Sushi Hero Documentation

Justin Christopher Abarquez

Mariko Ariane Vecta Sampaga

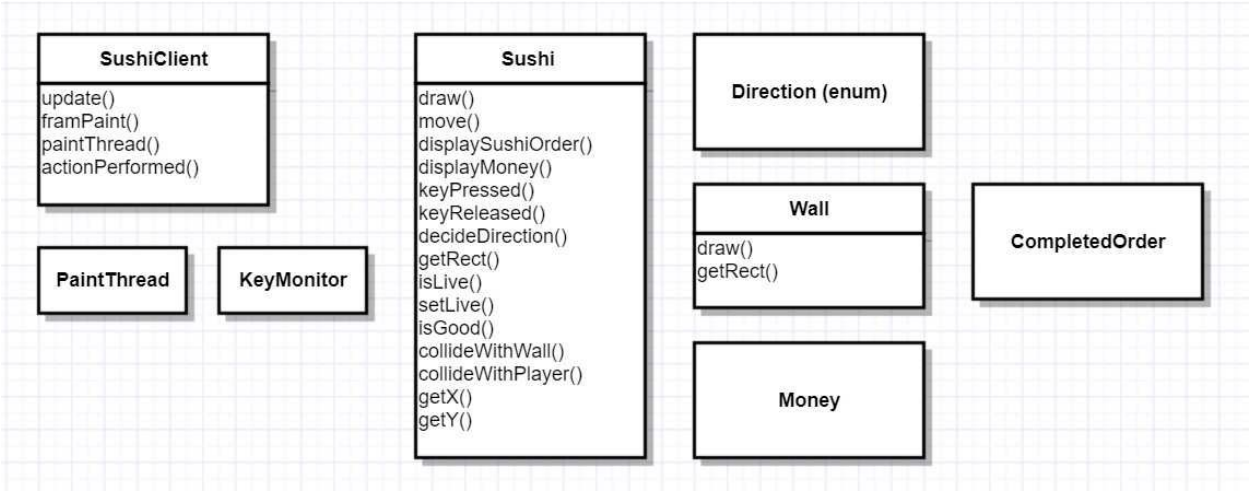
CSC 413.01 Software Development

San Francisco State University

December 22, 2017

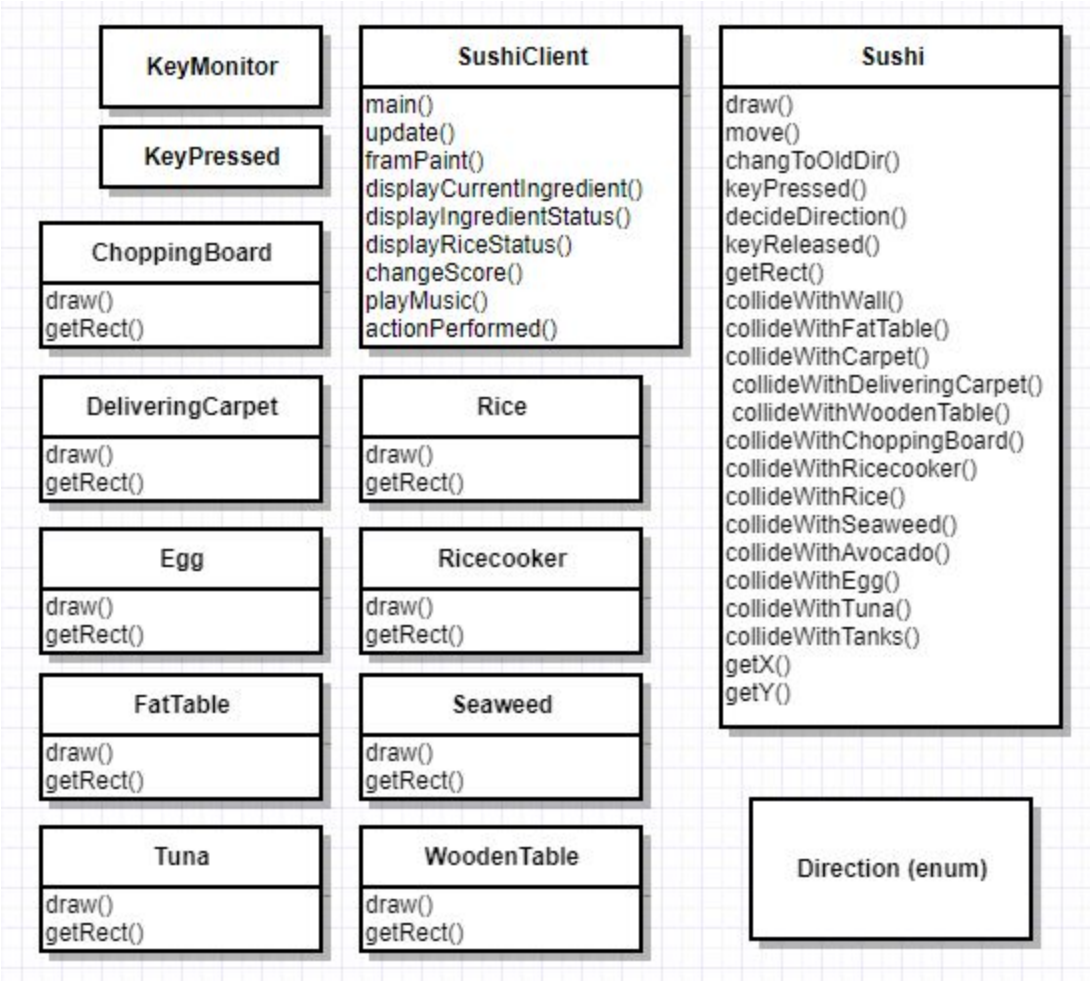
Table of Contents

1. Class Diagrams



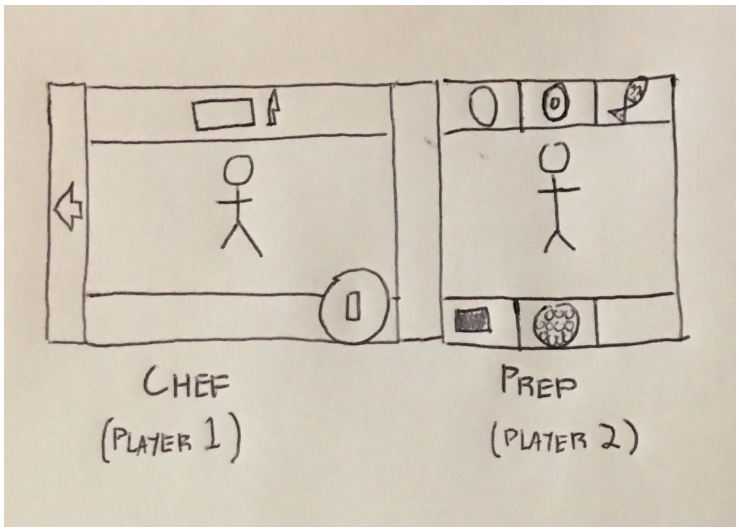
In our original, tentative class diagram (above), we initially assumed all classes to be independent of any hierarchy. Our **Sushi Client** is what starts the program, and **Sushi** and **Wall** contain constructors for each object resource (we eventually would add more). Sushi contains more fields and fields than the others since it represents the player objects. We also used a **Direction** enumeration to consider the different states of which keys are pressed through the virtual keyboard. We originally had a **Money** and **CompletedOrder** class to contain the stacks to be implemented, but as of now that data is stored in our client.

Here is a most updated UML diagram of our game:



2. Objective

We sought to make a 2D restaurant simulator in Java. The player must acquire the correct ingredients for specific sushi orders that are displayed in the top right corner. The player controls both the sushi chef (on the left) and prep (on the right).



Our game also relies a bit on memory as it requires some intuitive knowledge of what ingredients are in a specific sushi roll (i.e, avocado sushi = avocado + seaweed + rice). The prep begins with 'grabbing' (or colliding) with the correct ingredient and delivering them to the middle wooden table. After the player has grabbed the correct ingredient and collided with the wooden table, the chef will now

have it and be able to prepare it by either chopping it (at the chopping board) or cooking rice (at the rice cooker). After all correct ingredients are acquired and prepared, the chef can then finally deliver them to the counter (at the cashier on the left). A score is then determined above. If the player has succeeded, he/she will have some dollar amount added to their score. Each sushi roll will cost differently (to keep things interesting, we'll use exaggerated SF sushi prices):

Sushi Rolls

- Tamago (Egg Sushi) **$8**
  - Rice
  - Seaweed
  - Egg
- Avocado Sushi **$12**
  - Rice
  - Seaweed
  - Avocado

- Salmon-Avocado **$18**
  - Rice
  - Seaweed
  - Avocado
  - Salmon

However, if an incorrect order is delivered, their score will receive some kind of deduction.

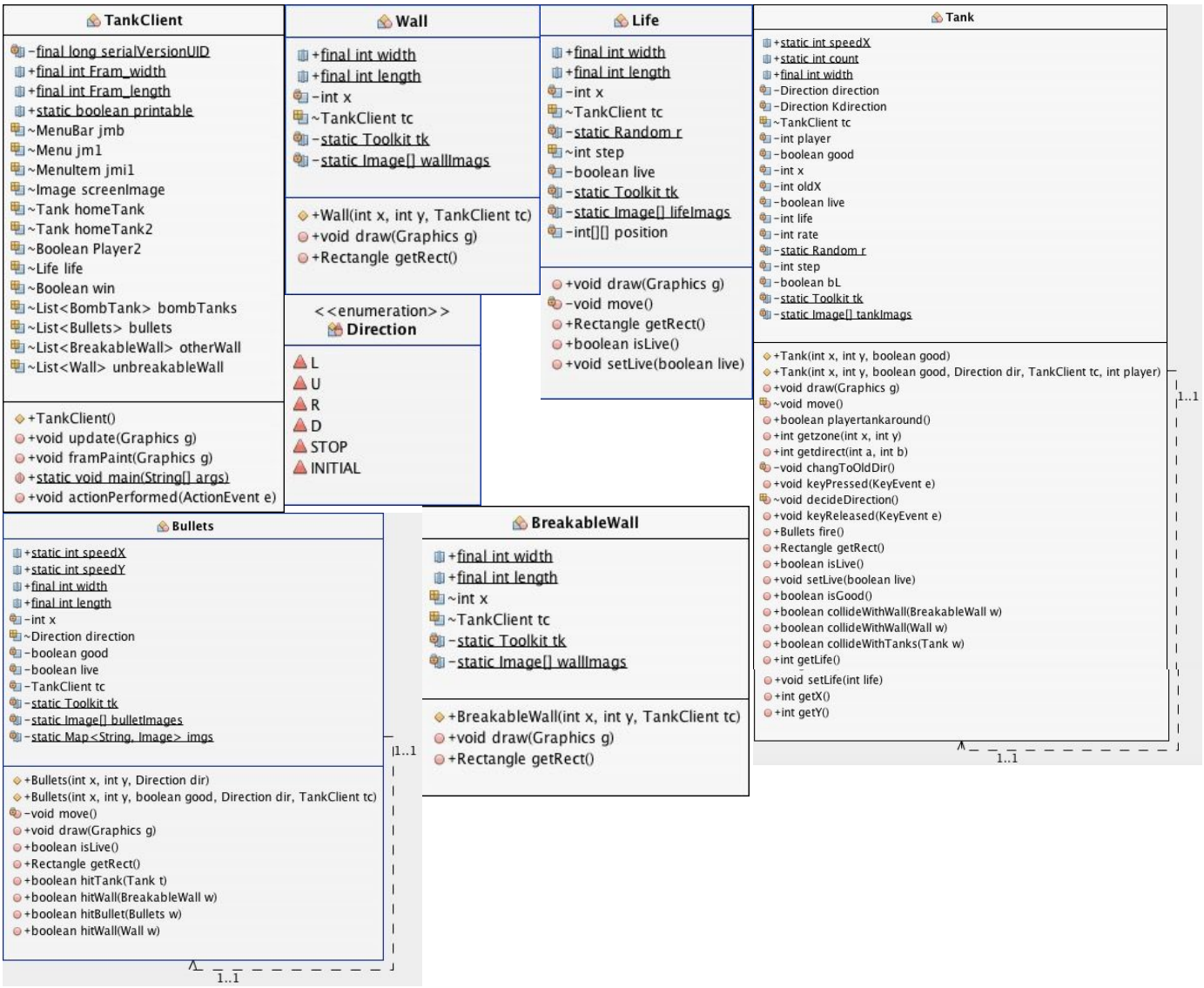
Below, are a list of tasks we gave ourselves:

- Intersect Methods: For the time being, instead of pick up items, we will create intersect methods; comparing when the two objects collide with each other. Each ingredient will be stationary.
- Display the Ingredient Stack: We will display the ingredients that the chef has in his/her possession.
- Display the Amount of Money Earned
- Arrange the 'Walls' (Tables)
- Add Background Music
- Make Resources
- *OPTIONAL:*
    - Add Sound Effects
    - Add Title Screen

The foundation of this game derives from the first game we implemented, the tank game. We used the same logic for movement and for drawing each resource on the screen.

3. Tank Game

a. Class Diagrams



b. Movement

The possible directions of the tank were designated in the class Direction enumeration. (**INITIAL** indicates where the tanks should be initially placed as the game is loaded.) The way those directions

were implemented in the game is stated in the Tank class. Player 1's movements are controlled by the keys **A**(left), **S**(down), **D**(right), and **W**(up). Player 2's movements are controlled by the keyboard arrow keys. Each possible direction the player could move to has a corresponding boolean variable to help indicate the actual movement of the tank. This can be seen in our Tank class method decideDirection(), where an if-statement utilizes the boolean variables and sets the direction variable to equal either **L**(left), **U**(up), **R**(right), **D**(down), or **STOP** if none of the booleans are true. The direction **INITIAL** is not included in decideDirection() due to the tanks only being placed in this position for the very beginning of the game. The direction variable is then used in a switch-statement located in Tank method move(). In case **L**, the X-coordinate placement of the tank is reduced. Case **U** reduces the Y-coordinate placement. Case **R** and **D** increase the X- and Y-coordinate placement of the tank, respectively. Case **STOP** and **INITIAL** do not change the current positioning of the tanks.

c. Walls

There are two types of walls, unbreakable and breakable. The unbreakable wall is initialized in our Wall java class and our breakable wall is initialized in our BreakableWall java class. The placement of each wall within the game can be seen in our TankClient

class, where for-loops are used to designated X- and Y- coordinates for each wall-type. The images used for each wall are contained in our Images folder and each image is contained in a rectangle initialized by method getRect(), which is present in both java classes. This method will be used to prevent the tanks from going through each wall.

4. Drawing the Resources on the Screen

As previously stated, each image used in both games is held in the Images folder. To have the images load onto the screen, we used java superclass Toolkit to grab the resources from the file and place into an Image arraylist. We then have method draw(Graphics g) to display the specific image desired onto the screen. For the movement of the player, each direction has a corresponding image associated. For the correct image to display regarding player movement, we used a switch-statement. Depending on the the direction a player was facing, each case would prompt the java superclass Graphics method drawImage to load the desired image from the Image arraylist.

a. Resource Collisions

For each image, an encompassing rectangle is set in the method getRect(). Once the initial placements of each image is defined, we incorporated boolean methods to control how each image interacted with

one another if their rectangles intersect. For the Tank Game, this would include wall collisions, tank collisions, and bullet hits. For the Sushi Hero game, this would include ingredient collisions and table collisions. A single if-statement is used for each boolean method, checking to see if the object's rectangle intersects with another. Generally, if the player's rectangle does intersect with another, the method changeToOldDir() is called to prevent the images from overlapping.

b. Calling Each Resource from the Client

To set each stationary image of each player on the screen to their initial placements, an arraylist is first instantiated. A new object from either the Tank or Sushi class must be created. For stationary images, for-loops were used to add multiple instances to create lines of walls or bundles of food in specific areas of the screen.

To add resources, we used Adobe Illustrator to create the graphic. Illustrator has convenient options to bound the whitespace to the graphic itself, and also make any remaining whitespace transparent. After exporting it as a .gif, we then copied it into our Images folder and created a new Java class for that graphic. We copy and pasted the code from our Wall class and pasted it into the new

java file, renaming Wall to the name of the resource. This code contains fields whose values need to be replaced for each resource. After finding the dimension properties of the graphic, we recorded its width and length (and added a few to ensure smoother collision) in the class. Then we needed to add collision methods so that they are able to take in each type of Object as parameters in our Sushi class. We then needed to adjust a few things in our client. In SushiClient, we created new Lists to contain those image resources so that we could later call them. For each resource, we also have four boolean variables for each of them (later explained). We are able to display each resource from our Lists that we created. It took a good amount of guessing and checking to place each image in the exact coordinates that we wanted.

5. Boolean Check System

In order for each method to run at the proper time and aid in smoother game play, boolean check systems are used to indicate that certain conditions have been met. These check systems prevent the player from, for example, going through walls and displaying incorrect scores.

a. Collision Methods

For now, all of our collision methods are stored in our Sushi class. For each resource, we had to determine which resources can be passable or not. All of our 'wall' resources (including Wall, FatTable, WoodenTable, ChoppingBoard and Sushi) are impassable, while all the ricecooker, chopping board, ingredients and carpets are. We figured out that in order to dictate what happens when the objects collide, the rectangles must be able to intersect.

b. Displaying the Last Received Ingredient

For the player to keep track of what ingredient the chef possesses, SushiClient class method displayCurrentIngredient() was created. We ended up creating a total of 21 boolean variables. The first two sets of variable signify whether each player has the ingredient or not.As we were trying to display the current ingredient, we were running into numerous problems. At one point, we had it stop displaying the correct item after the second ingredient. Later on, we got it so that it would consistently display

```
boolean player1Rice = false;
boolean player1Seaweed = false;
boolean player1Avocado = false;
boolean player1Egg = false;
boolean player1Tuna = false;

boolean player2Rice = false;
boolean player2Seaweed = false;
boolean player2Avocado = false;
boolean player2Egg = false;
boolean player2Tuna = false;

boolean riceAttained = false;
boolean seaweedAttained = false;
boolean avocadoAttained = false;
boolean eggAttained = false;
boolean tunaAttained = false;

boolean seaweedChopped = false;
boolean avocadoChopped = false;
boolean eggChopped = false;
boolean tunaChopped = false;

boolean riceCooked = false;
boolean everythingOK = false;
```

only if the prep was standing next to the wooden table. Eventually, we saw that the prep can only hold one item at a time. This means that when the prep collides with the wooden table while holding an item, *all other boolean variables must be set to false*. This would mean that the chef's boolean values will alter. We still need to signify when all appropriate items are attained in order for the sushi to be delivered correctly.That is why we created 'copy variables' (Attained) which have the same interpretation of the chef's variables except once they are set to true, it does not change back until the next sushi order. Variables, riceCooked and everythingOK, both must also be set to true in order for the sushi roll to be delivered.

For the player to keep track of what ingredient the chef possesses, SushiClient class method displayCurrentIngredient() was created. This method checks a set of booleans representing what ingredient the prep cook has grabbed (player2Rice, player2Seaweed, etc.) and is now placing on the dividing table to hand off to the sushi chef. The prep cook's ingredient-possessing booleans are set to false and the sushi chef's (eggAttained, tunaAttained, etc.) are set to true. This signifies the passing of ingredients once the prep's rectangle intersects with the carpet underneath the dividing table in Sushi class boolean method collideWithCarpet.

c. Displaying the Current Order

For the current order to display properly and change upon completion of each previous order, an arraylist of strings was created to hold the three different sushi rolls. An int variable representing the index within the arraylist is randomly generated and changed every time the chef collides with the leftmost table while holding each ingredient necessary for the designated roll.

d. Displaying the Score

Each roll is given a specific price, with price increasing as the rolls require more ingredients. In order for the score to properly reflect the price of a roll that is served, an if-statement is used. If the chef is holding all the ingredients necessary to create the current order while standing next to the serving area, the score was incremented by the price of the roll created. All of the booleans are then reset to false, signifying that the order has been served. The index is then randomly changed and the method changeOrder() is called to display the next order up.

6. Future Goals and Considerations

This project gave us both a lot of insight. We now see how all games essentially revolve around collisions, and programmers dictate what happen. Given that Tank Game and SushiHero are both of our first

attempts at creating games, each proved to be a challenge in their own ways. With this in mind, we believe there are certain aspects of the games we would like to expand and build upon in the future in order for game play to be more enjoyable.

a. Incorporate Pickup Items

In the Tank Game, including bombs and lives as pickup items would have contributed to a richer gaming experience. For Sushi Hero, visually picking up ingredients and chopping them would be ideal. One way we could have implemented this would have been to create instances of each item appear on the screen. Once a player's rectangle collides with the item, this would prompt the item to appear in the player's possession. We can then remove the image to simulate that it has been 'picked up.' When we learn to implement pick up items, we will then be able to brainstorm about power ups and downs. Ideas we had in mind included energy drinks that temporarily increase speed, or a bottle of sake that temporarily reverses direction.

b. Cleaning Up Methods

As of now, our class hierarchy is flat. The only form of inheritance is our SushiClient extending Frame. We figure later that we could clean up our collision methods by having each object class extend a single parent Collision class. That way, we can

polymorphically define just one single collision method in each subclass.

c. Add More (Complicated) Sushi Roll Orders and Levels

As a player continues to play the games, typically one would expect the games to increase in difficulty. One way we would like to do this with SushiHero would be to include more complicated sushi rolls that require more ingredients on perhaps a bigger canvas. For the tank game, it would include multiple levels with different wall placements and faster tank speeds.

d. Implement Time Systems

To add a sense of urgency, the idea was brought up to include time constraints in the games. One way this could be implemented in the Tank Game would be that once the bombs are picked up, a timer would begin counting down. If the player did not drop the bomb or get out of range in time, the penalty would be that their health would deplete. For SushiHero, each order in the queue must be served within a minute of being posted up. Another constraint would be giving the rice a required time to cook, where the player would have to take the rice out of the rice cooker before it becomes unusable for the order.

e. Animate .gif Resources

For a more enjoyable gaming experience, including animated .gifs was suggested. For the Tank Game, this would be adding an explosion every time bullets made contact with another object or having a player's health bar deplete with every hit. For SushiHero, smoother game play would have each ingredient on the chopping block visually seen chopped in a .gif. Having the players' avatar also show more animation as they move across the screen would be ideal as well.