayer = {GamePlayerCharacter@2146}
      position = {Position@2149}
        x = 4
        y = 5
      type = null
      image = {ToolkitImage@2150} "" ... Show image
```

During the bug, this was the type of visual that was seen and how the bug was discovered. (location was set to the parent's player whenever the game was reset so the player couldn't move)

The JUnit tests are there in case of anything getting changed it's much easier to find where the issue is. The Base Object had each part of its setters and getters tested.

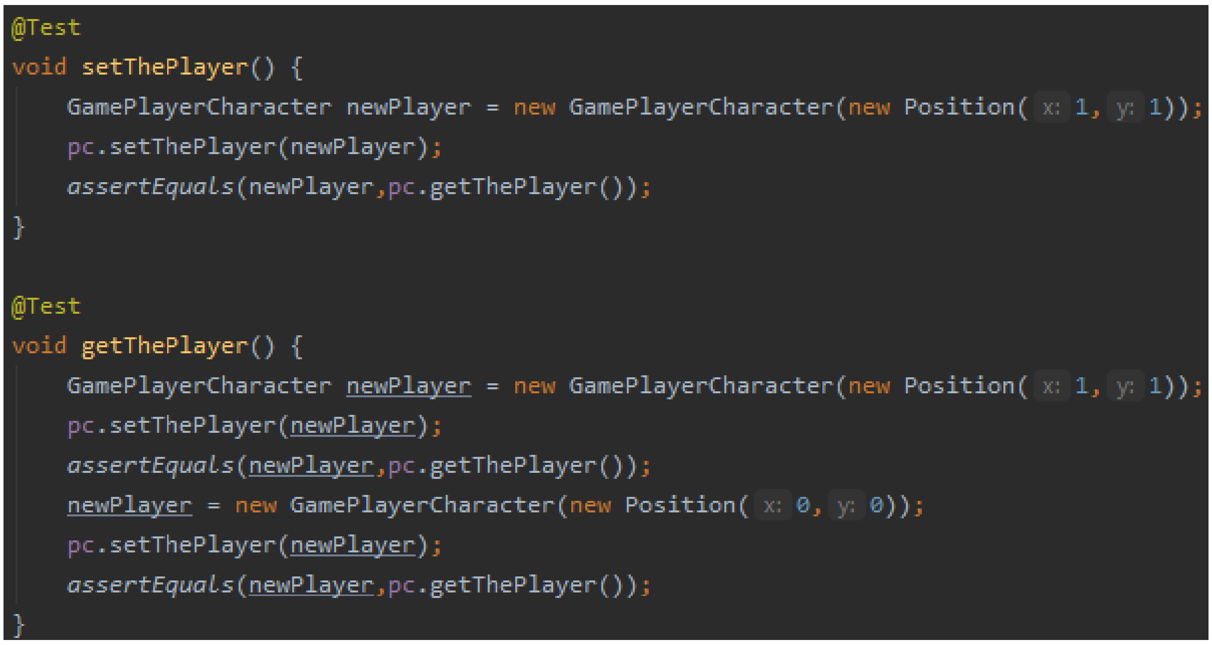


```java
@BeforeEach
void init() { object = new BaseObject(startPos); }

@Test
void getXIconLocation() {
    int result = object.getXIconLocation(iconSize);
    assertEquals( expected: object.getPosition().getX()*iconSize,result);
    testPos = new Position( x: 2, y: 2);
    object.setPosition(testPos);
    result = object.getXIconLocation(iconSize);
    assertEquals( expected: object.getPosition().getX()*iconSize,result);
}
```

Part of the tests on the BaseObject

Testing was also done on the PlayerController, testing the setters and getters there, if anything is changed so that they no longer work it will be caught in the JUnit tests.



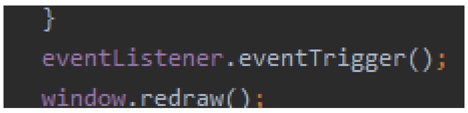
```
@Test
void setThePlayer() {
    GamePlayerCharacter newPlayer = new GamePlayerCharacter(new Position( x: 1, y: 1));
    pc.setThePlayer(newPlayer);
    assertEquals(newPlayer,pc.getThePlayer());
}

@Test
void getThePlayer() {
    GamePlayerCharacter newPlayer = new GamePlayerCharacter(new Position( x: 1, y: 1));
    pc.setThePlayer(newPlayer);
    assertEquals(newPlayer,pc.getThePlayer());
    newPlayer = new GamePlayerCharacter(new Position( x: 0, y: 0));
    pc.setThePlayer(newPlayer);
    assertEquals(newPlayer,pc.getThePlayer());
}
```

Setter and getter for the player's unit tests.

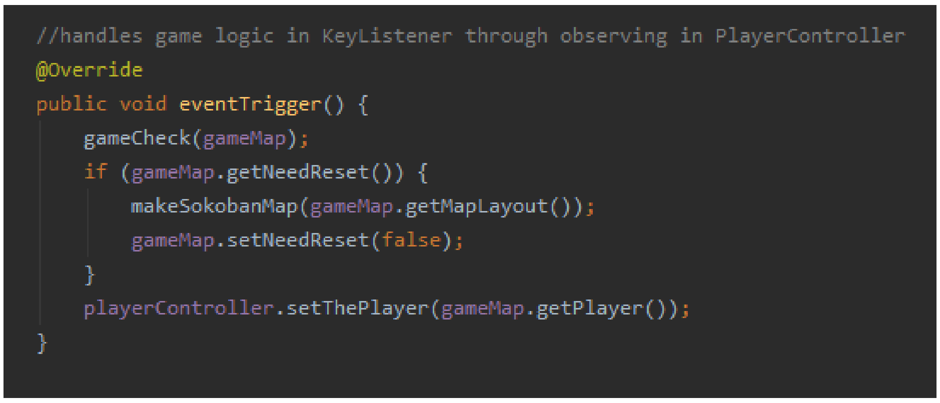
Notable parts

There were issues with getting the application to handle part of the game logic, for this, the observer pattern was introduced with the use of EventListener observing the PlayerController. This seems like a good use case according to (Refactoring.Guru., 2018)"Use the Observer pattern when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically."



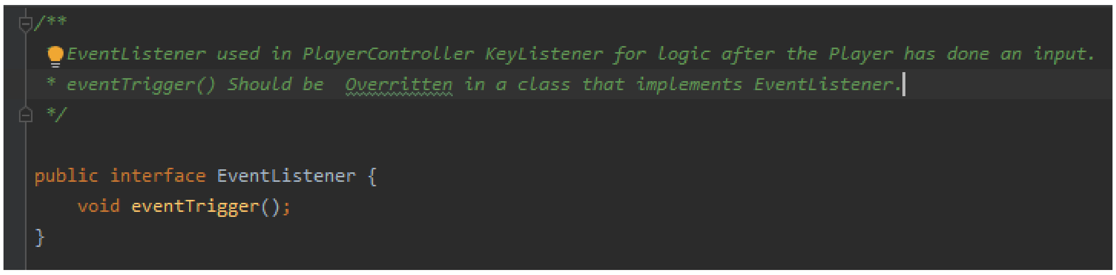
```
}
eventListener.eventTrigger();
window.redraw();
```

This event Trigger is located inside of the ActionListener in PlayerController to give the framework user the opportunity to perform checks each time the player has moved.



```
//handles game logic in KeyListener through observing in PlayerController
@Override
public void eventTrigger() {
    gameCheck(gameMap);
    if (gameMap.getNeedReset()) {
        makeSokobanMap(gameMap.getMapLayout());
        gameMap.setNeedReset(false);
    }
    playerController.setThePlayer(gameMap.getPlayer());
}
```

The trigger being used inside of the Sokoban rules



```
/**
 EventListener used in PlayerController KeyListener for logic after the Player has done an input.
 * eventTrigger() Should be  Overritten in a class that implements EventListener.
 */

public interface EventListener {
    void eventTrigger();
}
```

The Observer interface

Player Controller

In the example application Sokoban, a child is created to PlayerController, this is the recommended way to use this framework where you have the child be the controlling part of interactions between your different objects and your player.

This PlayerController handling is close to the Mediator Pattern and provides the user with an easier way of handling interactions between different Objects (Woyke, 2020) "The intent of the Mediator Pattern is to reduce the complexity and dependencies between tightly coupled objects communicating directly with one another."

The part in testing with debugging the PlayerController's variable player on the child and parent, was also a Notable part and will be remembered for a time in the future and less likely to be committed in the future.

Results

The intended functionality has been implemented into the framework, there are things that could've been added such as sound, a way to create maps while the framework is running (one window with possible objects, could be selected and then new ones could be created in the main window when pressed). Mouse clicking could be implemented, an easier way to make new levels could be created.

Considering all this the framework does allow the user to create a simple 2D puzzle game as intended. The framework has two classes that require children but they should be easy to create and be very easy to add logic to due to that they have clear things that you need to override and super, then you add your own game logic in it.

The framework's class GameMap could be the hardest part to understand and would need some testers to see if it needs to have improved clarity. Classes like PlayerController also could provide standard functionality between BaseObjects to make the framework easier to use for beginner programmers.

Github

Github proved to be very useful utilizing branches and doing pushing commits when the framework works as intended, then changes are made safely, if the changes don't work it's possible to stash some parts or all changes. It's also easy to handle version control while collaborating and it's easy to create pull requests. Github will be a tool used for future applications that we create as it proved itself very resourceful and time-saving once set up.

Some errors made using Github were non-ideal naming, not enough description text, too few commits, not committing when the branch is working as intended.

Bibliography

Bibliography

Refactoring.Guru. (2018). *Observer*. refactoring.guru. Retrieved 08 13, 2021, from https://refactoring.guru/design-patterns/observer

Weisfeld, M. (2005). *Encapsulation vs. Inheritance*. Encapsulation vs. Inheritance. https://www.developer.com/design/encapsulation-vs-inheritance/

Woyke, K. (2020, 8 5). *The Mediator Pattern in Java*. The Mediator Pattern in Java. Retrieved 8 13, 2021, from https://www.baeldung.com/java-mediator-pattern