Singleton Pattern

The singleton pattern is usually used when there is a need of only one instance of a class existing at any moment during the lifetime of an application. It is sometimes considered an anti-pattern, as it effectively introduces global states inside the application (such as global variables in languages like C/C++). However, they can be useful when several components of the application share similar behaviour, in which case the pattern helps reduce redundancy and repetition.

Specifically, we have two singleton Utility classes which help improve the flow of the application, and the quality of its code. One of them defines all direct calls to the backend API in the form of several methods, which are then used inside other components, without them depending on the specific API implementation. The other is a table column utility, where the table column data and column creation methods are defined.

Since the latter is shared within several components (three at the moment), we believe that this, together with the API singleton component, have significantly improved the quality of our solution.

MVC Pattern

The Model-View-Controller is an architectural pattern which helps divide application code into three parts, each with different functions, but dependent on each other. In this design format, the way in which data is represented in our system (database) is independent from the data the user sees and provides with his actions.

The View represents the user interface, with all the data that the user sees and can interact with. In our case, this means a developer's games, achievements, and highscores for his games, along with all the buttons to navigate through the different components or modify them.

The Controller plays the role of an observer, being the mediator between the view and the model. Namely, it responds to the actions of the user, user input (potentially validating it) and sends it to the model. Additionally, it also notifies the View when it should change its state in order to display the state of the Model. In our application, the Controller is represented by the devGames container. It controls which of the available views is presented to the user, according to his actions in the web application, and sends and processes data from the API when necessary, with the possibility of displaying an error component if there are issues with the nodeJS server.

We have two models in our application, one is the server (specifically the postgreSQL database), where we send our requests to in order to send and receive data, and the other is the internal model of the application, held in react component "states" . While the first one is always the accurate representation of data, it could be computationally expensive to interrogate for it at each input of the user, so it is managed by our client side model.

Builder Pattern

This is a creational design pattern which provides a flexible design solution for developers to create different representations of (potentially complex) objects. In our case, it allows us to create a various number of tables with different data (e.g. games, achievements, high scores or more) and different behaviour (e.g. adding/editing/deleting rows, button listener reaction), along with the possibility of displaying one or more of them at the same time.

Implementations of the pattern can be seen with the table components, all of which are instantiated at creation with functions and objects they need to use, but do not (and should not) care how they work or are implemented.

Strategy Pattern
Another pattern which is used in our application and was essential for eliminating redundancies in the code, is the Strategy Pattern. It generally allows for various implementations of the same behaviour to be easily interchangeable and vary from user to user.

Since the Game Developer Web Application is mostly concerned with displaying data in tables (for the time being), it made sense to be able to easily switch between them without too many changes in code (or in the application code), and we do this through our pageContent.

Separation of concerns
Separations of concerns is a design principle which concerns dividing the codebase into separate independent (but communicating) parts, each accomplishing a specific goal. The main advantage of implementing this in an application is its portability and extensibility, as individual components are not concerned with how another component does something, but rather in the end result.

As an example of our use of it, our controller component does not care how or where the calls to the API are made, it just sends and receives data from the Api utility class. Similarly, table components are not concerned with how the columns are created or what property each columns has, as this is defined in the Column utility class. It is obvious that in the case of adding new features or changing existing ones, changes would be minimal, as there is little functional dependency between components (as in, components directly depending on how other components completing their tasks).

Chain of responsibility
The main goal of this design pattern is to avoid coupling the sender and receiver of the request together. In contrast, it recommends creating a chain of objects which process both sending and receiving the requests, which, as with the separation of concerns principles, improves code quality and maintainability.

In our application, when a user wants to create a new game, for instance, the request is handled by various components, up and down the “responsibility chain”. When the “New” Button is clicked, a modal component handles the user input (and validating it), and then uses a callback function to the controller component. There, it receives the api request from

the Api Utility component, and saves the newly added row in its state. This means that it gets sent back down the chain (as “props”), to the table component and then to the modal, which sees that the row is a new one, and updates the user interface with the newly added row.