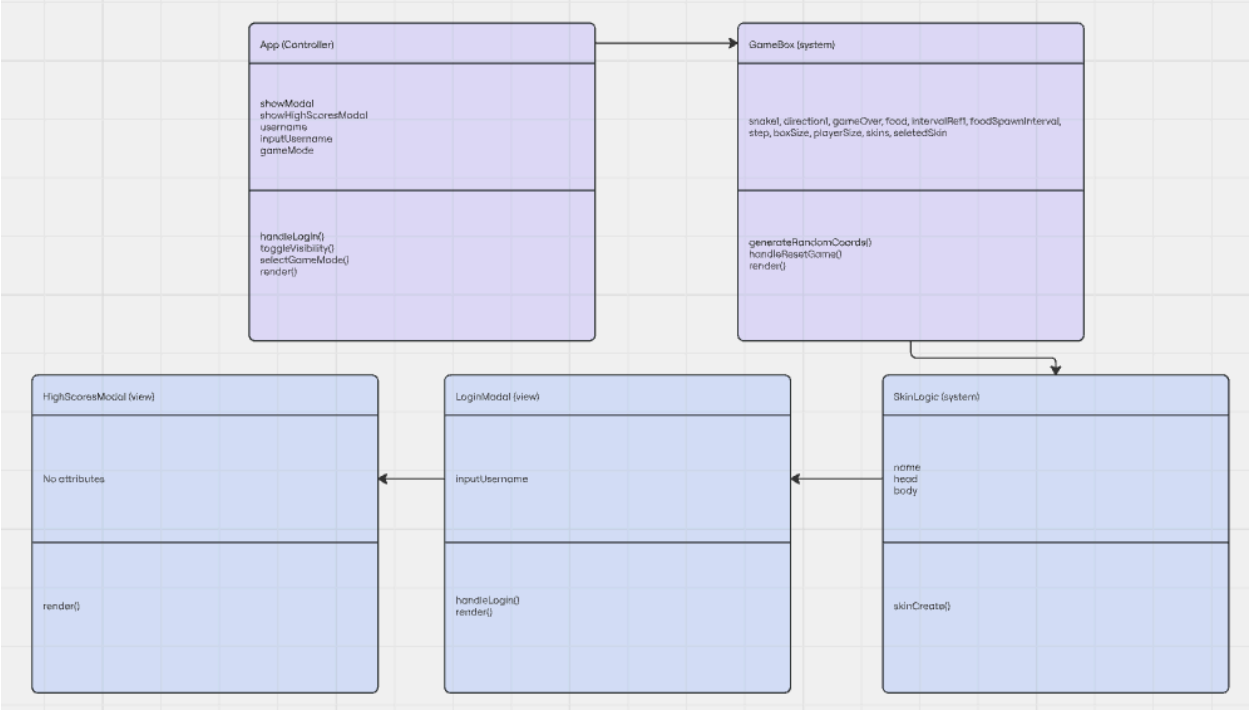
Domain Model - Super Snake

Super Snake is a web application developed using several react components to render different things stored in our database onto the screen. For the most part, each of our components are independent of each other. The skins class is a subclass of the snake, and the gamemodes and high scores buttons are subclasses of button.



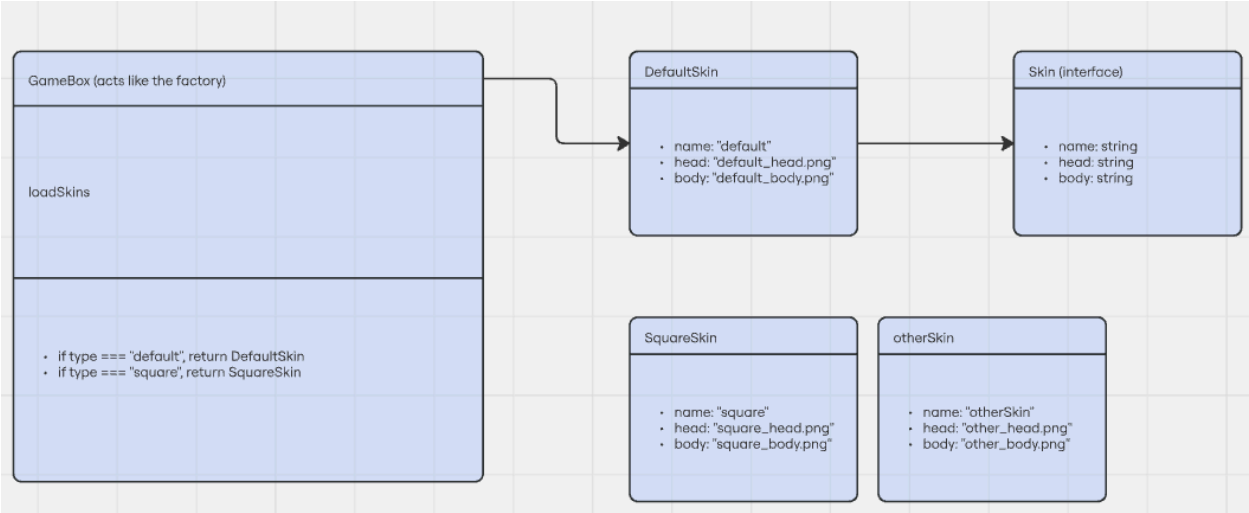
https://miro.com/app/board/uXjVIKRGFdk=/

The diagram above shows a detailed class design and which classes they use. The relationship is shown by the arrows between the different classes. The App component acts as the Controller and is the central point of the application, managing the state such as who's logged in, what game mode it is, and what skins the player is using. The app component uses GameBox, LoginModal, and HighScoresModal components to manage different tasks and user interactions. The GameBox component takes care of the game logic itself, and is in the System. It manages the snake's direction, where the foods are, and whether or not the player ran into a game-over screen. This component uses functions such as generateRandomCoords to spawn food at random positions. It also handles collisions with the foods and the game walls. This component is updated by the App component every time the game state changes. The LoginModal and HighScoresModal components are both part of view, and simply render the screen for logging in and viewing the high scores. Lastly, the Skin component is part of the system, and manages different skins. This project also focuses on separating things into

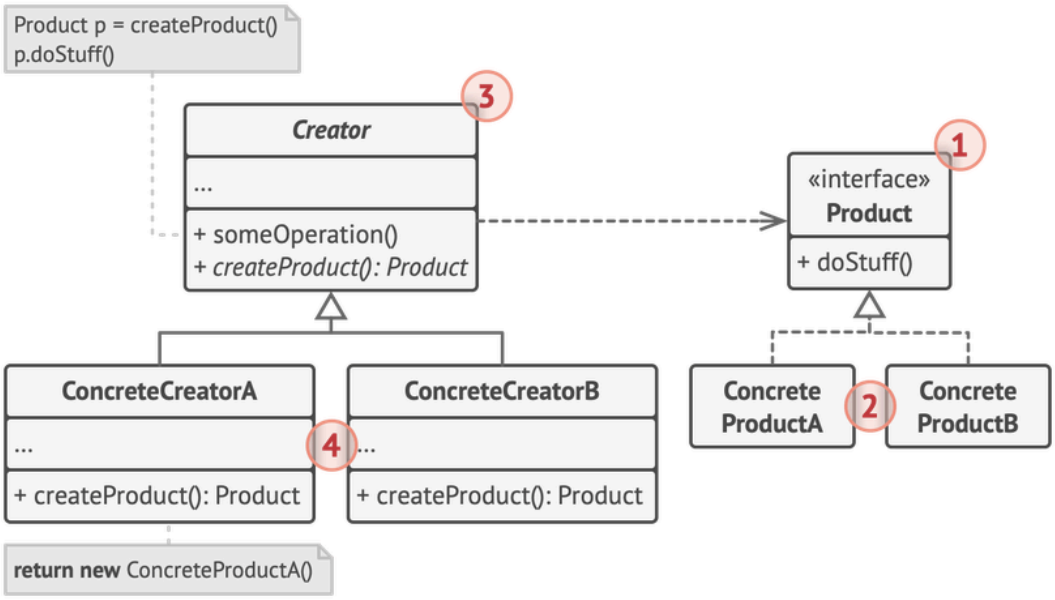
multiple different functions and files in order to prevent using redundant code, allowing for easy maintenance and further development and improvement without disrupting existing code.

Design pattern

Our project uses the Factory Pattern, among others. The loadSkins and SkinCreate functions work together to create the Skin objects. loadSkins generates a list of skin objects by calling SkinCreate for each skin name. SkinCreate then abstracts the creation of Skin objects, making sure things stay consistent. This makes it simple to add new skins without needing to modify the core logic of the game, and making it run slower if it had to cycle through a longer and longer list of skins every time new skins get added.



https://miro.com/app/board/uXjVIKRGFdk=/

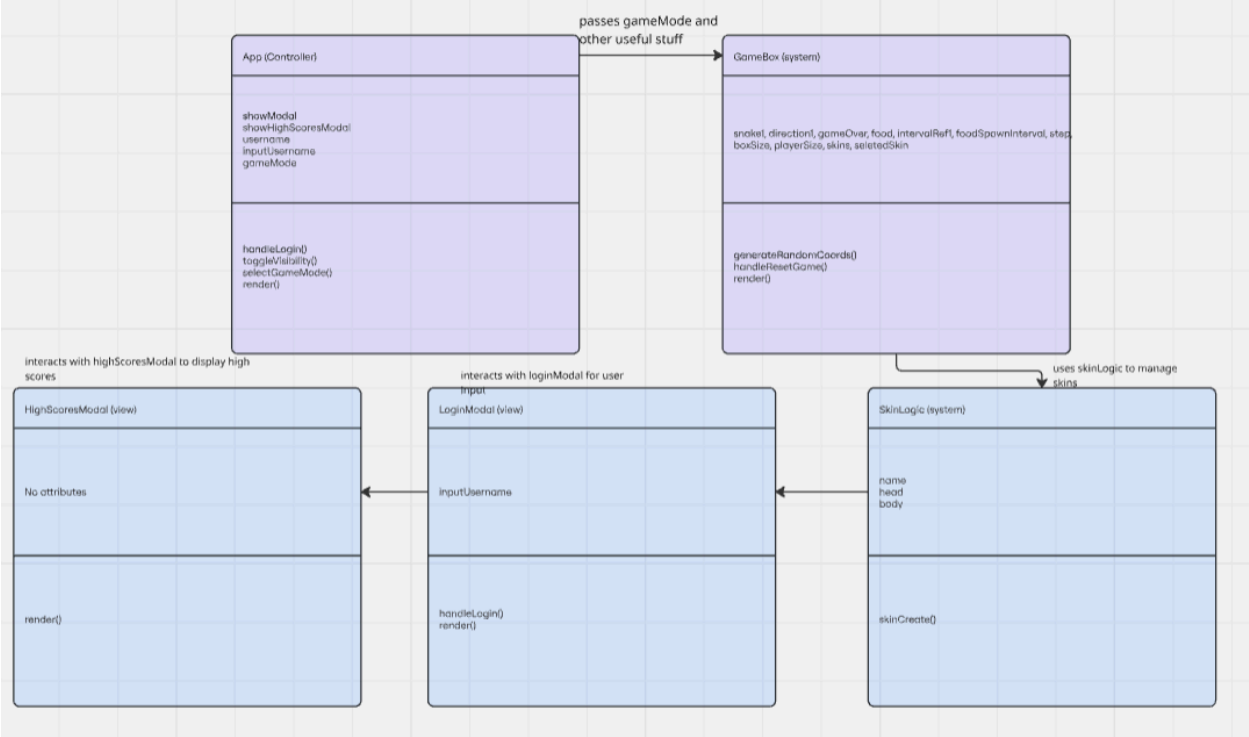


It does seem like our factory pattern follows a similar pattern to this one.

updated

Domain Model - Super Snake

Super Snake is a web application developed using several react components to render different things stored in our database onto the screen. For the most part, each of our components are independent of each other. The skins class is a subclass of the snake, and the gamemodes and high scores buttons are subclasses of button.



https://miro.com/app/board/uXjVIKRGFdk=/

The diagram above shows a detailed class design and which classes they use. The relationship is shown by the arrows between the different classes. The App component acts as the Controller and is the central point of the application, managing the state such as who's logged in, what game mode it is, and what skins the player is using. The app component uses GameBox, LoginModal, and HighScoresModal components to manage different tasks and user interactions. The GameBox component takes care of the game logic itself, and is in the System. It manages the snake's direction, where the foods are, and whether or not the player ran into a game-over screen. This component uses functions such as generateRandomCoords to spawn food at random positions. It also handles collisions with the foods and the game walls. This component is updated by the App component every time the game state changes. The LoginModal and HighScoresModal components are both part of view, and simply render the screen for logging in and viewing the high scores. Lastly, the Skin component is part of the

system, and manages different skins. This project also focuses on separating things into multiple different functions and files in order to prevent using redundant code, allowing for easy maintenance and further development and improvement without disrupting existing code.

Design Pattern

Our project uses the component-based design Pattern, among others, because we're using react. The program is structured into components such as the app and gameBox. These components are each modular and reusable, but most importantly, changes to one component won't necessarily affect the others. For example, gameBox holds the game logic for snake movement, food spawning, etc., and changes to that game logic won't interfere with the app component itself. This overall makes the code easier to maintain and further develop.