OOAD Project 7

Dane Rieber

May 3, 2023

1 Project Info

Name: Object Clicker

Team Members: Dane Rieber

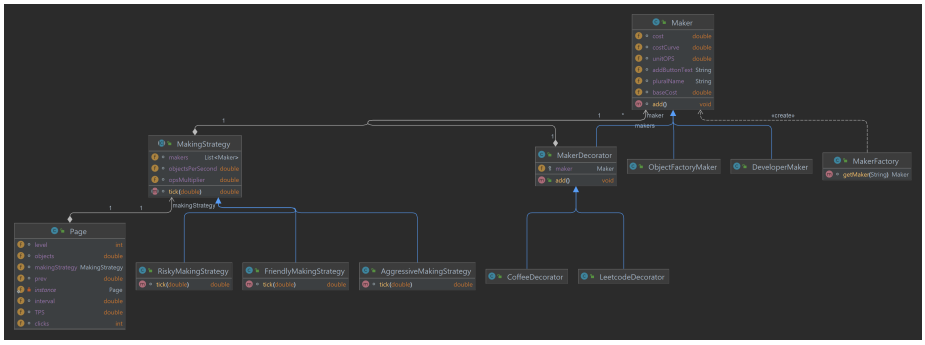
2 Project Summary

All features from Project 5, except for saving/loading game state, were implemented. As complexity increased, and since I have no experience with browser cookies or local storage, saving/loading became something that was not a priority, since I rather needed to focus on completing the full functionality of the game itself (most importantly, the OOP-related code).

Other than that, the game is complete with object and OPS (object per second) counters, a progression system, automated object "makers" that can be purchased (using objects), and upgrades. It also complied with the constraints of fitting on 16:9 screens and running all logic client-side.

3 Class Diagram

The final class diagram was able to remain exactly as detailed in Project 5! All of the planned classes and their relations were able to be programmed as planned, so the UML diagrams for Project 5, 6, and 7 are all the same.



4 Third Party Code

The Svelte framework was used for easily creating reactive webpages that run client-side. See `https://svelte.dev/` for more information. Much of the structure for my Svelte files are taken from code examples and snippets on their tutorials. The rest of the code is original, and relies on my existing knowledge of Typescript and my experience with creating game loops.

5 OOAD Process

- Although OOP can be used with Svelte, it is much easier to use Svelte in a FP style. I originally prototyped object clicker without any OOP and then added OOP elements afterwards. During this, I encountered strange bugs and weird workarounds but eventually was able to get it to work, although it did not feel like how I was intended to use the framework. As I was new to Svelte, I did not have this knowledge that I now have in hindsight.

- Using OOP started to fragment my code across many different files and make my program logic difficult to follow. I would have preferred to write most of the program logic in the Svelte files themselves rather than splitting it up into abstractions that seemed to over-complicate things. Rather than being able to follow the program's relatively simple procedures in one large file, you instead have to understand the purpose of each smaller file before being able to follow the control flow.

- Type checking became so much of a pain with the decorator pattern that I eventually just had to include a "type" property on my decorators to be able to property identify them. If I wanted to add another type of Maker or Decorator, for instance, it would require altering code in many different places and duplicating much of the code. This is, however, more of a flaw with Typescript's inheritance system rather than the design itself, but it made for some messy and unwanted code duplication to just get things working properly.