Prior Learning Assessment Portfolio

Brady Underwood (08646866)
22823 SE 268th Pl
Maple Valley, WA, 98038
11/11/2024
425-358-1910
underwood.brady@gmail.com
CSCI 495 - Field Work/Practicum

Table of Contents:

Letter of Intent:

With this portfolio, I intend to demonstrate the learning outcomes in software design, implementation, and debugging through three substantial real-world projects I have developed: LocateCRM, AdafruitGFX, and Red Trade. These projects showcase:

1. Design scalable software systems, including:
    a. Service-oriented architecture (SOA)
    b. UML modeling and documentation
    c. Data modeling and entity relationship design
    d. User flow mapping
2. Implementation of software:
    a. Applying fundamental data structures and algorithms
    b. Writing maintainable, modular code
    c. Using appropriate design patterns
    d. Integrating multiple technologies
3. Debugging of full-stack applications through:
    a. API troubleshooting
    b. Frontend and backend debugging strategies and tools

Additionally, throughout this portfolio, I demonstrate my ability to effectively communicate technical concepts to lower-division CS students by breaking down complex topics into clear, digestible explanations while maintaining technical accuracy.

The included projects align with the learning outcomes of CSCI 495 and reflect knowledge of previously learned computer science material.

Sincerely,

-Brady Underwood

Design - LocateCRM:

Introduction:

In this section I aim to explain the design process for a “production-level” app from basic UML modeling to ultimately picking the tech stack. I will attempt to explain ideas to lower division CS students in a way that is digestible and informative while still remaining technical in nature.

Background:

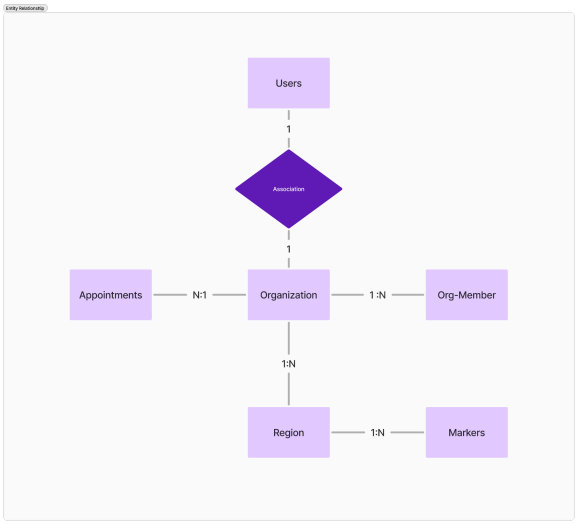
During the spring semester of 2024 I created a location-based customer relation management app called “LocateCRM” that runs on IOS, android, and the web. Other examples of CRM tools include hubspot, salesforce, or monday.com. The idea for this project was inspired by my brother Callan who started a door to door window washing company here in Bozeman. They needed a tool to mark what houses they have been to on a map, track leads, and keep track of jobs on a calendar. Upon discovering there were no good tools out there that weren’t hundreds of dollars a month they reached out to me to develop a similar app they could use. I decided to take on the task. In order to replicate the apps they were looking at I needed the following features: an interactive map, a database to store leads and markers, authentication, and a calendar service.

Design:

The application was designed with a service-oriented architecture (SOA) to promote modularity and maintainability. SOA divides core functionalities into distinct services, each containing individual business logic.These services are accessed through provider classes, which act as a layer of abstraction. This follows the software design strategy of separation of concerns which encourages dividing a program into unique sections, each addressing a separate concern. The alternative approach is a design-first or monolithic approach which relies on a singular service with code designed around exactly what the front-end needs. SOA was well-suited for this app due to the service requiring vastly different API’s and libraries. Additionally, by isolating code, the risk of unintended side effects is minimized. In this architecture the flow of information through the app is then as follows:

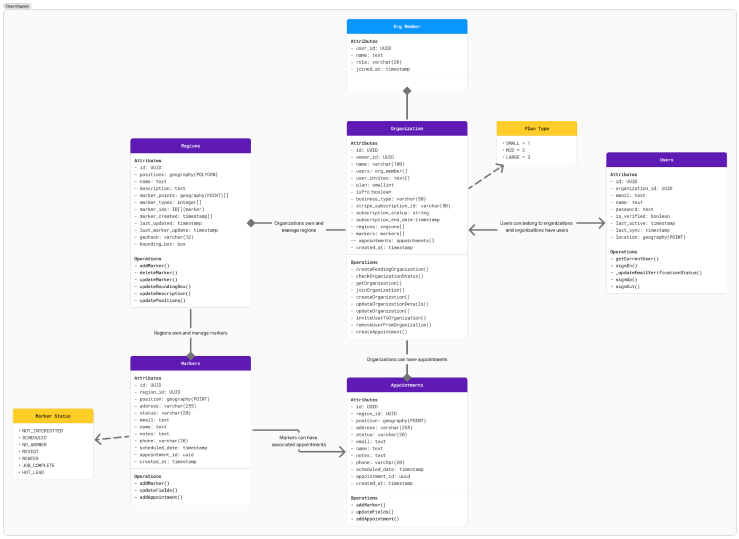
Essential business functionalities, such as database’s, mapping, scheduling, or authentication are written as independent services. Frontend components interact with these services through provider classes, which enforce strict data models ensuring type safety. These providers are

implementation agnostic in regards to the services they call, meaning they “provide” a set of public functions that the frontend can call without worrying about the underlying service implementation. This prevents bugs from propagating throughout the app when updating the backend (for example if switching authentication from one library to another). Finally, specific screens in the app asynchronously watch these provider classes, following the observer design-pattern. This means that if the state of the provider is updated, all its dependents (the "observers") are notified and updated automatically. This is particularly useful for authentication related scenarios where UI elements must adapt to changes in user status or allow or deny access to certain pages.



With the architectural foundation established, the next step involved defining the data model, storage strategy, and user flow. I will now go through each of the diagrams that contributed to the software design of the app, and ultimately influenced the decided tech-stack.

When first designing software it is good to have a solid understanding of what data the program will be interacting with. In this case, lists were created to show any potential data that needed to be stored. When creating data models, it is important to list the data needed without worrying about relationships or implementation. For this app data models for organizations, users, and appointments were created.

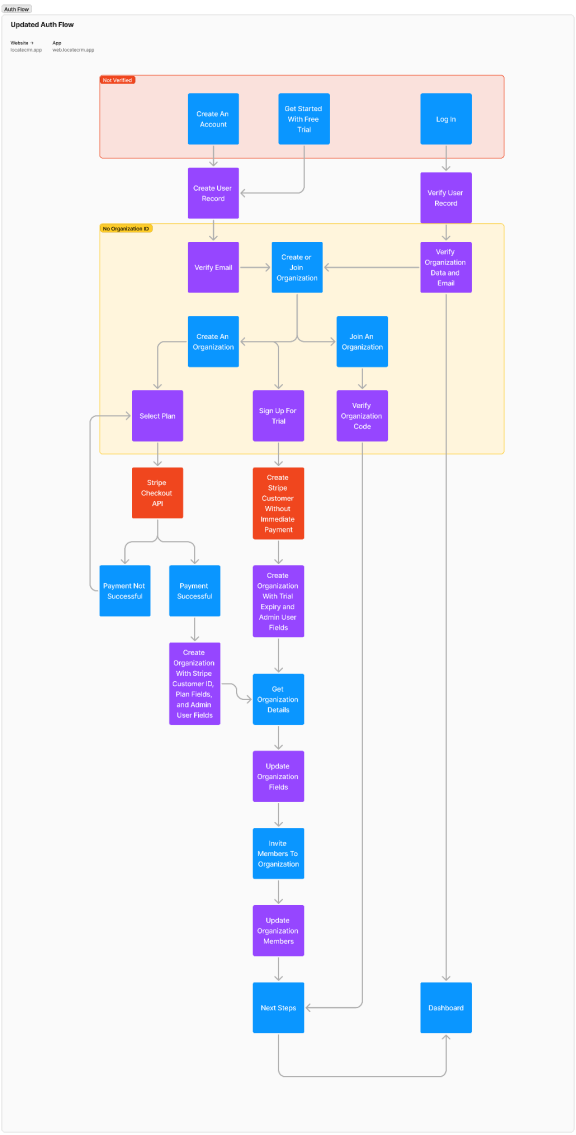


To visualize relationships between these data models, an entity-relationship diagram was created. This diagram provided a clear overview of how the different data entities are connected.

Once the data model and relationships were understood, the next step was to design the class structure. Class diagrams were used to represent the

classes, their attributes, and their interactions. These diagrams served as a blueprint for the actual code implementation.

Lastly, once what data is needed and the relationships between them are understood one can then start thinking about how the UI/UX design incorporates. The most helpful tools for that are user-flow diagrams and UX designing.



To ensure a seamless user experience, careful consideration was given to the user flow. A user flow diagram was created to map out the various user journeys, including authentication, organization setup, payment processing, and member management. This diagram helped identify the necessary screens and backend services required to support these flows. For example, this diagram illustrates the authentication process, starting from the login/signup screen and progressing through organization creation, payment, and finalization. It also highlighted the different access levels for authenticated and unauthenticated users.

Conclusions:

Based on the diagrams, I was able to make informed decisions about the technology stack and codebase structure. Given the requirement for both web and mobile access, Flutter was selected for its cross-platform capabilities and robust ecosystem of libraries. It also has a strong selection of libraries it integrates with such as stripe for payments or any database or authentication. For the backend, Supabase was selected as a comprehensive solution for authentication and database services. Its PostgreSQL foundation, supported by the PostGIS extension, is ideal for handling the geospatial data associated with markers and regions. This choice offers advantages such as scalability, flexibility, and cost-effectiveness compared to alternatives like AWS or Firebase. It also allows for easy support of partitioning and inheritance which are crucial when querying thousands of markers and customer data. Finally, Mapbox was chosen for the mapping service

due to its superior flexibility in visualizing regions and markers, seamless integration with Flutter, and competitive pricing.

Implementation - AdafruitGFX:

Introduction:

The goal of this paper is to describe the implementation process for AdafruitGFX, a visual editing tool (like microsoft paint) that translates designs directly into code compatible with Arduinos and other microcontrollers.

Background:

In my freetime I have always been interested in working with microcontrollers and microcomputers to automate and improve things around the house. Unfortunately, as someone who cares about the design of things I constantly ran into a problem, any display I wanted to connect to an Arduino used a separate graphics library and had different dimensions. Furthermore, there was no easy way to test how display code would look unless you waited for it to compile. For every slight adjustment in graphics you want to test out you have to recompile all over again, and if you ever made the decision to change displays you would have to write the code all over again accounting for the new dimensions, not knowing if everything would fit. That's where I came up with the idea for the Adafruit Visual Editor, based at its core on the AdafruitGFX library, which is the most popular library for displaying graphics on small displays and written in C. This tool allows users, such as myself, to test how different shapes and text look on different sized displays instantly and generate boilerplate code for users to copy into their editor of choice (also providing details about the unique library for each display). No more struggles with sizing, formatting, or waiting minutes to get an idea on how something would look. To accomplish this I had to utilize everything I learned in my data structures and algorithms class at MSU to write reliable data structures for the shape objects and algorithms for the drawing of shapes.

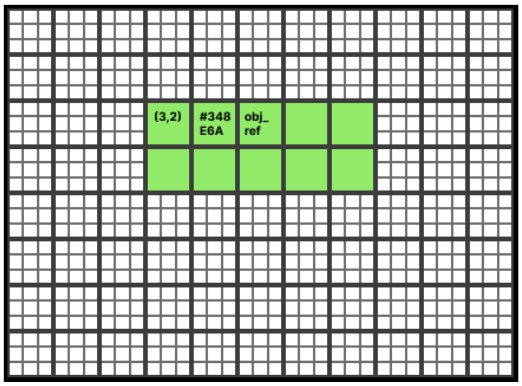
Implementation:

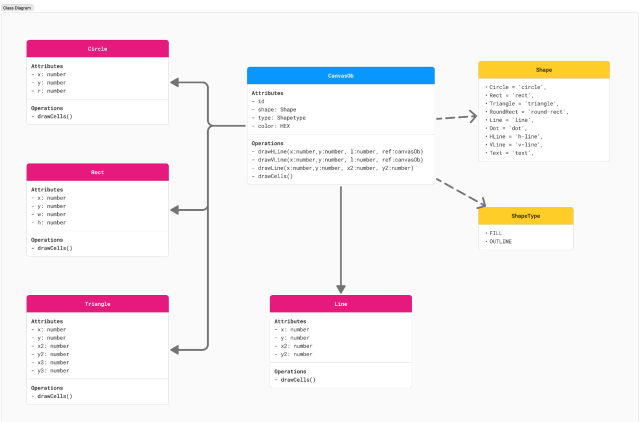
The frontend was built using SvelteKit, a framework that allows developers to write native TypeScript and HTML, leveraging a component-based architecture. This architecture promotes modularity and code reusability, as components can be defined once and used throughout the

application. Components can be as simple as a Button or a Title, or as complicated as an Accordion dropdown or entire Canvas. This also allows for a separation of concerns and simple code organization, allowing specific logic to be contained within the component file.

While the choice of frontend library is important for optimizing performance and developer experience, the core functionality of the application is independent of the frontend framework. Therefore, the rest of the paper will focus on the implementation details of these core components.

Inspired by the Adafruit GFX library, the virtual canvas in this application is a grid of cells, each characterized by its x and y coordinates, size, color, and a reference to the shape it contains. When a shape is drawn, the corresponding cells are updated to reflect the shape's properties. To render the canvas, the application iterates through the cells and draws them onto the HTML canvas element. The "Cells" contain the following parameters; x cord, y cord, size(w=h), color, and a reference in memory to the object that occupies the cell. When a shape is drawn, the shape is stored in memory and the cells it overlaps are updated to reference the object. To display to the user the canvas simply just redraws the list of cells that have been updated with the corresponding color. In order to generate code it is as simple as taking the list of objects stored in memory, and then creating the respective line of code based on Adafruit GFX's graphic library. Each shape is a class with parameters and inherits from a parent CanvasOb class which provides parameters for defining shape and fill type and simple utility functions for drawing lines.





Derived shape classes, such as Triangle, extend the base class and implement specific drawing algorithms. For instance, the Triangle class takes six parameters to define its vertices and overrides the drawCells method to calculate and update the affected cells. In code, the triangle class calls

“super()” in order to access the methods and properties of its parent.

To support undo and redo functionality, a linked list data structure is employed. Each node in the list represents a complete canvas state, including the positions and properties of all shapes. When a user performs an action, the current state is pushed onto the list, and the previous state is stored as the parent node. This allows for efficient undo and redo operations by simply traversing the list.

Another interesting implementation is a bitmap text algorithm which stores text styles in bytes. This works by representing each character by a 5x7 grid of pixels. Each pixel in that grid is either ON (1) or OFF (0). When converting these 1’s and 0’s to on pixels and off pixels this creates a simple bitmap pattern for each letter. By reverse engineering the code from the original C library, the visual editor is able to show a full representative font system.

Conclusions:

The implementation of AdafruitGFX is largely based on a core understanding of basic data structures and algorithms, as demonstrated above. By leveraging techniques like linked lists for undo/redo functionality and bitmap representations for text, the project is able to emulate the original C library's capabilities. The virtual canvas, with its cell-based structure, provides a flexible and efficient foundation for rendering various shapes and patterns. This project serves as a testament to the power of fundamental computer science concepts and their application in practical software development.

Debugging - Red Trade

Abstract:

The goal of this paper is to describe the debugging process for Red Trade, a financial visualization startup with the goal of providing advanced and customizable financial analytics to everyday retail investors

Background:

This project was an ongoing project from the years 2020 to 2023 constantly evolving as I learned more about financial visualization and software. It was ultimately written in Sveltekit for the front-end, node.js for the backend, and used data provided by TD Ameritrade's developer API. Throughout the process I had to do extensive debugging with both the API and my own backend code. I will describe the process I used when attempting to debug parts of the app and the API that was provided.

Debugging:

Debugging for full stack applications is complicated because there can be separate debugging tools required for each part of the tech stack. On the front-end one might use chrome developer tools, while on the backend one might instead use simple logging or Postman for API testing. A strategy that works on either side of the coin however is the native IDE debugger which will be explored more later in the paper.

The core workload of debugging for this project was API integration, where the initial version of Red Trade utilized the TD Ameritrade developer API, which has since been discontinued and migrated. Common issues that occurred included authentication errors, data inconsistency, and rate limiting. Authentication errors would occur when access tokens expired and for access to the API TD Ameritrade required routine “session tokens” that lasted an hour at a time. This led to inspecting network requests via Chrome DevTools, and routinely verifying access tokens. It was also important to examine the API documentation as it listed exactly the requests and responses it could provide. The next issue was data inconsistency, where within this project each user had different accounts set up with TD Ameritrade, and different types of accounts would return vastly different data models from the API. Debugging this required comparing the API documentation for different account types and performing strict type checking on the frontend with specific models for user data. If the returned data didn’t fit these models an error

was thrown, immediately notifying either the user or the developer that an issue had occurred and where it occurred in the code. The last smaller issue to debug was rate limiting, this was only an issue briefly and was simply solved by writing better code that reduced the need for as many API calls. A helpful tool for inspecting API's is Postman which enables sending mock requests to various API endpoints and verifying the response. This helps isolute issues with the API itself rather than code-related problems.

At the same time of API debugging, debugging took place on both the backend and front-end whenever writing new code with regards to data consistency and manipulation. There are a few important strategies to follow when encountering errors. The first, and easiest, is strategic logging of data at various points. Logs helped identify unexpected values or errors at different stages of the app. Another important strategy is breakpoints in VS Code which is my editor of choice. Breakpoints allow you to step through specific functions and visualize specific data changes and inspect variables at runtime. A specific example of this was when attempting to visualize the top growing stocks of the day, a function was created to query the API to return the top 50. This caused an issue in the UI because the API employed pagination (providing 10 stocks at a time) while the UI expected to display all 50 at once. At first the issue was thought to do with the UI not showing the stocks properly. However, after breaking up each component of the app into its unique stages: API request, API handling, Array Conversion, UI. It was discovered that the error was actually in the API request and handling stages.

In general there are best practices to follow to reduce the amount of time spent debugging. Whenever dealing with API's or code where you don't always know the result, use try-catch blocks or some data validation to catch errors before they propagate to the user. Another method is step-by-step code isolation, where if the cause of a problem is unknown step through the code function by function writing tests or logs for each to make sure the data is fully understood. Furthermore, while writing code it is sometimes helpful to follow Test Driven Development (TDD). TDD involves writing automated tests before writing the code in critical areas of development. This ensures the code meets the specific requirements of the project and will work as intended. Through writing tests first, developers can focus more on writing clean, well-structured code less prone to bugs.

Conclusion

By utilizing these debugging methods and tools, I was able to effectively troubleshoot and resolve various bugs within Red Trade. This ensured smooth functionality for users of the application and a better developer experience.

Bibliography:

https://www.educative.io/answers/ssr-vs-csr-vs-isr-vs-ssg

https://aws.amazon.com/compare/the-difference-between-soa-microservices/

https://www.geeksforgeeks.org/test-driven-development-tdd/