Fundamental Object Oriented Programming 2021
Final Project

李宜璟 B07401012

陳冠廷 B07902025

陳捷堂 B07902018

鄭仲語 B07808005

Summer 2021

1 Assignment of responsibilities

2 Object Oriented Design

In this section we'll provide a brief illustration for OOD in our final project **Make It On Time**. Although we won't go through the detailed design and responsibility for each class, we do provide a simplified version of class diagrams, which demonstrate the relations between our classes.

There are three major division of classes in our design:

1. The **model** of the world, which includes all sorts of objects inside the game world and encapsulates their relations.

2. The **controller**, which is the interface between player input and the game world.

3. The **View**, which is responsible for displaying the world to the player.

2.1 Model

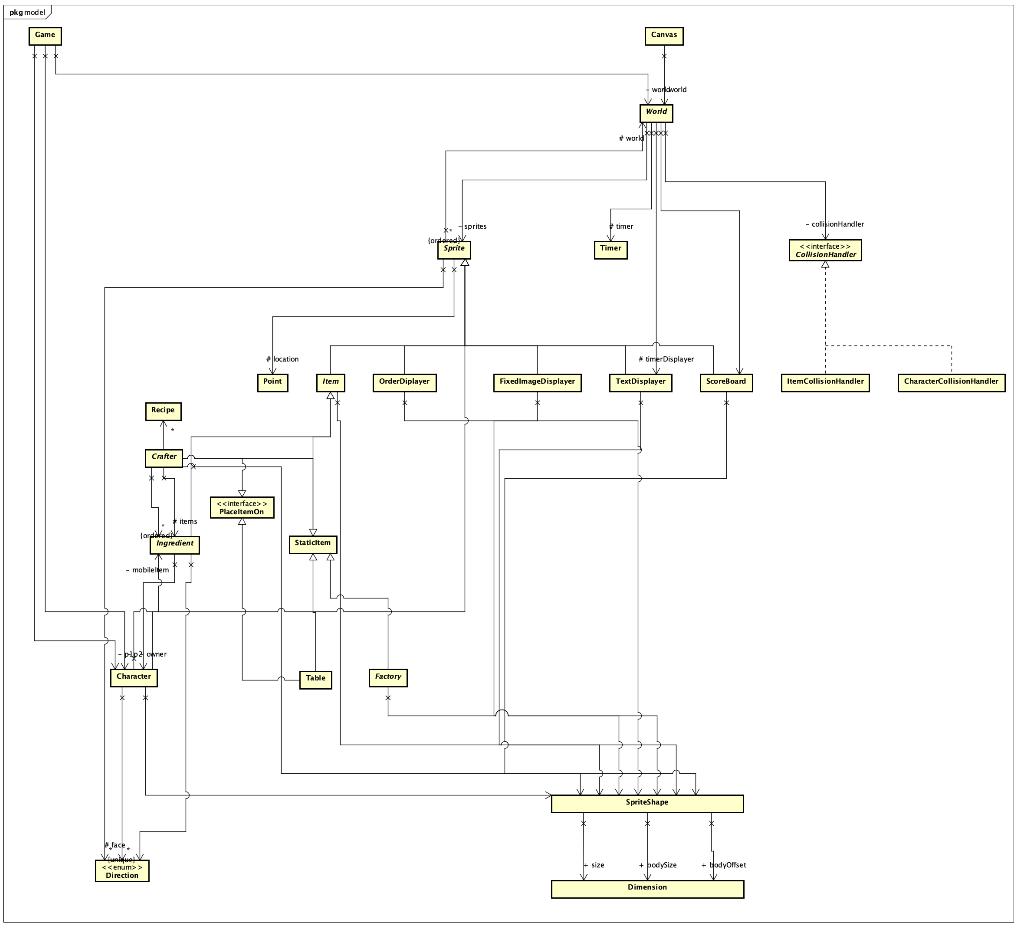
The model of our game world is implemented in the class file **World**, which contains the following attributes. as shown in *Figure 1-(a).*

1. *Sprite*, a collection of the **Sprite** residing in our game world, they can further be classified into several classes

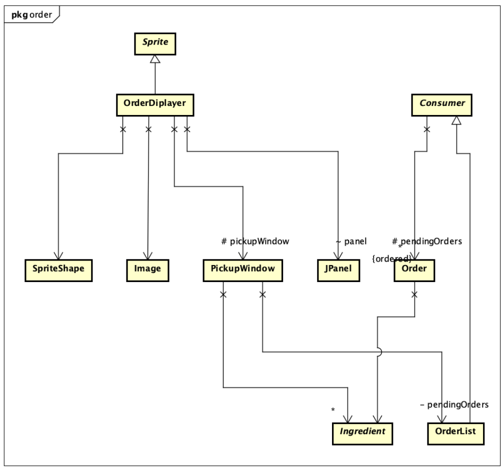
   (a) **Character**, which can be controlled by players and is able to interact with other items, e.g. picking and releasing **MobileItem**, and collision with **StaticItem**.

   (b) **Item**, which includes all sprites other than player in our game, divided into

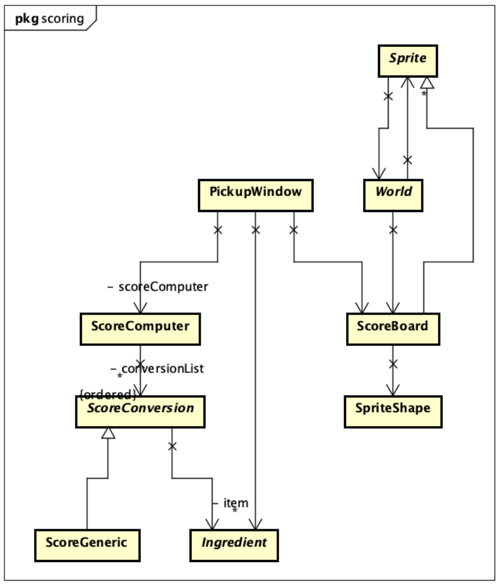
      i. **MobileItem**, or equivalently **Ingredient** in our design, which can be picked up and moved with **Character**. Also they can interact with **StaticItem** to be discarded and crafted into a new **Ingredient**.



(a) A simplified class diagram of **World**.



(b) A simplified class diagram of **OrderList**.



(c) A simplified class diagram of **Scoreboard**.

Figure 1: Class diagrams of the game model in this project.

   ii. **StaticItem**, which isn't mobile but may be equipped with different function, allowing them to interact with **MobileItem** and **Character**. (See the following subsection for details)

2. **OrderDisplayer** is a special sprite, which shows the incoming order, requiring character to deliver them.

3. **Scoreboard** is also a special sprite, demonstrating the score player had gotten via controlling **Character** to complete the orders.
   The simplified class diagram of **OrderDisplayer** and **Scoreboard** are shown in *Figure 1-(b), (c)*, and worth noting is that they can interact with **Ingredient** (or completed order the player make) via a static item **PickUpWindow** is our design.

4. **Timer**, which counts down the game time.

5. **TextDisplayer** and **FixedImageDisplayer**. As the name suggested, these items can display text and images in the **World** as background.

6. **CollisionHandler**, which is responsible to handle the overlap the rigid body between sprites.

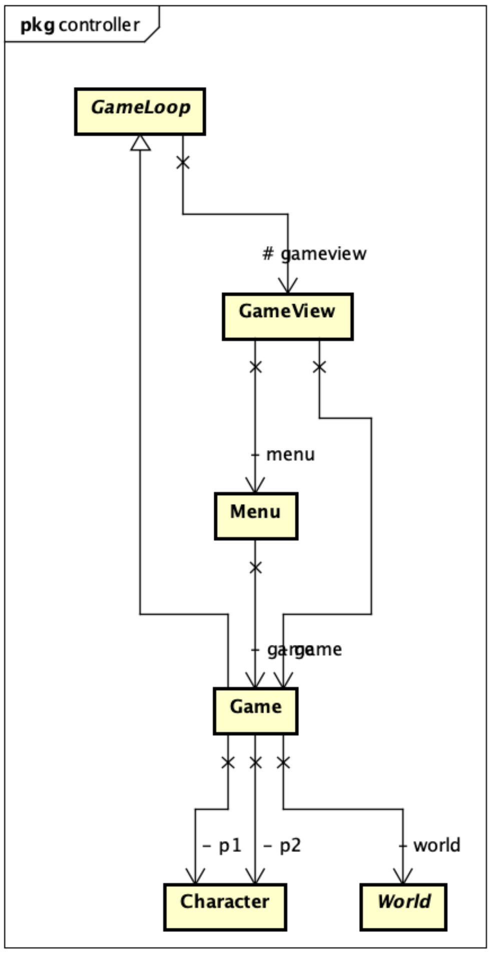
2.1.1 Detail about Item Relations

Several type of static items are implemented with some properties allowing them to interact with mobile item, or **Ingredient**. These includes,

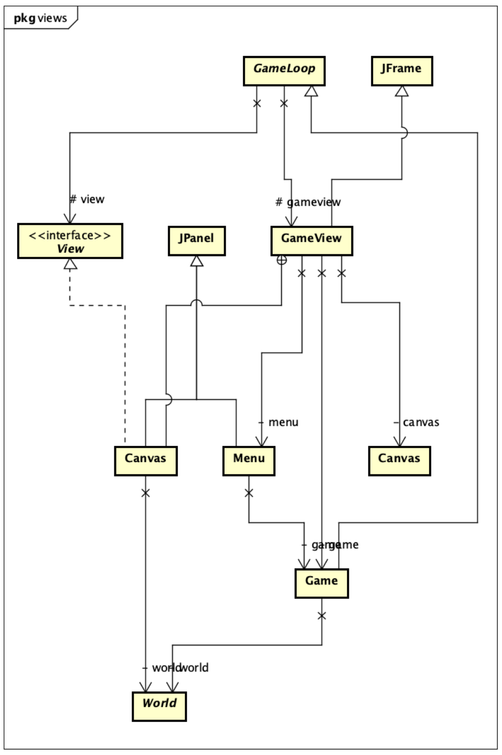
1. **Factory**, an abstract class, which encapsulate the function of limitlessly produce ingredient.

2. **Crafter**, an abstract class, which encapsulate the function of transforming ingredient(s) into new ingredient. Inside the **Crafter** are several **Recipe** attributes enclosing the transform formula.

3. **PlaceItemOn**, an interface which allows ingredient(s) to be released on this item.



(a) A simplified class diagram of game controller in this project.



(b) A simplified class diagram of game view in the project.

Figure 2: Class diagrams of the game control and view in this project.

2.2 Controller

As shown in *Figure 2-(a)*, class **Game** implementing **GameLoop** operates the game flow, and **GameView** allows player to interact with **Game** and **Menu**, which in terms alters the **Character** and **World**.

2.3 View

As shown in *Figure 2-(b)*, class **GameView** utilizes **Canvas** to render the world and the action of the character in it.

3 Advantage of Design

1. Open-Closed Principle (OCP) We achieve OCP in food ingredient, recipe, factory, static item, world. One can add any ingredient he/she by extending the Ingredient class by

```java
public class IngredientOneWants extends Ingredient {
    public IngredientOneWants(Point location, SpriteShape shape) {
        super(location, shape, " IngredientOneWants ");
    }
}
```

One can creates any recipe by extending the abstract ConcreteRecipe class

```java
public class  SomeIngredientRecipeOneWants extends ConcreteRecipe {
    public ApplepieRecipe(SpriteShape productShape) {
        super(productShape, "ingredientNeeded1", "ingredientNeeded2",
           ... );
    }
    protected Ingredient getResult() {
        return new SomeIngredientOneWants(new Point(0, 0), productShape
           );
    }
}
```

One can creates any ingredient factory by extending the abstract factory class

```java
public class IngredientFactoryOneWants extends Factory {

    public IngredientFactoryOneWants(Point location, SpriteShape shape,
         SpriteShape productShape) {
        super(location, shape, productShape, "eggbasket");
    }

    @Override
    public MobileItem produceItem() {
        RawEgg newItem = new IngredientOneWants(new Point(0, 0),
           productShape);
        this.world.addSprite(newItem);
        return newItem;
    }
}
```

One can creates any static item by extending the abstract StaticItem class public class

```java
public class StaticItemOneWants extends StaticItem {

    public StaticItemOneWants(Point location, SpriteShape shape) {
        super(location, shape);
        ImageRenderer imageRenderer = new ItemImageRenderer(this);
        idle = new WaitingPerFrame(4, new Idle(imageStatesFromFolder("
           assets/item/staticItemOneWants", imageRenderer)));
    }
}
```

Finally, One can also creates any static item by extending the abstract World class public class.

2. 沒有用到其他 package, dependency 很乾淨

4 Disadvantage of Design

4.1 File IO

Beacuse we use java.IO.File to access our assets, it is nearly impoosible to package the whole game as a file. A proper way to load image from .jar file is to use getClassLoader().getResourceAsStream(). However, since the utility is design to load state by all file name, it is not likely possible to do so.

4.2 Design limit

We us java AWT as our GUI engine, and because it is quite old package, some of our design is limited by its ability.

4.3 Doa Dao Dao

5 Packages Utilization

6 How to Play

玩家進入遊戲後，可以選擇遊玩人數及遊戲世界，目前提供一至二人進行遊戲，並有四個遊戲地圖供玩家選擇，玩家透過點擊視窗上的遊玩人數及遊戲世界的數字進行可做選擇，選擇完畢後點擊 START 按鈕即可進入遊戲。

在進入遊戲後，可見遊戲本體在視窗中央，視窗右方從上到下分別顯示遊戲倒數、玩家目前分數及食譜，而視窗下方則顯示目前的訂單，透過達成訂單要求即可獲得分數。

玩家一可利用鍵盤按鍵 W, S, A 及 D 分別進行上、下、左及右的移動，並可利用鍵盤按鍵 Q 觸發食材的提取, 並以鍵盤按鍵 E 觸發食材的放置; 玩家二可利用鍵盤按鍵 I, K, J 及 L 分別進行上、下、左及右的移動，並可利用鍵盤按鍵 U 觸發食材的提取, 並以鍵盤按鍵 Q 觸發食材的放置。

遊戲中，可進行食材提取的物件共有 7 個, 分別是 EggBasket, BreadBasket, CheeseBlock, SpinachGarden, PieBox, FruitBasket 及 TomatoBasket, 除了 FruitBasket 會隨機給予 Apple, Banana 及 Orange 其中一者之外, 其他的物件都只會進行單一一種食材的生成。

若有 4 種類型物件可進食材的放置, 分別是可臨時放置至多一項食材的 WoodPlatform, 可進行食材棄置的 TrashCan, 用於製作產品的 ApplePieStove, SaladBowl, SandwichMaker 與 FriedEggStove 及提交最終產品的 PickupWindow。

玩家可參考右方的食譜進行產品的製作，並將訂單相對應的產品放置於 PickupWindow，若所放置的食材滿足其中一項訂單，則可獲得相對應的成績，並可見 ScoreBoard 即時更新成績，倘若所放置的食材不滿足任何一項訂單，則會受到分數懲罰，每次將倒扣 10 分直到分數為 0。

訂單會隨時間增加，至多累積至 5 筆訂單。

當時間倒數至 0 時, 遊戲會強制中止並顯示玩家的分數, 玩家可透過點擊 Play Again 回到 Menu 再次進行遊戲。