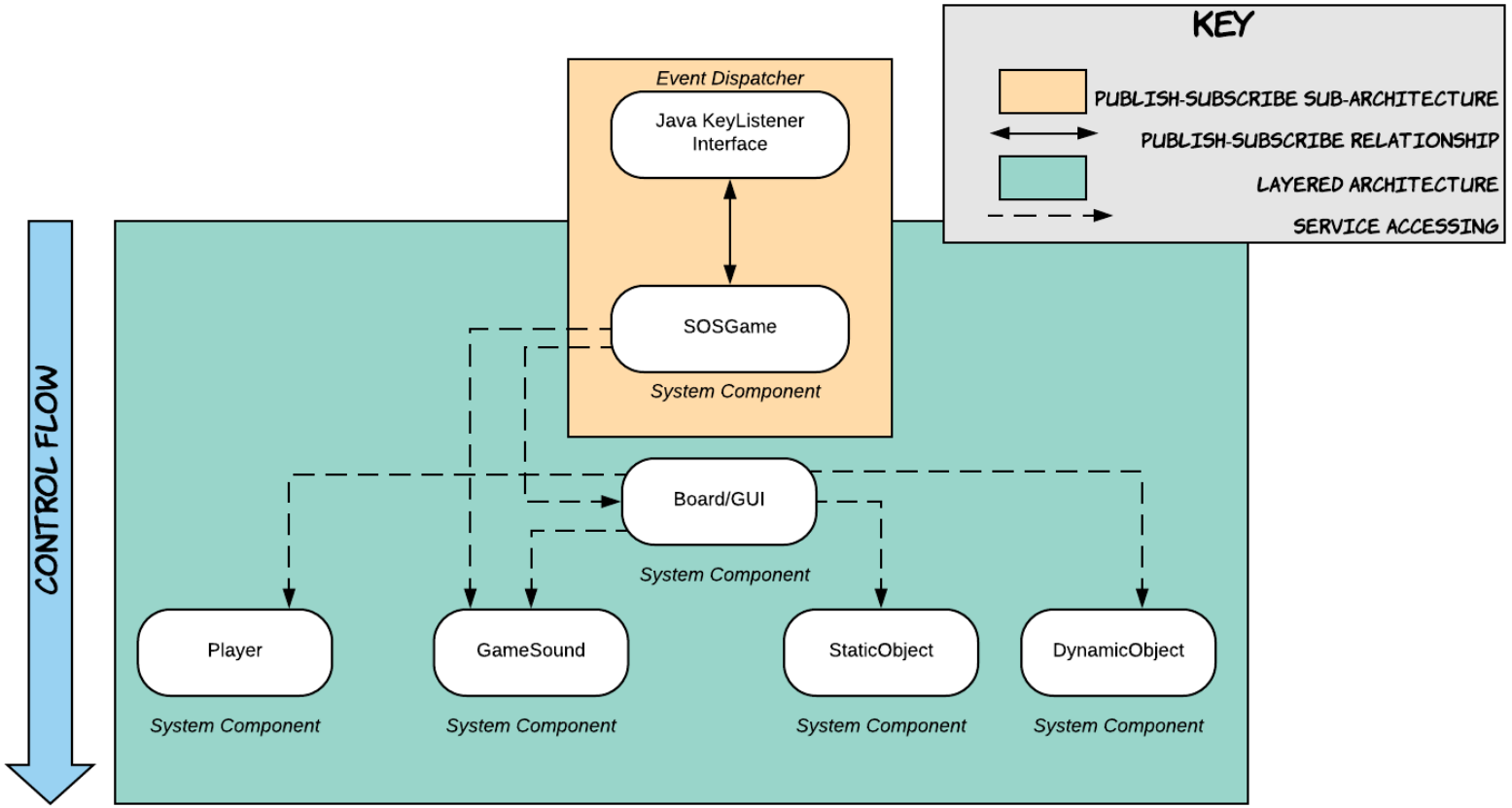
System Design

Here is our Architecture Design Diagram:



Explanatory Text

For our particular project, we will implement a custom event-based and layered system that will manage both user input and the graphical user interface (GUI), which is responsible for displaying our game's board, character, static objects, and dynamic objects. As the previous architecture diagram illustrates, our game will be implemented using a combination of layered and publish-subscribe architecture designs. It is a layered architecture since certain layers in our system will access services while others will provide them. A perfect example of such scenario is when the SOSGame class accesses services from the Board class whenever the “repaint” method is called. When this method is called, the Board class will repaint the board and anything “on top of it” using the system’s GUI. We note that the converse of this is NOT true. That is, the SOSGame class does not provide services to the Board class and the Board class does not try to access services from the SOSGame class. Another example that illustrates our system’s layered architecture design is when the Board class access services offered by the Player, GameSound, StaticObject, and DynamicObject classes (e.g. the Board class will need to access a Player's X and Y coordinates in order to display him or her in the correct location within the board; the Player class will offer the “service” of correct coordinate points). It is important to note that the Board class does not directly change the classes it is trying to access (i.e. Player, GameSound, StaticObject, and DynamicObject) since these classes “change themselves” through any specific methods called by the Board class. In other words, the interaction of the four classes at the lowest layer (i.e. Player, GameSound, StaticObject, and DynamicObject) will be facilitated by the Board class. Since our system is designed in this way, we note that the control flow goes from up to down, meaning that levels higher up “control” lower levels. This implies that our system will use a "top down" approach when dealing with our system's control flow since more complex classes should execute first. That is, class complexity decreases the lower it is in the layered part of our architecture design diagram. Our game will also have a publish-subscribe sub-architecture system design since the SOSGame class subscribes to Java’s keyListener interface and will “observe” if any keys are pressed by the game player. If a key is pressed by the user, the SOSGame class will “activate” the system’s control flow by calling specific methods in the Board or GameSound classes.

Description of the Class-Level Design

Our system design will be composed of six main classes: GameSound, Player, StaticObject, DynamicObject, Board, and SOSGame. We point out that all six are major classes and are critical to our system design, but their complexity varies depending on their location in the layered architecture design. For example, the SOSGame class is more complex (and thus is a major class since it is the most complex class) than the Board class, and the Board class is more complex than the Player class. It is important to note that the Player, Sound, StaticObject, and DynamicObject classes are equal in complexity and that the complexity of the Board class lies in the middle (i.e. it manages the interactions between the classes in the lower layer). SOSGame also implements

the KeyListener interface, so a separate class for this “event dispatcher” is not needed. The following paragraphs describe more in detail the responsibilities of each of the classes' objects.

SOSGame Class

The SOSGame class is the only system component that belongs to both the layered architecture as well as the publish-subscribe sub-architecture. The SOSGame class participates in the publish-subscribe design by implementing the KeyListener interface to subscribe to KeyEvent events. It simultaneously behaves as the highest layer in the layered architecture by requesting appropriate services from the Board class based off valid keystrokes. In this manner, an SOSGame object comprises the highest layer of abstraction of system execution. An SOSGame object is also responsible for validating the keyboard keys pressed by the user at any specific time of the game. A perfect example that illustrates such validation is when the SOSGame object ignores other keyboard keys pressed except for the four arrow keys while the user tries to move the game character around the game board. Because users can only execute one SOSGame object at a time and an object of that class contains the main function for the entire system, we can conclude that the SOSGame class is classified as a singleton class. Note that the single instance of layer bridging in the Architecture Design Diagram involves the SOSGame class, allowing it to access the services of the GameSound class in the lowest layer. This architectural design decision was made to improve performance by efficiently implementing the universal game background music at the highest level of abstraction.

Board Class

We should emphasize that this class contains the paint function that updates the GUI. Consequently, the main responsibility of a Board object is to display to the user the board (which is sometimes called the game's "map") and anything extraneous placed on top of the board (e.g. game character, rocks, etc.). We note that the Board class triggers changes within the Player, GameSound, StaticObject, and DynamicObject classes but does not directly change them in the Board class, reducing coupling between the various modules in the system. Additionally, the Board class requests information (accesses provided services) from the Player, GameSound, StaticObject, and DynamicObject classes as a client. One example of such a request will be a Player object's location and image since a Board object needs this information to paint the correct game character image and location on top of the board. Because the game can only have one board that changes accordingly to user input, the Board class will be implemented as a singleton class. Lastly, in addition to having all the opening scenes/map images, an instance of a Board/GUI class will also contain and manage most of the game's logic on what to display at any given time (e.g. the Board object will be responsible for displaying the static and dynamic game objects available for interaction with the Player object at any given instant).

Player

The main responsibility of a Player object will be to store all relevant information pertaining to the game's character, including its x-coordinate, y-coordinate, name, and image. In other words, a Player object will represent the actual game character that moves around the board. The Player class will offer services to the Board class. Such services include providing the location of the game player and its corresponding image to the Board class so that it can paint it on Java's GUI. Another example of a service that the Player class can offer is the name (in the form of a character string) that was originally inputted by the user. We note that a Player object will represent either the male game character or the female game character. Because the game can only have one game character at a time (there is no multiplayer option for our game) that changes accordingly to user input, the Player class can be implemented as a singleton class.

GameSound

The main responsibility of a GameSound object will be to play distinct sounds every time a meaningful game event occurs. Examples of such events include playing thunderstorm sounds during the title screen GIF or a "slurping" sound every time the user eats a poisonous fruit. The GameSound class will offer services to both the Board and SOSGame classes. Such services include playing a specified audio clip through the computer's speakers. For example, a GameSound object will play the splash audio clip whenever the user touches the water on level 2. We note that a single GameSound object will be composed of several audio clips. Because multiple sound objects with different audio clips will be played throughout the game, we can conclude that the GameSound class is not a singleton class.

StaticObject

The main responsibility of a StaticObject object will be to represent static game objects that the game character can interact with (e.g. rocks, fruits, etc.). A StaticObject object is like a Player (or DynamicObject for that manner) object since both will be represented by an image and a location on the board. However, a StaticObject object will not move during gameplay and will be allowed to be picked up by a Player object (if both objects have the same location within the board then the Player object will have successfully picked up the StaticObject object). One extra attribute that the StaticObject class will have will be a type that will represent what type of game object to display to the user. The StaticObject class will offer services to the Board class. Such services include providing the location of the static object and its corresponding image to the Board class so that it can paint it on Java's GUI. Because multiple static game objects with different locations will be painted on the board, we can conclude that the StaticObject class is not a singleton class.

DynamicObject

The main responsibility of a DynamicObject object will be to represent dynamic game objects that the game character can interact with (e.g. bees, monkeys, etc.). A DynamicObject object is similar to a Player (or StaticObject for that manner) object since both will be represented by an image and a location on the board. However, unlike a StaticObject object, a DynamicObject object will move during gameplay and will be able to "harm" a Player object (if both objects have the same location within the board then the Player is "killed" and the current game level is reset). One extra attribute that the DynamicObject class will have will be a type that will represent what type of dynamic game object to display to the user. The DynamicObject class will offer services to the Board class. Such services include providing the location of the dynamic object and its corresponding image to the Board class so that it can paint it on Java's GUI. Furthermore, the DynamicObject class will also offer the service of "randomizing" the locations of the dynamic game objects. Because multiple dynamic game objects with different locations will be painted on the board, we can conclude that the DynamicObject class is not a singleton class.

Control Issues

With reference to the Architecture Design Diagram, control flows from top to bottom in our system. The above discussions of both the architecture design and the class-level design hinge on our approach of using a layered architecture with a publish-subscribe sub-architecture. This design decision, although seemingly complex, actually provides the framework for simple implementation of our system. First we will discuss control hierarchy for our publish-subscribe design, next we will discuss the control hierarchy for our layering, and finally we will discuss how these two decisions fit together in the overall top-to-bottom control structure.

The publish-subscribe sub-architecture, which encompasses the top of the system (orange box in the Architecture Design Diagram), was a direct consequence of our strictly event-based system in which the game only changes state in response to user input. While this was a straightforward and logical choice, we still had to make the best selection in converting this high-level design choice into a low-level design pattern. We are using the Observer behavioral design pattern, which is characterized by allowing objects to be notified when other objects change state without hardcoding those connections. This choice supports the most significant control pattern of our system: game state changes are dictated strictly by user keyboard input. To achieve this publish-subscribe sub-architecture, the singleton SOSGame class implements the Java KeyListener interface. Simultaneously, the singleton SOSGame class behaves as the highest abstraction of our layered architecture (described next).

The layered architecture encompasses the majority of the system (green box in the Architecture Design Diagram). At the highest level of abstraction, the singleton SOSGame class accesses services offered by the Board and GameSound classes in order to drive and update the GUI. The next highest level of abstraction is our middle layer, occupied solely by the Board class. The Board class

acts as a client of the Player, StaticObject, DynamicObject, and GameSound classes by triggering applicable updates of objects within any of those classes, based off the event information that the Board class receives from the SOSGame class. Subsequently, the Board class accesses the services of some or all of the already-updated objects in those subordinate classes. This is because full variety of game objects impacted by user input must be composed onto the unified GUI for the game player to observe, giving the Board class higher control complexity than that of the Player, StaticObject, DynamicObject, or GameSound classes. The Board class naturally aligns with the Singleton creational design pattern, since our game is limited to a single graphical interface. This design pattern decision should be enforced in our system by disallowing external classes to create new Board objects.

Finally, we see from the Architecture Design Diagram that the publish-subscribe structure fits into the top level of abstraction of our overarching layered control structure. Namely, our system is designed to be event-based, with control flowing from the SOSGame class (which implements KeyListener) down to the Board class and then to each of the four remaining classes, with each updating its own objects as needed in reaction to user keyboard input. Hence, each subscribed event ultimately calls follows the layered control structure down to individual elements of the map (as represented by the Player, StaticObject, DynamicObject and GameSound classes) which update and provide informational services to the Board client to produce a responsive GUI.