Code Quality Review: Growing Gamers

Table of Contents:

1.0 Design Patterns/Formatting

1.1 Application Patterns:

- Each app window uses the singleton pattern to prevent more than one copy from existing.

- The MVC pattern is implemented in the app. The background window acts as the controller by deciding what to show at what time and handling data movement within the app. The in-game window and the server act as the model, they collect and provide data to the controller. The server also performs functions at the request of the background window. The launcher window provides the view, it shows messages and information to the user.

1.2 Server Patterns:

- The server has a sort of internal MVC pattern. The model is the MongoDB database. The responses sent out by the nodeJS server can be thought of as the view. The controller is the nodeJS server itself.

1.3 System Architecture:

- The App and the parent portal make up both the view and the controller.For the app, the background window handles logic and the launcher window handles the view. For the parent portal, the JS file handles logic and the HTML/CSS files handle the view. The nodeJS server alongside the MongoDB database make up the model, they handle and process the data that the respective controllers use to decide what happens.

1.4 Formatting

- The formatting overall is rather consistent. All the get and post requests are formatted the same in server.js. Most variables, names, and ids are named with consistent standards based on their class throughout the code base. The format of data given and expected in post requests is consistent and well formatted. File naming conventions are consistent and well respected.

- The formatting does not all hold to best practice. Variable names are often too long. Some of the functions are simply too large (will be further discussed later). Some files are too large and are clunky (i.e server.js). Comments are brief and perhaps not descriptive enough although they seem to be generally meaningful and increase the readability of the code.

2.0 Non Functional Requirements

2.1 Readability

- There is dead code commented out in a number of places, this seriously hinders readability of the code and exacerbates the unreasonable size of some of the files.

- The comments that are there are meaningful and readable. There is a lack of annotation when describing what a function accomplishes (what information does it expect, what does it promise to deliver, etc.). This is particularly noticeable in the background window code.

- Comments are well used to create visual separation within large functions (i.e. get-message from server.js)

- The structure of functions is logical and highly legible. The organization of functions (clustered by purpose) helps manage the larger files.

- Variable names are meaningful and consistent although some of them are quite long.

- Function names are meaningful and consistent although some of them are quite long.

- HTML in general is well indented and readable. Some lines containing multiple <divs> in the same line are more difficult to decode but still reasonable. A great deal of the HTML is repeated and so by decoding one component insight into many components is gained.

- CSS is written consistently but there is some dead/unused code and a lack of organization. A small clean up job of the CSS will make it extremely readable.

2.2 Reusability

- Reusability is generally good. There are exceptions that will be further discussed.

- There is a clear effort to break functions into single responsibilities and support adequate separation of concerns, this supports modularity. Nonetheless, some functions have too much responsibility and functions should be extracted (particularly in server.js and parentPortal.js).

- In general, modularity is good. There is no convoluted net of dependencies. The code is structured so many independent modular functions are used to build larger more complex functions above, it is a pyramid and not a net. This too promotes loose coupling.

- Modularity is reasonably implemented. Any given function has a known input and output. So long as a function exists that can handle the request passed and provide the promise the structure and mechanical logic of a function can be changed at any time without damaging any other code. This greatly supports loose coupling.

- A point for improvement of modularity is making queries to the MongoDB server. There are a few spots (in server.js particularly in some post requests) where the query is hard coded rather than reusing a single module. This is some WETness that should be made DRY.

- Information hiding is not well done. There are variables that should be better protected as constants instead of variables. Privatization of functions should be increased. A great increase in data hiding should be achieved (particularly in server.js and background.ts).

2.3 Reliability

- There is some attempt to support good reliability in some parts of the code. In general, reliability appears to be a weak point.

- Reliability is supported when queries are made to the database, there are checks in place in case connecting to the database fails.

- Some amount of regex is in place for the parent portal form. There is room for improvement but at least there are some checks in place.

- There are checks in place in particularly vulnerable portions of the code (i.e. spots where issues arose).

- Overall there are not enough checks in place, this is a weak point in our code that will need to improve for reliability and scalability. A special effort should be deployed to increase reliability and security.

2.4 Scalability

- There is some degree of scalability present in the code but additional techniques will be required to make it viable at a large scale.

- Scalability for users with more than one child is currently poor. Only one child can be bound to a phone number and only a single number can be used on a given machine at any given time. This is something listed as future work to make the product scalable in terms of type and number of users. Not only should parents of multiple children be considered but also individual adult users looking for help in self regulating or stat tracking.

- By using amazon EC2 we can easily use more or less resources as they are required, this increases scalability a large amount. Nonetheless a limitation is our architecture, currently we only have a single centralized server serving all users, this will present performance limitations at a larger scale. To overcome this a number of strategies may be employed, these include having multiple servers serving users from different regions or using a microservice style architecture where each user gets a micro instance of the server dynamically created to personally serve them (using kafka or something similar). Combining these strategies with big data methodologies for managing the databases could transform this product into an infinitely scalable program.

- The scalability of the code in terms of our ability to grow the code base is quite solid. There is no blatant reason for future work to interfere with existing code or vice versa.

- One challenge in terms of scalability is that some values that should be universal constants and only be defined once are actually declared in multiple places. So to make a change or scale the product, many lines of code must be modified in order to properly make a change.

2.5 Usability

- Usability of code is quite good. There are a few minor pain points but overall the code is quite usable.

- Good implementation of the single-responsibility contributes to making the code simply usable.

- While there are some shortcomings in reliability, expected use cases of the code are handled well overall.

3.0 SOLID Principles

3.1 Single-Responsibility

- The Single-Responsibility Principle is generally respected with a few points for improvement.

- A few functions have too much responsibility and functions should be extracted to better enforce the single-responsibility principle (particularly in server.js and parentPortal.js).

3.2 Open-closed

- The open-closed principle is followed well throughout the code.

- The inheritance between app windows respects the open closed principles

- We make little use elsewhere of inheritance, so there is not much else to say about the open-closed principle.

3.3 Liskov Substitution

- The Liskov Substitution Principle is well followed wherever relevant.

- There is only one significant example of inheritance in the code being the ingame and launcher window objects inheriting from the appWindow parent class. Any modifications to the child classes are extensions and do not interfere with the normal functioning of the parent class functions.

3.4 Interface Segregation

- This criteria has little relation to the work so far accomplished.

3.5 Dependency Inversion

- Abstractions and interfaces were not implemented in our program, so the dependency inversion principle was not followed.

4.0 Conclusion

The quality of the code is reasonable for this stage of development. Additional implementation of SOLID principles, practices that promote scalability, and reliability should be high on the list of priorities moving into future work. Strong points of the code are readability, usability, and reusability.

There are a few smells present.

- There is dead code that is just commented out, it should be removed entirely.
- There are files that are too big, they should be broken up into smaller files.
- There is a lack of error catching/exception handling throughout, some of the more crucial areas have good exception handling but overall the code lacks.
- The file structure of the overwolf app is weird, why is there a folder named .ts inside of the folder called OverWolf-App?