SOFT2201 Assignment 3 2022
510460295

The given codebase displays the design principles of SOLID and GRASP. Single responsibility is showcased extensively through the entire application, but especially in the classes involved with reading the config seen in Figure 1.

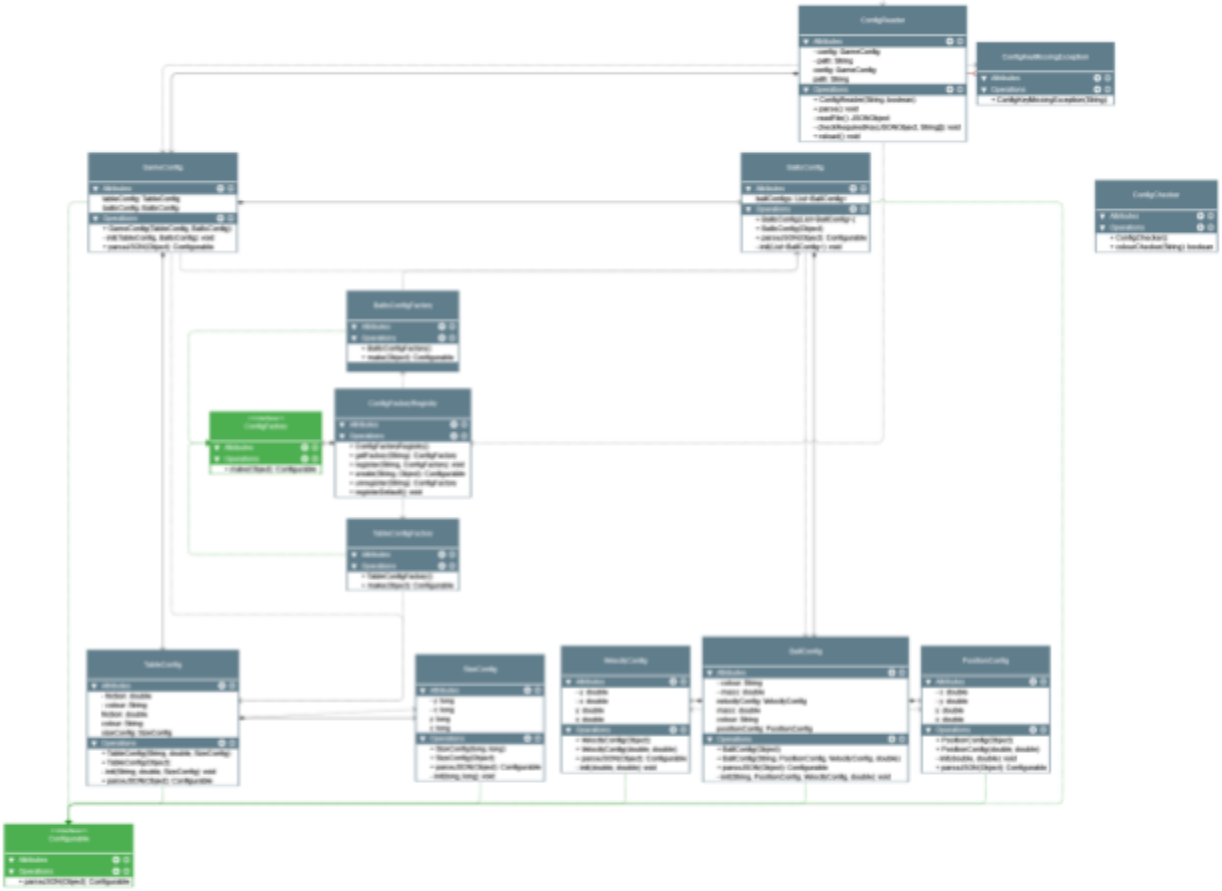


Figure 1: Config classes shown in UML

Each part of the configuration file is read by its own specific class which implements the configurable interface. This ensures single responsibility as there is no single class responsible for reading all of the information, becoming responsible for distributing this information to all classes which need it. This also means each of these config classes is solely responsible for handling errors to do with that specific part of the config.

Taking for example the creation of a ball, the classes involved with this task only have the information to read into exactly 1 layer further into the JSON than the class which instantiated it. Since each of these classes also only have the ability to store that information they can access plus any other configFileReader classes it creates, the information expert principle is used as the responsibility for holding data is only given to the class with the information to get

that data. However, this section does not utilise the dependency inversion principle as each class is referenced directly instead of through the interface which each of these classes implement.

Another benefit of this design is that these configuration readers can be reusable for future objects that need to be configured on screen. For example the positionConfig objects can be used to define the x and y of any object from the config file, not just the x and y of just the balls. Itemising each configured object like this also makes this design easily extendable, as any other items being configured can just have their own config object which will utilise each reader in the same way.

The design patterns used in this implementation were Factory, Strategy and Builder. The factory pattern can be seen in Figure 1. It uses a FactoryRegistry to store the two Factories of Balls and Table. The use of a FactoryRegistry keeps coupling low between the ConfigReader class and the factory classes and obeys dependency inversion as it only needs to depend on the registry instead of each factory type. This is extremely extendable as in order to create new factories, you would only need to add the factory to the registry.

The strategy and builder patterns are used closely together as seen in Figure 2.

Figure 2: Builder and Strategy patterns

The strategy pattern interacts with the BallBuilder classes through the BallPocketStrategy interface utilising the dependency inversion principle. Liskov Substitutability is also used here as each behaviour could be swapped with each other and there would be no harm done, this is especially useful if you wanted to change a ball's ability at run time or dynamically. The builder pattern also displays the dependency inversion principle as the director accesses the BallBuilder classes through a Map which uses the BallBuilder interface as the type. As these two patterns are working so closely together, there is a high degree of coupling, however still a high amount of

cohesion, as the same input variables are creating different objects with different behaviours very easily. This cohesion can again be seen as the task of building a ball object is split into many classes, which arguably could be done in a single BallBuilder class. This is because each ball should only differ by its colour and its Strategy behaviour which would be passed in or created depending on what type of ball is being made.

The documentation for this project is extremely well done. The readme explains nuance concepts in the physics engine which would be difficult to find out if they were not documented, which is extremely helpful. Further the commands listed on the readme are clear about what they create. The javadoc explains each documented function extremely well allowing for the codebase to be quickly understood, especially when trying to alter or extend a function or certain types of functionality.

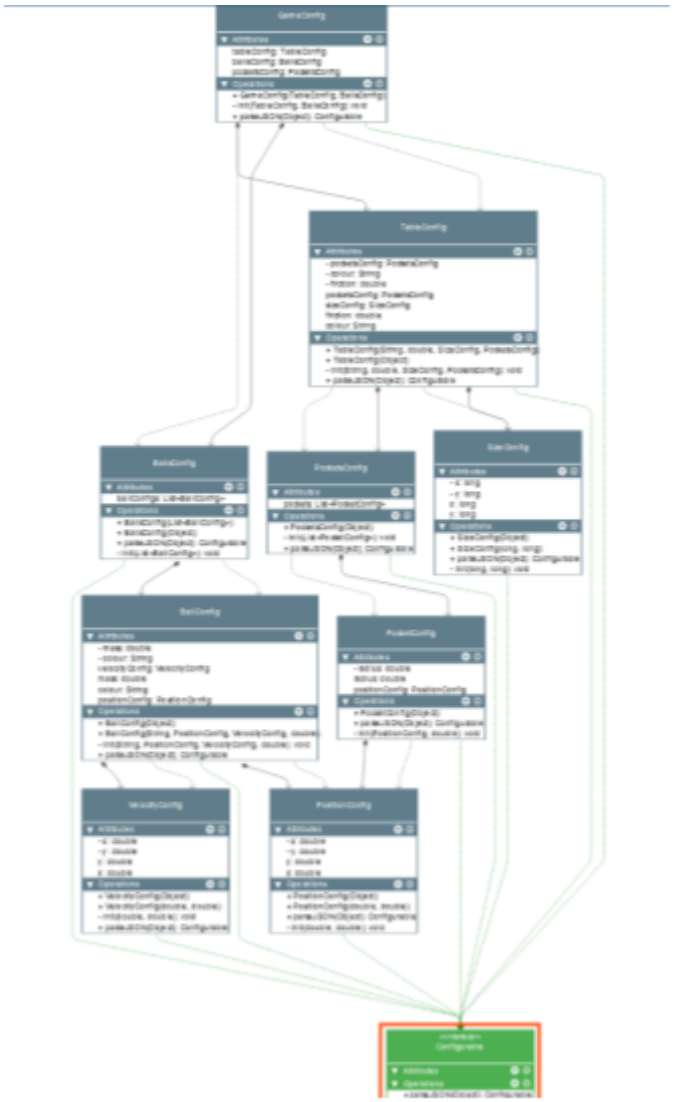


Figure 3: Modified Config Neighbourhood

The given codebase made implementing the extensions relatively painless with no major refactoring required. This is because each class was extremely focused and the responsibilities and information held by each class was appropriate. Because of the extendable nature of the Config reading, I was able to simply add the PocketsConfig and PocketConfig classes which implement the configurable interface. These config classes were then used in the TableConfig class to hold the variable pocket information instead of the constant pocket information being used before in this class. This can clearly be seen in figure 3. This upheld the single responsibility nature of this design as each element held its own information coming from the config file. This was also able to reuse the previously established PositionConfig class to hold the positions of the pockets further establishing the extendability of the given code.

Furthermore, the Builder pattern used in the established codebase was also easy to extend as for each new coloured ball, I simply made another Builder class pertaining to that colour. This allowed me to register the specific ball colour in the director class, then create a new ball based off the colour in the config file. These extra classes can be seen in figure 4.

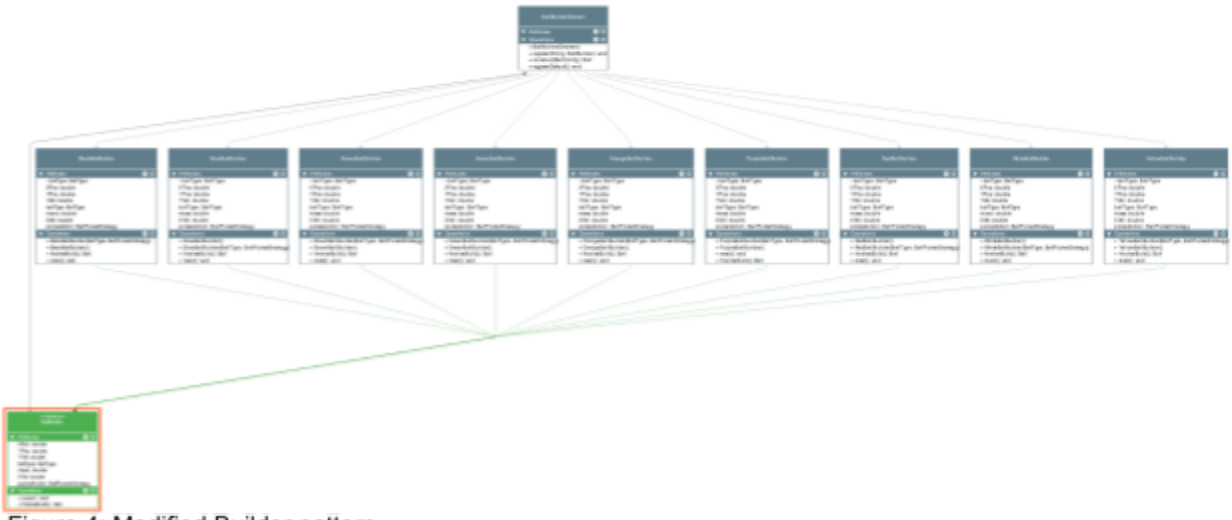


Figure 4: Modified Builder pattern

However, I do believe this function could have been achieved with a single builder class, where the director is used less as a registry and instead stored the corresponding score value and strategy pattern for each colour, so then it could just set the strategy and value in the single builder class. This would reduce repetitive code in each of the Builder classes and would allow the UML to be neater, thus potentially more understandable though I do understand the idea of segmenting that responsibility of knowing the each coloured balls value away from the director.

The strategy pattern was also simple to extend as I created a new class called PocketThrice which also implemented the BallPocketStrategy interface to represent the ball needing to be pocketed 3 times. This class essentially just changed the variable FALL_COUNTER_THRESHOLD to 3 from 2, however by creating a new strategy class, it allows for an extremely extendable functionality for each coloured ball in future beyond just falling multiple times. This can be seen in figure 5.

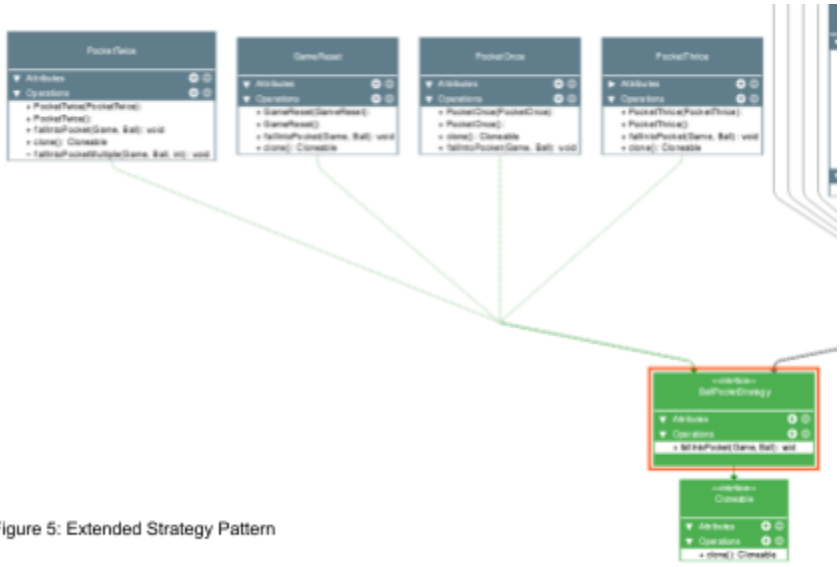


Figure 5: Extended Strategy Pattern

Extensions

Configurable Pockets

As discussed above, I implemented the configurable pockets using 2 more Config Classes. PocketsConfig contained each PocketConfig object, just like how in the original codebase BallsConfig contained all of the BallConfigs. This PocketsConfig object was created and stored by the TableConfig class. Thus when the TableConfig was passed to the PoolTable class by the Game class in order to create the tables, the Pockets were then created by the Table. This is a good use of the Creator design principle as can be seen in figure 3, as each class is only created by the class which contains it.

More Coloured Balls

Again, as seen above I extended this functionality using the builder pattern originally provided. By adding new classes for each new coloured ball, I needed to change the director class to add these new coloured balls to the registerDefault method.

CueStick

Implementing the CueStick meant I had to remove the original 'Visual Cue' present in the game. To do this I removed the line being created in the Ball class and replaced it with the CueStick object which by default was set to not visible. I was able to repurpose the registerCueBallMouseAction() method to set the CueStick as visible and move its location to where the CueBall is when it is clicked. By doing this I was able to reuse the setType() method from the original codebase which only ran the register method on the CueBall, thus making it the only clickable object. To ensure single responsibility the CueStick needed to have its own class as the logic of having the stick rotate to face the ball and the mouse actions of the CueStick is not the responsibility of the Ball. The downside of this is these classes have high coupling as the Ball must pass itself to the CueStick as the CueStick needs to use the setVel methods of the Ball in order to make it move. One drawback on this implementation is that each Ball creates its own CueStick as it contains it, although this obeys the Creator principle, it means that there are as many CueStick objects as there are balls. This clutters memory and it would be better for there only to be one CueStick object only available to the CueBall.

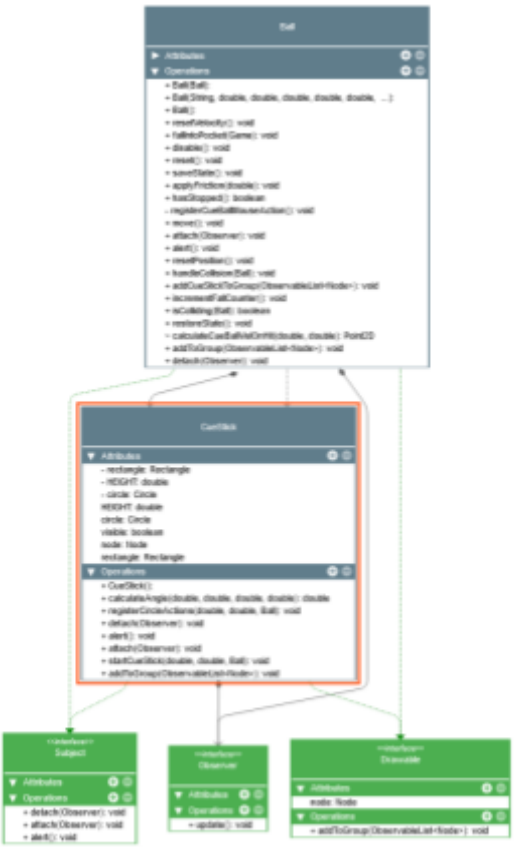


Figure 6: CueStick Creation

Difficulty Level

This extension was entirely implemented inside the App class, although this class was originally intended to only start the game, I found it to be the most appropriate place to do two things. First, I needed to be able to create setOnKeyPressed events on the scene, as this scene was created in the App class it is readily accessible here. Secondly the changeLevel method also fit well here as I needed to be able to access the root node's children in order to clear it and add the new drawable objects created by the new game and set the timeline to the new game's tick() method. Although this denies single responsibility, this decreases coupling within the program as a separate class would need to be highly coupled to the App class, potentially increasing complexity of the code for no real benefit.

Figure 7: Modified App Class

Timer and Score

The Timer and Score extensions meant I needed to change the layout manager of the scene from the default to a border pane. This is so I can have the pooltable in the centre of the scene and the top of the scene be a header which contains both the score and the timer.

Both of these extensions were implemented using the Observer pattern. This was used to keep coupling low and ensure single responsibility by utilising indirection where the timer and score interacted with the ball class through a separate interface. The ScoreKeeper and Timer object only needed to be alerted when a ball was disabled, thus in order to prevent wasted instructions per frame, I passed the Observer classes their own creator which they would notify when the observer was notified. This is because the Timer needed to know to stop if the CueBall had been pocketed or if all of the balls besides the CueBall had been pocketed. The ScoreKeeper needed to update its value depending on what ball had been pocketed for the final time.

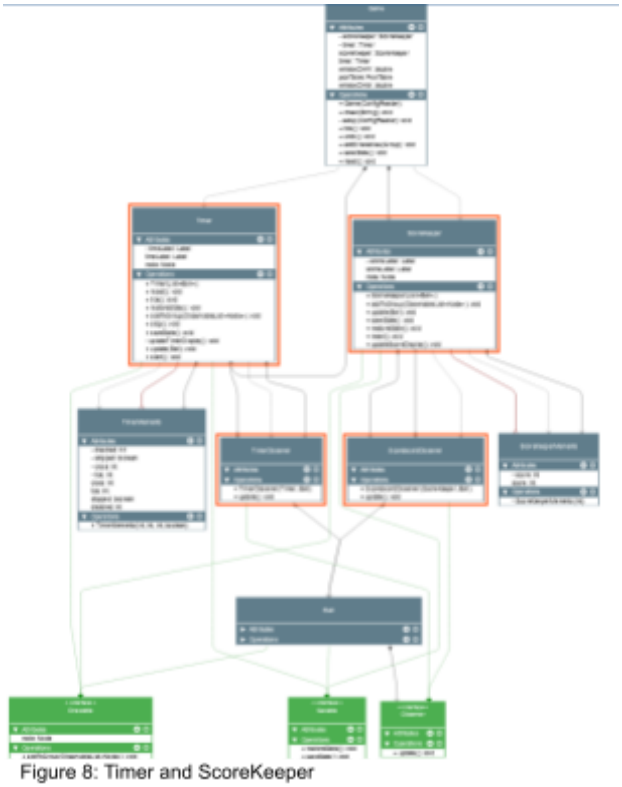


Figure 8: Timer and ScoreKeeper

This meant the Ball was the subject being observed for these two objects, thus needing to implement the subject interface. Whenever the ball was disabled, it would alert its observers, where those would then alert the timer and score objects. The Timer and Score items are created in the Game class, which contains them obeying the Creator principle. The Timer and Score classes are in charge of creating the observers and attaching them to each Ball. As can be seen by figure 8, using the observer class decreases coupling as the Timer and ScoreKeeper class do not directly depend on the Ball class.

Undo

Undo is called by pressing the key ‘U’ on the scene, this means the first edit I needed to make was in the App Class inside the setKeyBindings() method to set ‘U’ to game.restoreState(). This meant that the App was still only dependent on game and config, however it could trigger a restore using a keybind on the scene. In order to store the state to restore to I used the memento, prototype and observer design patterns.

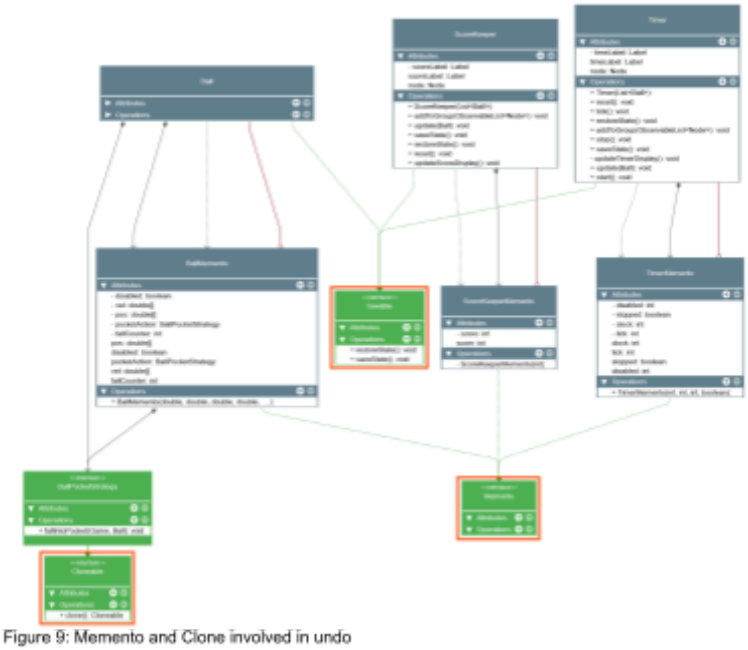


Figure 9: Memento and Clone involved in undo

The memento design pattern was used in the 3 classes that needed to be undone, those being the balls, timer and score. The way I did this was by creating two interfaces, one for the actual object being saved named Savable, and one for the nested class inside this savable object named Memento.

Currently, savable enforces two methods, saveState() and restoreState(). saveState() uses the nested Memento class to save all of the current restorable attributes of the object, which is then set as the currentSave attribute of the Savable object. restoreState() checks the currentSave attribute of the Savable object isn't null, if not then the saved attributes are applied to the current state of the object. The Memento interface does not currently enforce any methods and so is not necessary, however is useful in identifying the nested classes involved with being saved, and it also in place in case any common methods are required in future, helping the extendability of the project. Using nested classes to create the Memento it delegates creating the state snapshot to the owner of that state, this encapsulates the creation of this saved state as the owner of that state has full access to all of the private variables needed for the save ensuring the data inside the original object does not need to be as easily accessible to other object.

Any complex objects inside the Memento classes need to have a deep copy stored of them. This is where the prototype pattern is used. Currently the complex attribute being saved and restored is only inside Ball which is their BallPocketStrategy. Although this is technically unnecessary now as they do not have any changing variables inside them-selves, future extensions to pocketable behaviour may involve this, meaning it is important to have a deep copy of these Strategies. This is done by having the BallPocketStrategy interface extend the Cloneable interface, forcing each strategy to have a clone() method. This clone method is then called inside of the saveState() method when creating the BallMemento, so that a copy of the PocketStrategy

is stored inside of the BallMemento, instead of a reference to that ball's current PocketStrategy. The above is shown in figure 9.

In order to know when to automatically save the undo, I needed to decide when a good time would be to actually save, that the player would do without just pressing a save button. I decided that the moment just before the cueStick hits the white ball would be the best time for this as this would represent the moment just before the player hit the ball and would only happen once per turn keeping the number of instructions per frame down. The only object with this information is the CueStick, and the only object with the information to call the saveState method in all of the balls, timer and scorekeeper is the game. In order to avoid dependency inversion, I used another Observer where the CueStick is the subject. The CueStick alerts the observer when the player takes their shot, which then alerts the Game object to saveState of the savable objects, thus saving the game at the correct time. This can be seen in figure 10.

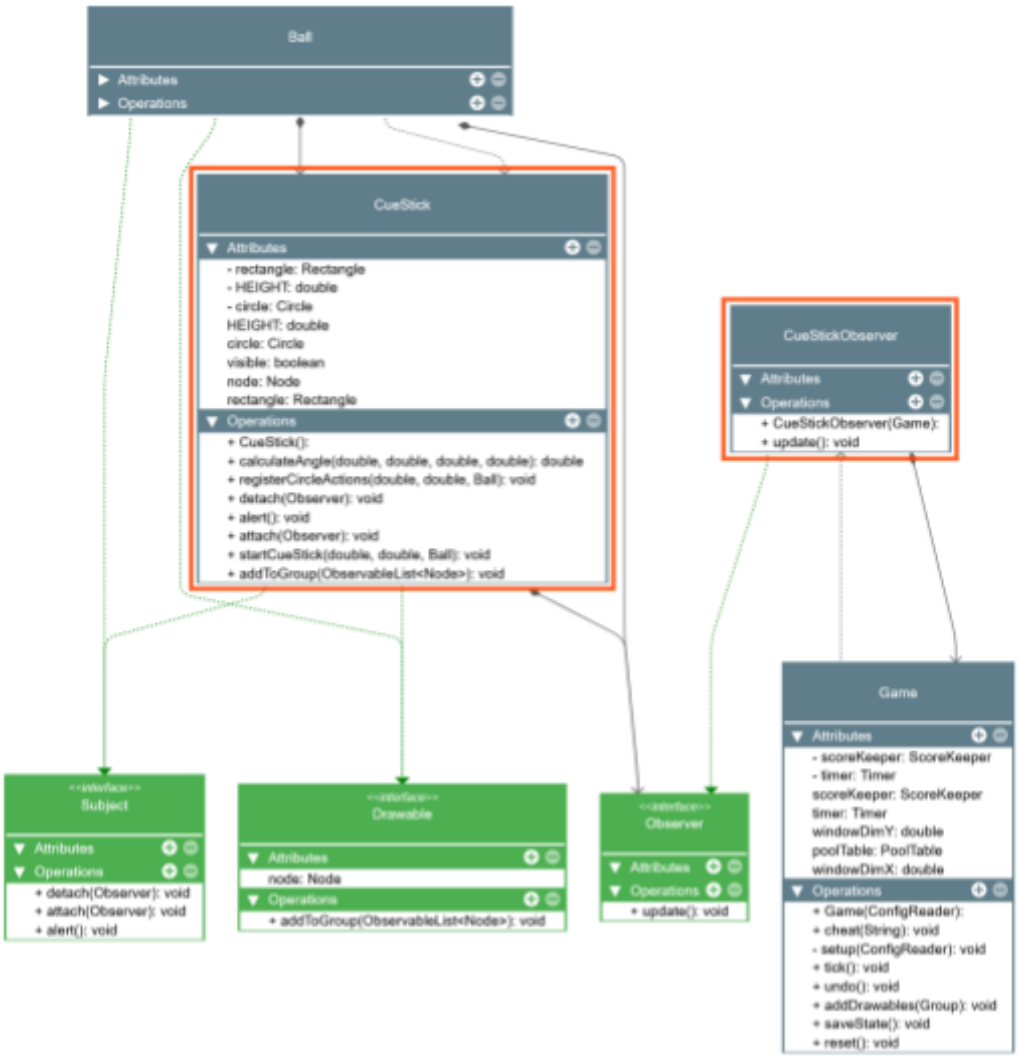


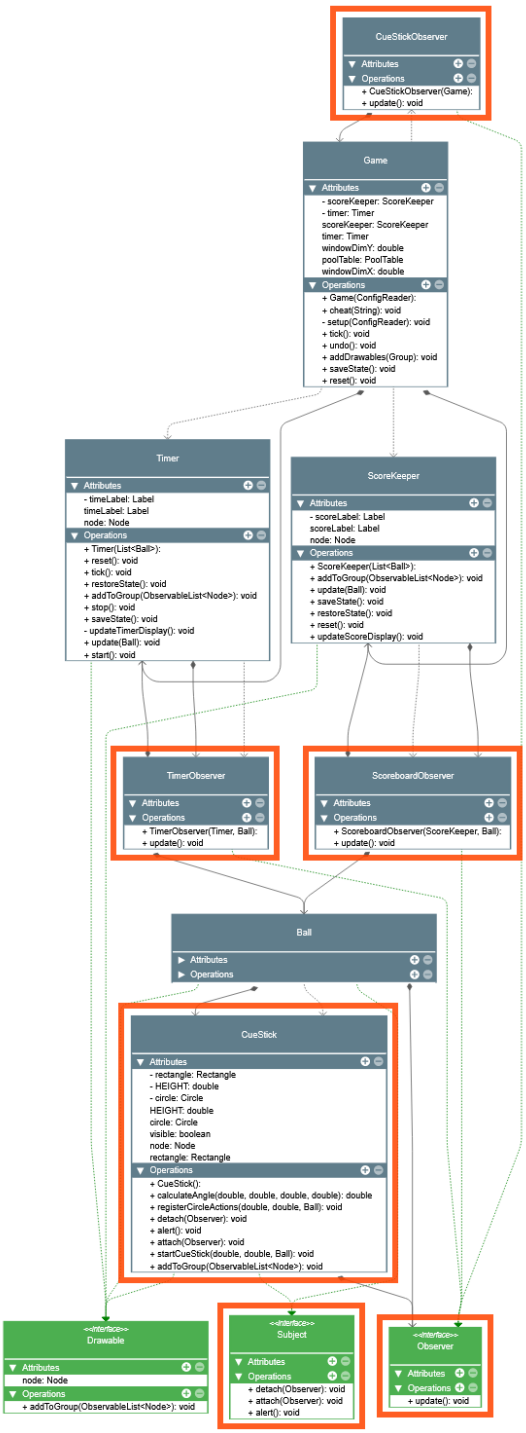
Figure 10: CueStick Observer

Cheat

The cheat functionality was implemented first by providing the keybinds on the scene in the App class. For example, if the player pressed 1, the scene would call game.cheat('<hexvalueofyellow>'). This then went to a for loop in the game class that would iterate through the balls and disable the Balls whose color.toString() matched this hex value passed to the function. This implementation is not very complex, however by giving further responsibilities to the app class it may be jeopardising the single responsibility principle.

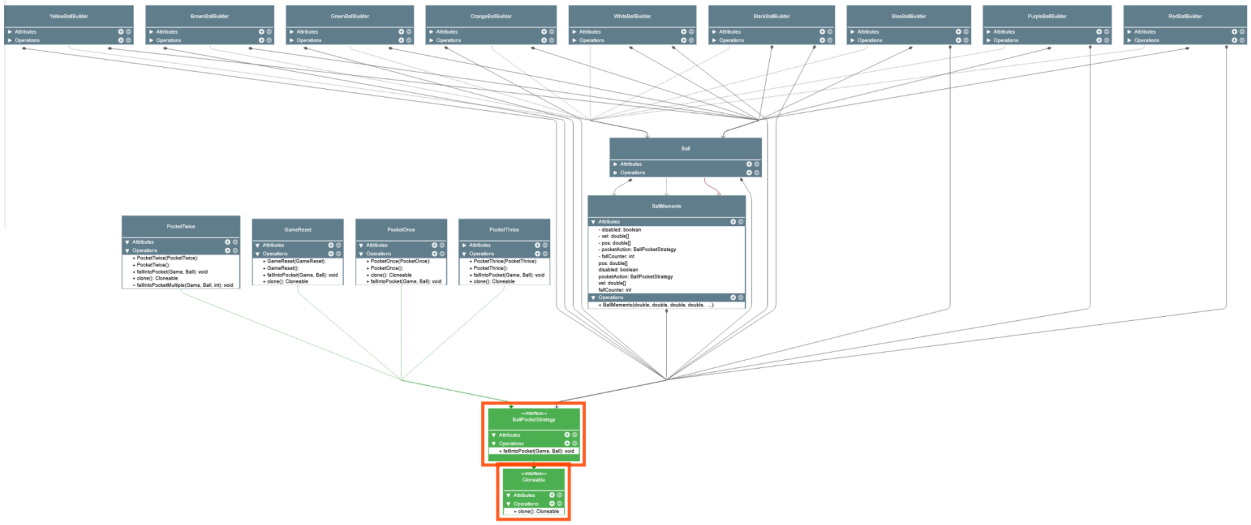
UML

Observer



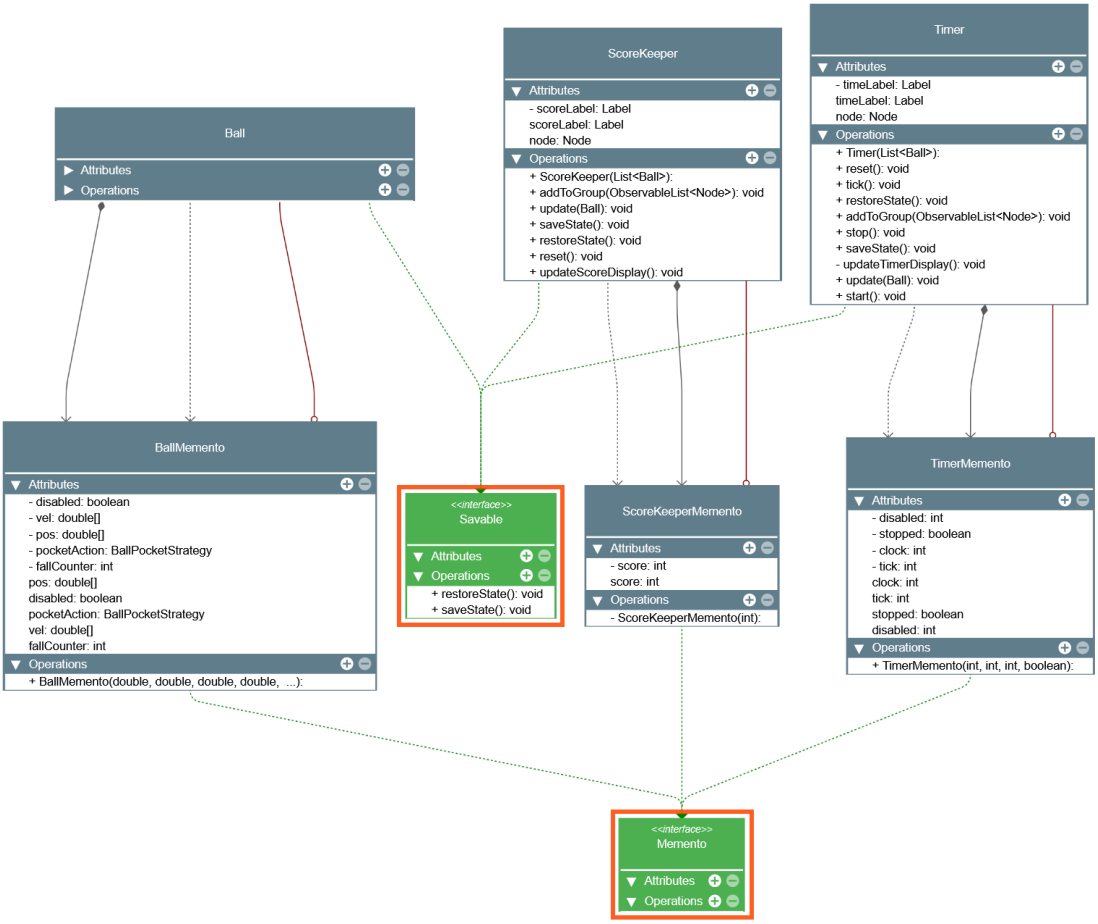
As mentioned above, the Observer design pattern was used to ensure single responsibility and dependency inversion. This also allowed for complex objects to indirectly interact with each other, decreasing coupling and increasing cohesion within the program. It also allows for the ball class to simply alert its list of observers when it needs, meaning that if there is another observer created later that depends on when the ball is disabled, then it could be created and attached to the ball without needing to change any code in the ball, thus increasing the overall extendability of the code.

Prototype



As mentioned above, the prototype pattern was used in this application to support future extensions and more complex behaviour in the PocketStrategies. This is necessary for the memento pattern to work seamlessly when these more complex (ie, needing to store its own variables) strategies are implemented as a deep copy of the ball object in its memento needs to be made. Using this decreases coupling as cloning the BallStrategyPattern is encapsulated inside of itself.

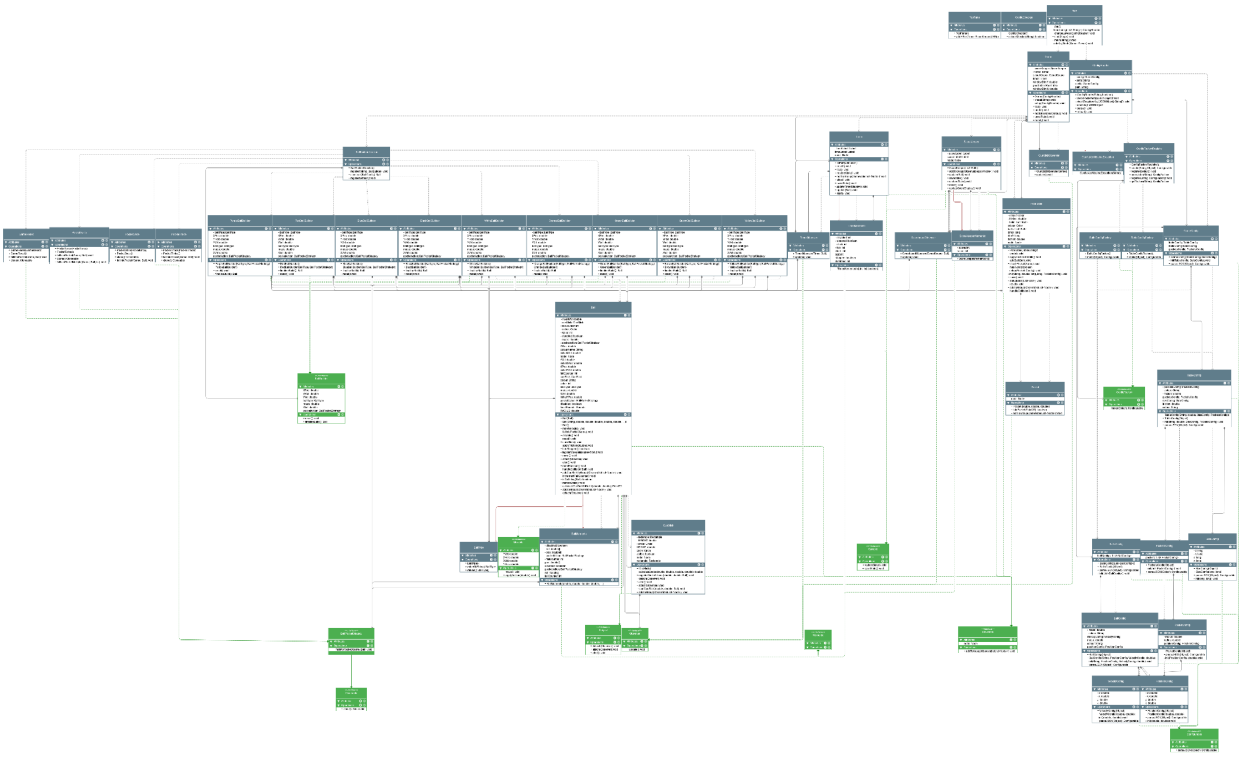
Memento



By using nested classes to create the Memento for the savable objects, the memento design pattern in this code obeys the creator design principle, as the savable object will contain their own memento. This pattern also delegates the creation of the saved state to the object itself that is being saved, this is beneficial because a nested class is able to access any private data inside the Savable object class, this was particularly beneficial for saving the Timer and ScoreKeeper objects as omits the need for getters and setters on many of their attributes making the data more secure.

Note: In order to restore the state from pressing the ‘U’ key, I needed a method in game to call in order to restore state for all of the savable objects. This means app, game and table were also involved somewhat in using the Memento pattern, however I would not say they are participants in how the pattern was used.

Final UML



https://drive.google.com/file/d/1zEY7UaQxgmwejrSZP4JmgJDaFoAI_FNB/view?usp=sharing

I couldn't get the UML to fit with all of the necessary information, the link provided will take you to the full resolution PNG where it can be zoomed into with full clarity.