Design Pattern: Factory

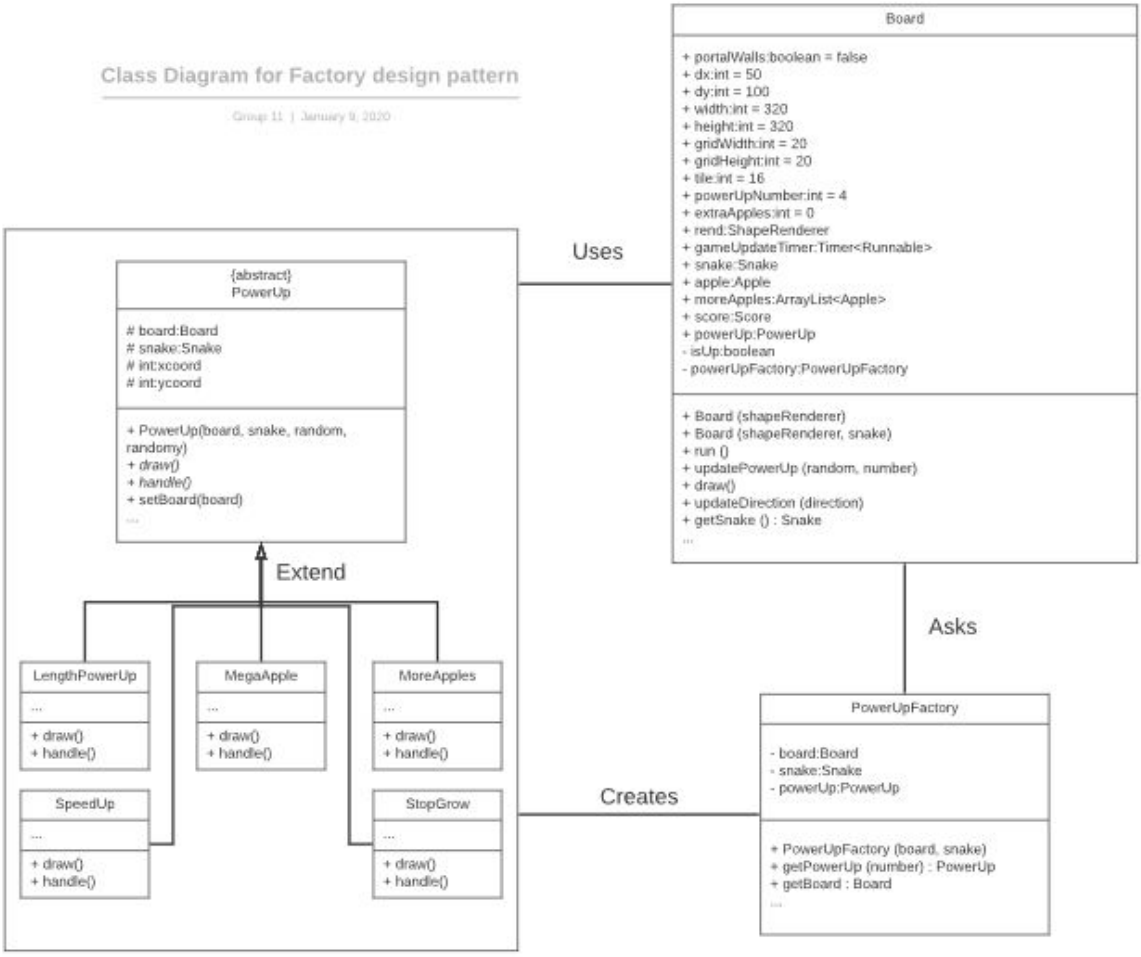
Why was this Design Pattern implemented and what are its benefits?

The design pattern: **Factory** was implemented since we needed to implement a method for randomly selecting and creating a specific powerup on the board. Since we have 5 distinct powerups and plan to possibly add more we decided to delegate the responsibility of creating a powerup to the **PowerUpFactory** class. This allowed us to decrease the overall coupling in our game's code as we put the responsibility of creating the powerups to this specific factory class. Overall this will lead to higher maintainability in the future: when we need to add or remove power ups all the relevant code towards creating them is located in 1 class (**PowerUpFactory**). Implementing the **PowerUpFactory** has also increased the Cohesion within our game as it has allowed us to increase the independence of our classes by giving the decision of which powerup to instantiate to a singular class (**PowerUpFactory**).
In conclusion we used the Factory design pattern to create the helper class **PowerUpFactory** within the game so that we can in general increase our code quality and maintainability by decreasing the coupling of our classes, and increasing our overall cohesion.

How was this Design Pattern implemented?

The implementation of the Factory was done by shifting the logic of creating the individual power ups to the class **PowerUpFactory**. Within this class there is a method called getPowerUp(powerUp) which contains a switch statement which dictates the powerup that will be returned when this method is called.
The **PowerUpFactory** is used by the **Board** class which instantiates the factory and uses it to generate the power ups. The Board class also contains the updatePowerUp() method which is called every run cycle to update the chosen power up randomly so that we can vary which power ups spawn on the board each time the board is drawn.

Class Diagram:



Design Pattern: States

Why was this Design Pattern implemented and what are its benefits?

The design pattern: **States** was implemented so that we can better separate the different states an object is in and differentiate between them. This design pattern allows us to change the behavior of the object (Game) based on its current state. This is important as the game should behave differently depending on which state it is currently in. Another benefit and reason why we used this pattern is that it allows us to make state transitions explicit. This means that whenever the game changes state the enterState() method is called and depending on which state the Game is transitioning to this function can have different effects. For example, while this might not be implemented just yet, when the game enters the FinishedGame state the enterState() function will be called and will send the player's score to the server. In conclusion the benefits of allowing us to separate an object's behavior based on state, and explicit state transitions are the 2 primary reasons why we chose to implement **States** in our game.

How was this Design Pattern implemented?

The implementation of **States** into our game was relatively straightforward and intuitive. We created an interface **State** and 3 seperate implementing classes for each phase of the game (**ActiveGame**, **PausedGame**, **FinishedGame**). Each state contains 2 primary methods, enterState() which is used upon entering the state, and observe() which is a function that is called continuously by the game. For the different states this methods have differing functions. The **Game** class contains a state object simply called **state** which contains the current state of said game. The class contains methods for changing the state and for calling the observe function of the state it is in.

Class Diagram: