RTS Analyzer

Software Engineers:

* Cody Strange – ETL Developer
* Jaden Albrecht – Database Administrator
* Hoyoung Kim – DevOps

Institution: Utah Valley University

Course: CS4550-601 2024 Spring

# Abstract

The RTS Analyzer is a comprehensive software tool developed by students at Utah Valley University for the CS4550-601 course in Spring 2024. This application is designed to process and analyze StarCraft II replay data to extract and display strategic gameplay metrics such as build orders and win rates. Utilizing technologies like Django for backend development and Flutter for frontend interface design, the RTS Analyzer aims to enhance real-time gaming strategies by allowing users to overlay build orders onto live game scenarios. The tool leverages a Python module, sc2reader, for accessing replay data, and implements rigorous coding standards to ensure robustness and maintainability. The project integrates advanced data processing techniques to determine similarities between player actions and predefined optimal build orders, offering a significant tactical advantage in gameplay. The application is structured to meet high standards of performance, scalability, and security, addressing both functional and non-functional requirements meticulously. This paper will discuss the design, development, and operational strategy of the RTS Analyzer, showcasing its potential to revolutionize game strategy development and analysis.

# Acknowledgements (from Cody Strange)

I extend my heartfelt thanks to several professors whose guidance was instrumental not only in shaping the design of this project but also in providing the foundational knowledge that made it possible.

* **Professor Reza Senati**: His review and advice on our ER Diagram and database schema were invaluable, as were the lessons in his Database Theory class that introduced me to relational databases. Our collaborative work on a database research paper bolstered my confidence and ability to implement these designs effectively.
* **Professor Lynn Thackery**: He meticulously reviewed our UML, data flow, and architecture diagrams, ensuring we could properly design and document our entire project. His teachings in the Principles and Software Patterns class, especially the principle "Program to an interface, not an implementation," have been the cornerstone of this project’s maintainability and scalability. His influence has significantly reduced the time I spend debugging, making this complex project the smoothest I have undertaken.
* **Professor Craig Sharp**: From his Software Engineering One class, I learned the crucial importance of design and documentation in software projects. His class not only deepened my love for software engineering but also inspired me to switch my major from Computer Science to Software Engineering, marking a pivotal point in my academic and professional life.
* **Professor Brian Knaeble**: He provided critical feedback on the algorithm for determining user build orders. While I have not yet incorporated all his recommendations, they are slated for future implementation. His Analyzing Algorithms class transformed my problem-solving approach and sparked my interest in enhancing my mathematical skills.
* **Professors Frank Jones, JP Tang, and Peter Aldous**: Each of them imparted valuable lessons and insights that have been integral to my development.

My experiences with these educators have profoundly shaped my approach to software development. I am deeply grateful for their instruction and support, which have left an indelible mark on my educational journey.

# Organization

## Coding Standards

### Commenting

##### Functions

* Brief description of function
* Parameters
* Return Values

*Classes*

* Brief description of class

*ChatGPT*

* Highly recommend to let ChatGPT do most of the commenting
* Double check any comments by ChatGPT

### Programming Paradigm

* Object oriented

### Naming Conventions

* Classes: CapWords
* Functions: snake\_case
* Variables: snake\_case
* Constants: ALLCAPS
* Files/Folders: snake\_case

### Code Formatter

* Run all python files through black

### Type Safety

* Every function should have the parameters types listed and the return type of the function listed
* 

## Software Requirements

### Data Collection and Processing

*Sc2reader*

* Utilize python module for accessing SC2 replay data.

### Backend Development

*Python 3.12*

* Our programming language of choice

*Black*

* A python code formatter

*Pytest*

* Used for python unit testing

*SQL Alchemy*

* For storing replay data
* For storing build order data

### Frontend Development

*Flutter*

* Used for building dynamic and responsive user interfaces.

*Tkinter*

* Used for overlaying build orders to user screen

### Version Control

*GitHub*

* Used to store project, allows for collaboration and version control

### Communication

*Discord*

* Used for meetings and messaging

*Microsoft Teams*

* Used for meetings

*Message App*

* Used messaging

### Task Organization

*Trello*

* Used to track tasks and progress on the project

### Documentation

*Word*

* Used to document and organize the process of creating and maintaining the application

*Excel*

* Used to document testing plans and similar documents

*Lucid Chart*

* Used to create diagrams like UML, architecture, and schemas

*Draw.io*

* Used to create ER Diagrams

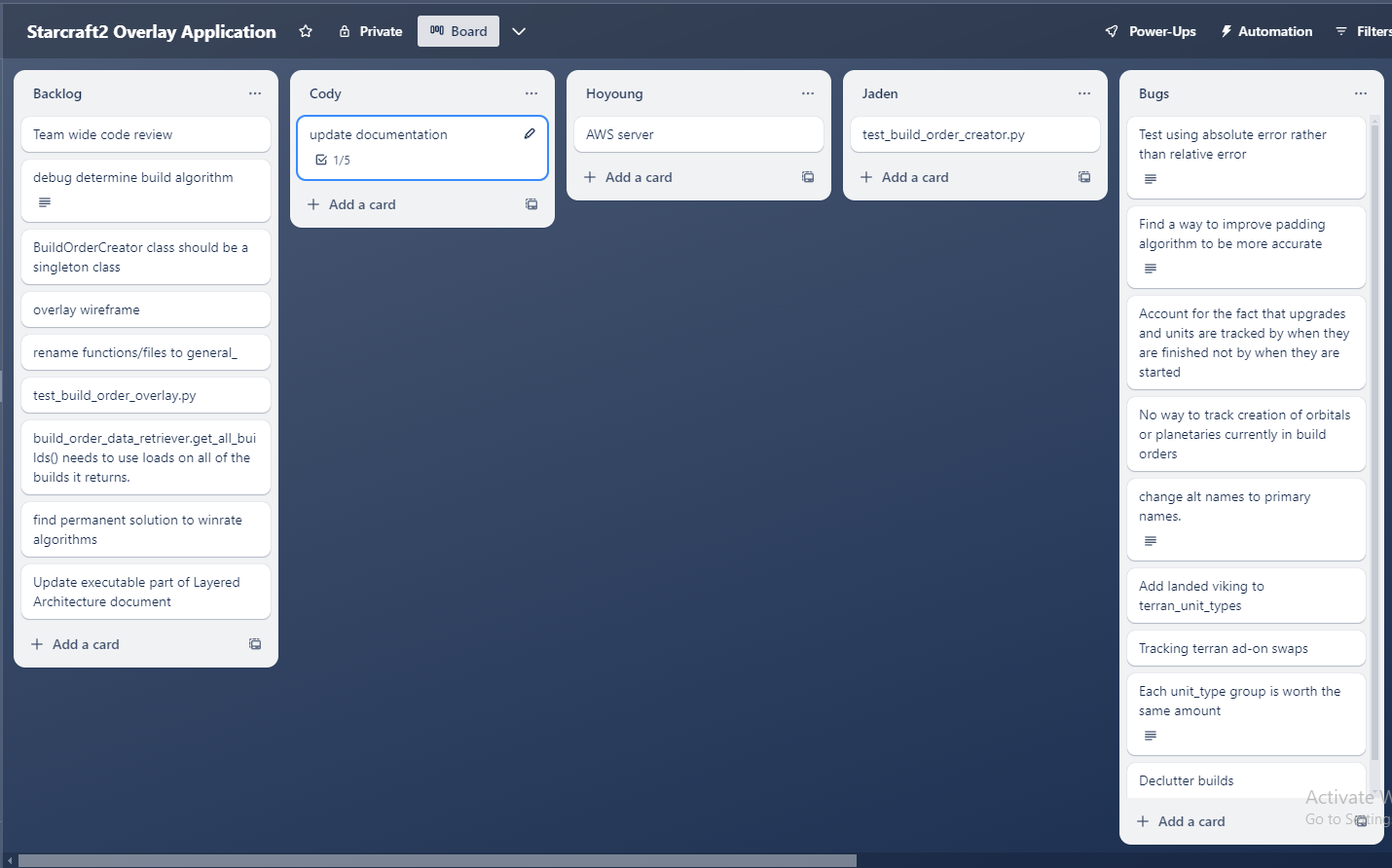
## Scheduling

### Meetings

* About every other day
* Otherwise by Appointment

## Backlog

Description: *This contains the tasks that we have completed each iteration as well as what we are currently working on, what known bugs exist, and what remains to be done. We add to the backlog as we discover new tasks but ignore them until we finish what we are working on.*



A screenshot of a chat

Description automatically generated

# Requirements Gathering

## Functional Requirements

Description: *These requirements are the general goals that our application should be able to meet. How they are met is determined later.*

* Analyze groups of replays.
  + Determine build order
  + Determine win rates based on races
  + Determine win rates of build ‘A’ vs build ‘B’
* Import and display build order to live game.

## Non-Functional Requirements

Description: *These requirements do not pertain to specific behaviors or functionalities of the application but rather to its overall attributes and characteristics.*

##### Performance and Responsiveness

* The application should be capable of processing and displaying data with minimal latency.
* It should handle high volumes of concurrent users and data requests efficiently.

##### Scalability

* The system should be scalable to accommodate a growing number of users and an increasing amount of data.

##### Reliability and Availability

* The application should have high uptime, with minimal downtime for maintenance or updates.
* It should be reliable in delivering accurate and consistent analytics data.

##### Security

* Strong measures for data security, including encryption of sensitive data and secure handling of user information.
* Implementation of proper authentication and authorization mechanisms to protect user accounts and data.

##### Maintainability and Modularity

* The codebase should be well-organized and documented for ease of maintenance and updates.

##### Usability and Accessibility

* The user interface should be intuitive and user-friendly, catering to both novice and experienced gamers.
* The application should be accessible to users with disabilities, complying with relevant accessibility standards.

##### Compliance and Legal Requirements

* Adherence to legal and regulatory requirements, such as data protection laws (e.g., GDPR, if applicable).

## Risk Analysis

### Technical Risks

##### Risk of Inaccurate Analysis

There's a risk that the program may not accurately analyze replays due to incorrect logic, outdated algorithms, or compatibility issues with different SC2 versions.

* Mitigation: Regularly update the program to align with the latest game patches, and thoroughly test the program with a variety of replays
* Contingency: Temporarily remove feature that is inaccurate until we can guarantee accuracy

##### Risk of Incompatibility with Future SC2 Updates

Future updates to SC2 might change the replay format or introduce new features not supported by the current program.

* Mitigation: Plan for regular updates and maintenance and stay informed about upcoming SC2 updates.
* Contingency: Make it so the program doesn’t accept replays past the date of the new update until the program is compatible with the new version of sc2 replays.

### Legal and Compliance Risks

##### Risk of Data Privacy Violations

If the analytics tool collects user data, it must comply with data protection regulations like GDPR or CCPA.

* Mitigation: Implement strong data privacy policies and only collect necessary data with user consent.
* Contingency: Shut down program until it complies with data protection regulations.

### Operational Risks

##### Risk of Dependency on External Libraries

The project might rely on external libraries (like sc2reader) which could become outdated or unsupported.

* Mitigation: None
* Contingency: Drop project

##### Risk of Insufficient Testing

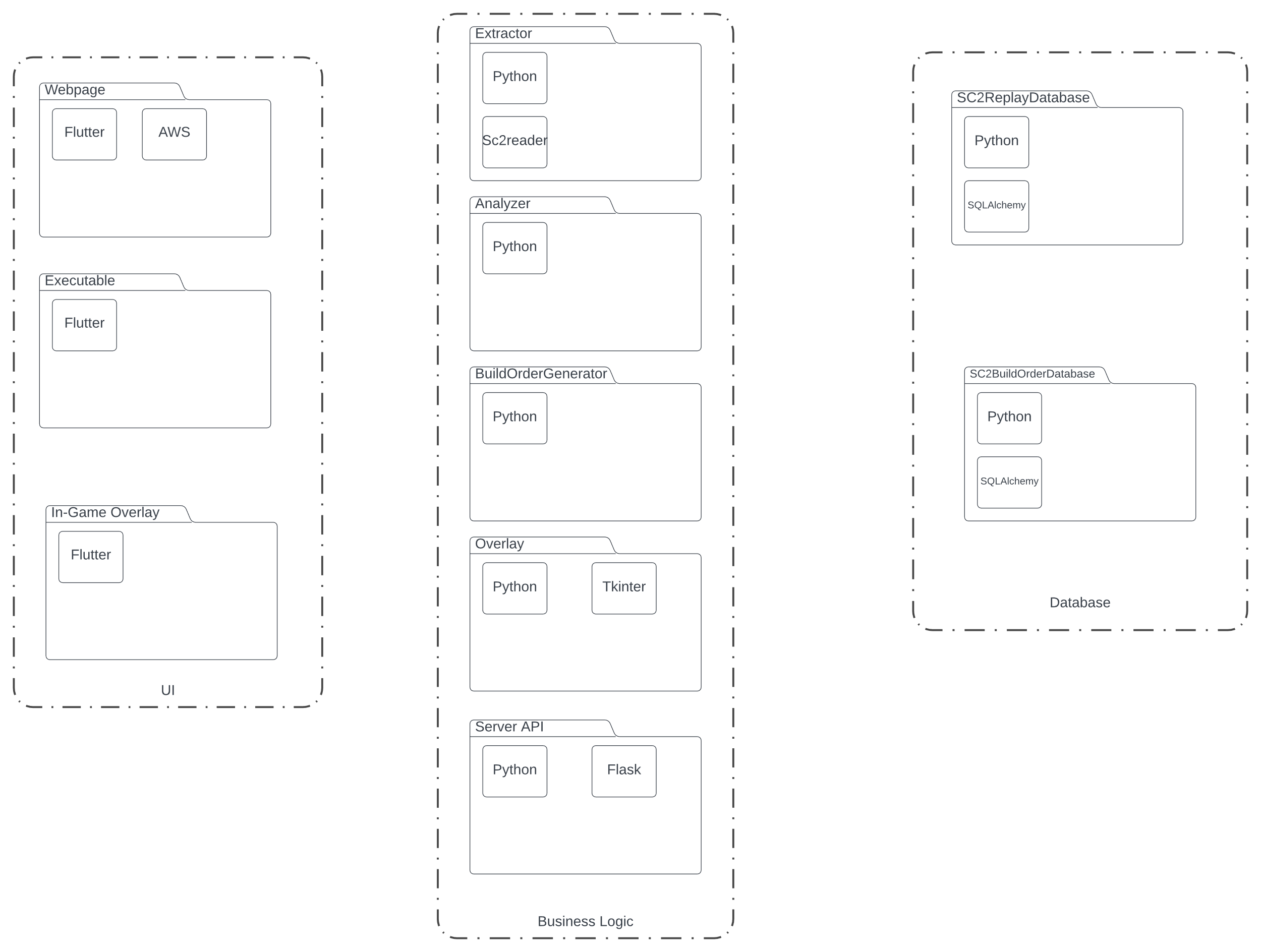
Inadequate testing can lead to undetected bugs and issues in production.

* Mitigation: Implement comprehensive testing strategies, including unit tests, integration tests, and user acceptance tests.
* Contingency: Fix bugs, possibly rollback to previous version of product and add more comprehensive testing

# High Level Design

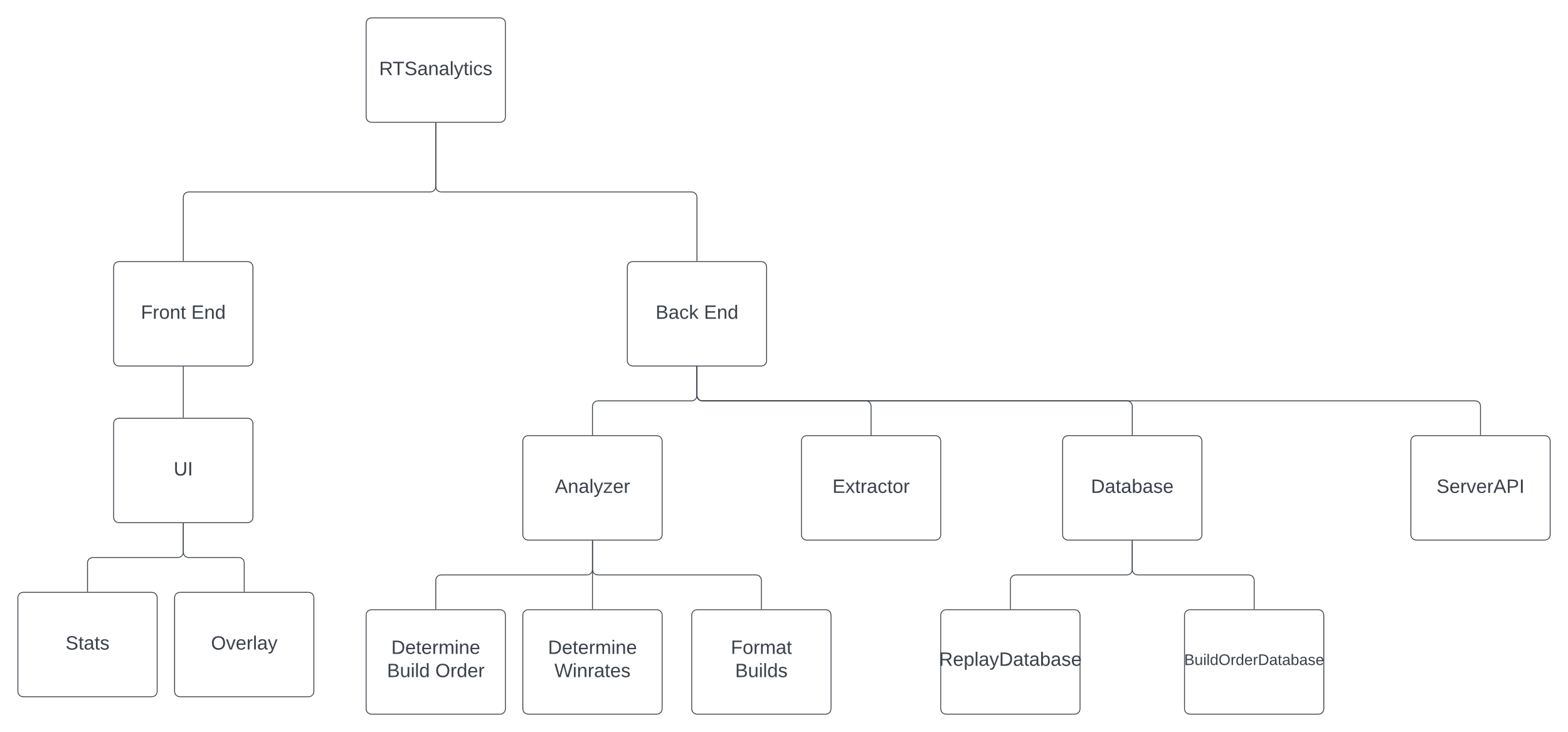
## Layered Architecture

Description: Depicts all the high-level modules, what tools are used to create them and how they can be categorized

**

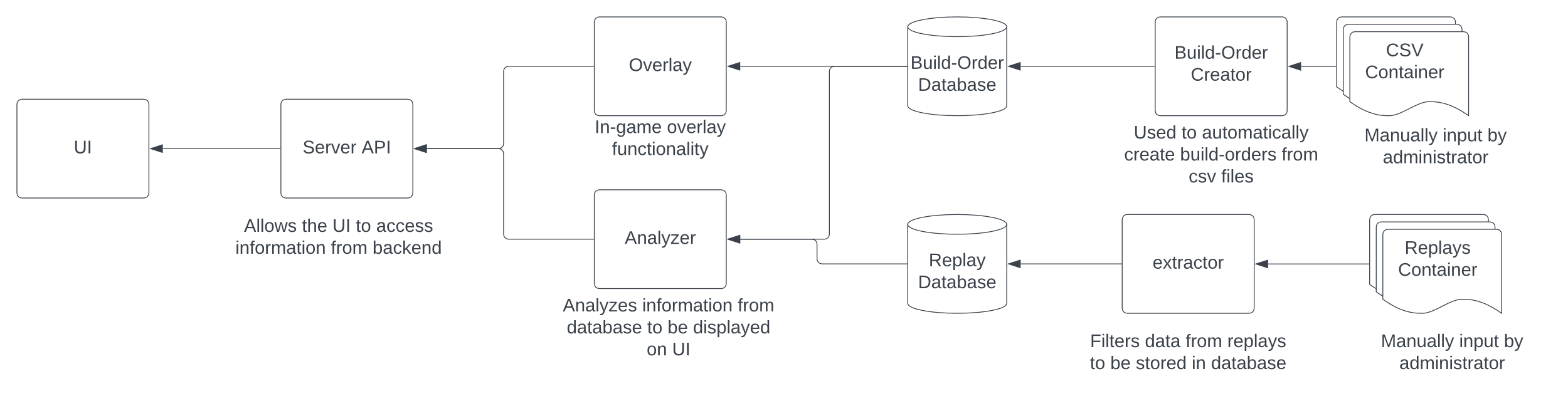
## Hierarchical Architecture

Show how all the high-level modules are split into low-level modules

**

## Data Flowchart

Description: Depicts all the high-level modules in the project and how data flows between them

**

## SC2 Replay ER Diagram

Description: Depicts all the entities in the database and their relationships to one another

*A black background with white circles and white text

Description automatically generated*

## SC2 Server Architecture

Description: this is the architecture for the server. The green is implemented and red is yet to be implemented

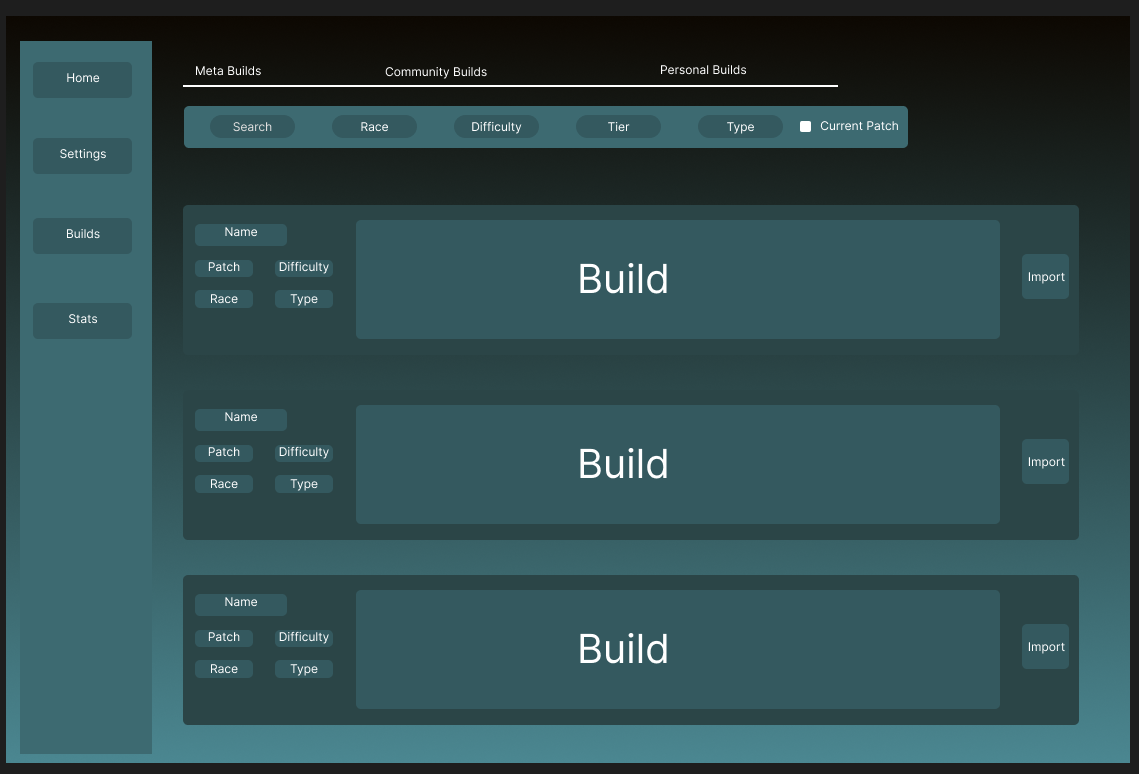
## 

## Wireframes

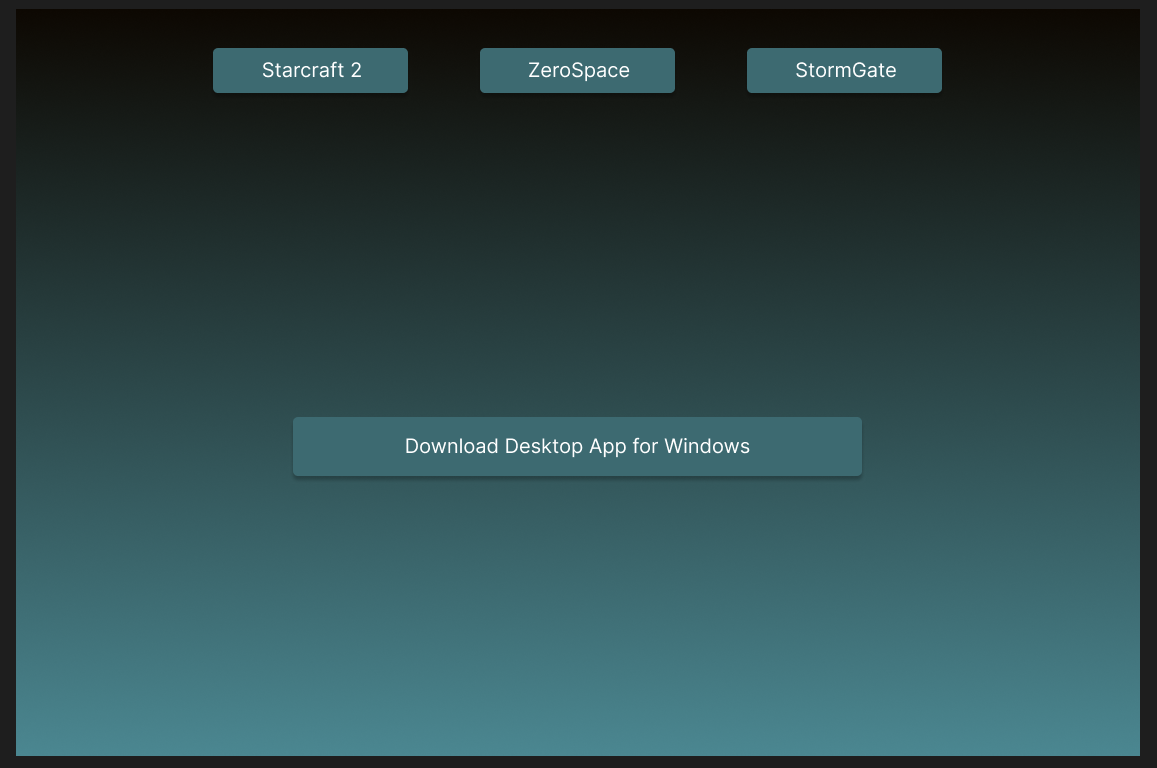
### Accounts



### Builds



### Home



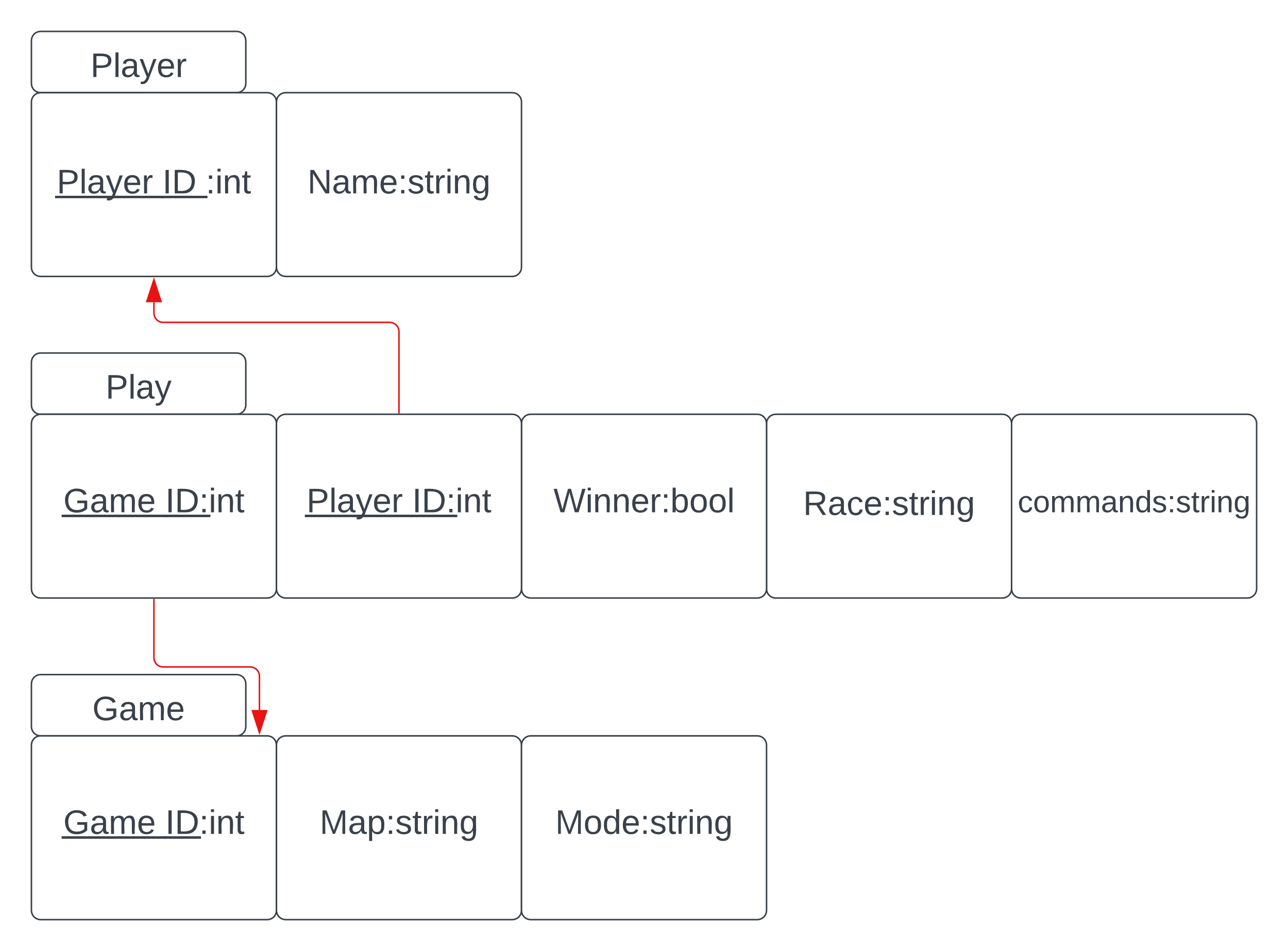
### Stats Page

### 

# Low Level Design

## SC2 Replay Schema

Description: Depicts how the entities and relationships from the SC2 Replay ER Diagram will be converted into tables for the database



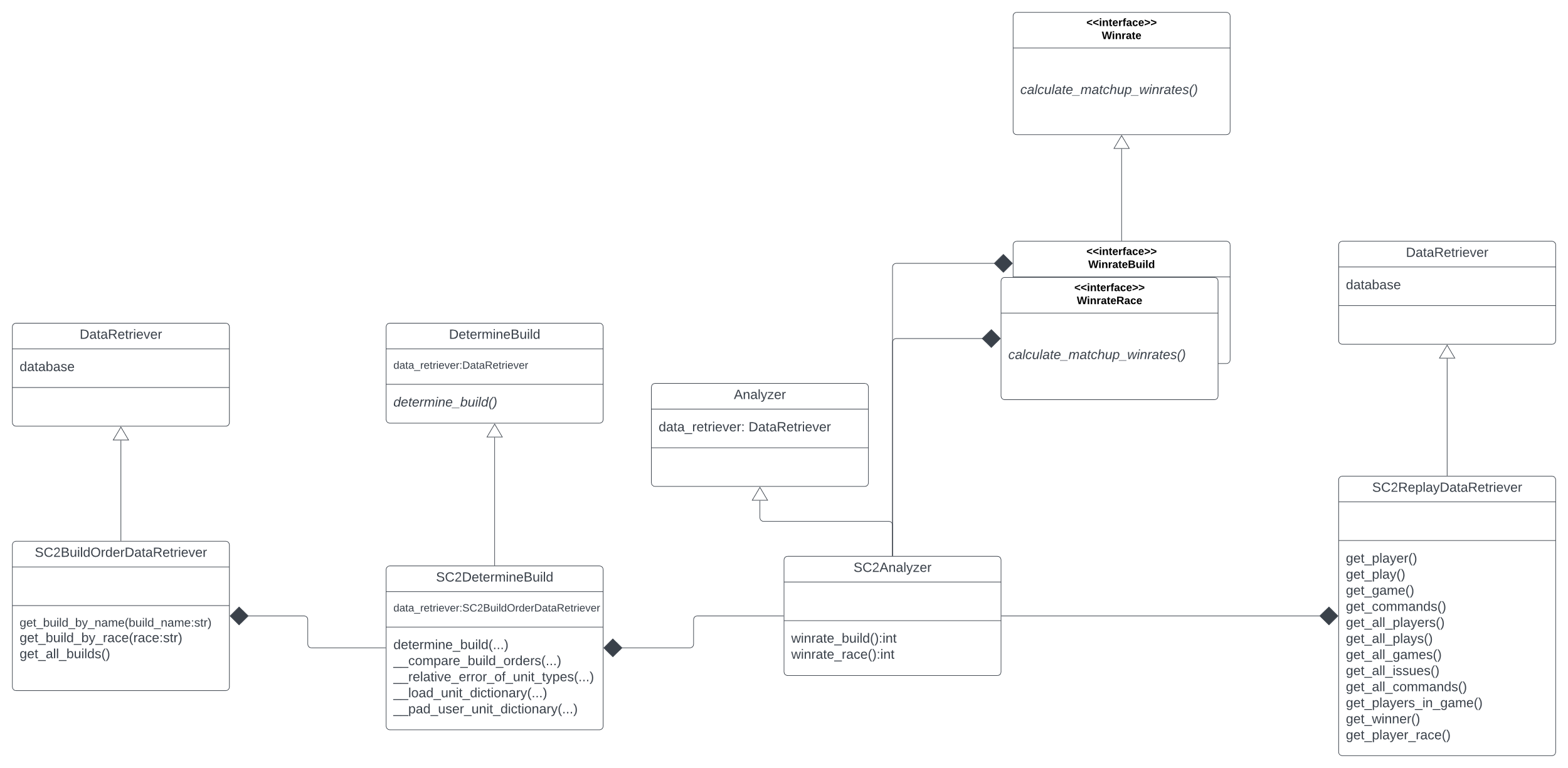
## Server API Class UML

Description: UML class diagram of the python server API portion of the project



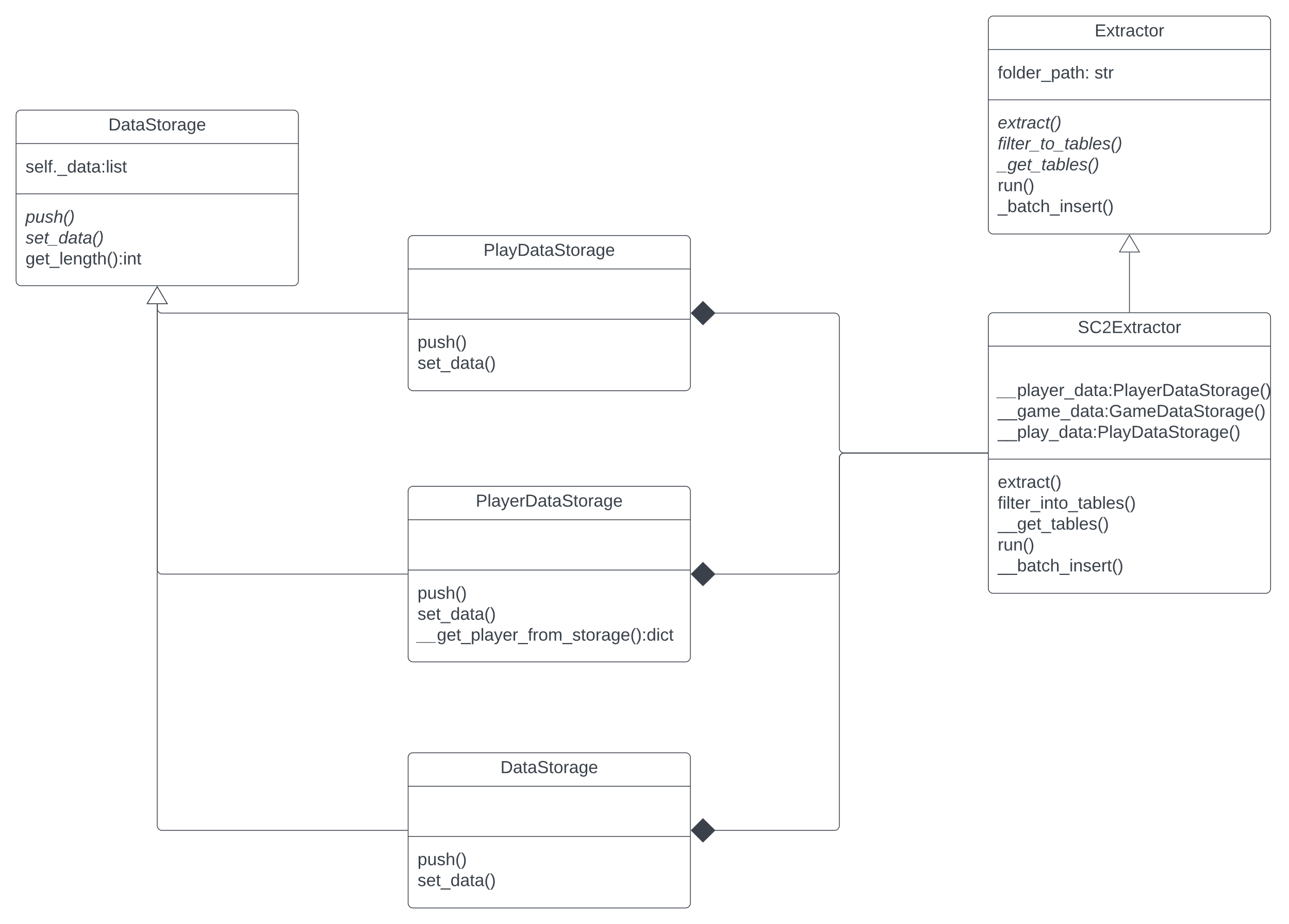
## Analyzer Class UML

Description: UML class diagram of the analyzer portion of the project



## Extractor Class UML

Description: UML class diagram of the extractor portion of the project



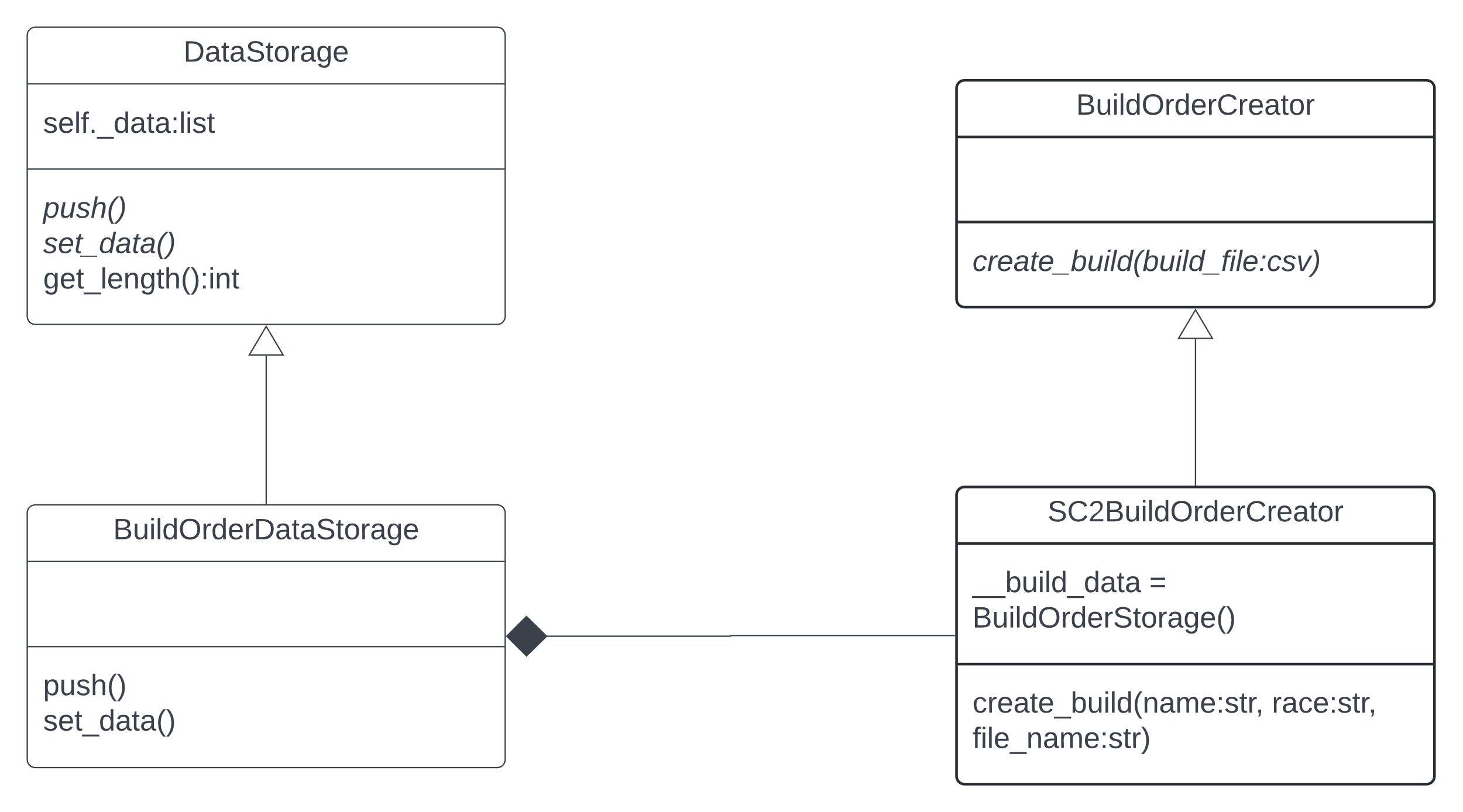
## Overlay Class UML

Description: UML class diagram of the overlay portion of the project



## Build Order Creator Class UML

Description: UML class diagram of the build order creator portion of the project



# Development

## File Structure

### Root

Description: We organized the files by their functionality at the highest level the functionalities they are split on are, configuration, analysis, database, data extraction, user interface, and server integration.

We also have folders for documentation, logging, and running examples.

A screenshot of a computer

Description automatically generated

### Data Analysis Tools

Description: Data analysis contains code for analyzing information, such as being able to determine what build order a user is doing, and win rates based on multiple factors.

It also includes code for creating and overlaying build orders, it is debatable whether that code should be under data analysis tools or under its own dedicated folder.

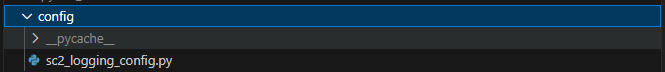
The general folder contains the parent classes and interfaces that RTS specific classes can inherit from. For now we only have SC2 specific classes.

A screenshot of a computer program

Description automatically generated

### Configuration

Description: The config file currently only contains the logging configuration for all sc2 specific logging. We will likely create RTS specific folders as we need more configuration files.



### Database Tools

Description: This contains all database related code, the data folder contains all of our database files. Each database has three file dedicated to accessing it, first is the database.py files these files are what directly create and access the .db files. However in order to guarantee that we can easily change databases when needed we have database\_access.py and database\_retriever.py files that act as a middle man for any files that want to push or get data from the database.

The general folder contains the parent classes and interfaces that RTS specific classes can inherit from. For now we only have SC2 specific classes.

A screenshot of a computer

Description automatically generated

### Examples

Description: The examples folder contains code that helps show how to run certain functions or how to implement certain python libraries. As well as containing folders of data needed for the example file to work

A screenshot of a computer

Description automatically generated

### Replay Extraction Tools

Description: This contains files that are used to pull and filter information from RTS replays. It pushes the relevant data to database tools files. Currently it is also where we store the replays that we want to get information from

The general folder contains the parent classes and interfaces that RTS specific classes can inherit from. For now, we only have SC2 specific classes.

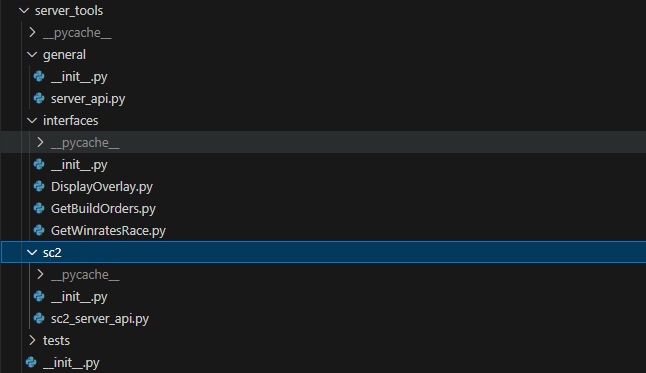
A blue line on a black background

Description automatically generated

### Server Tools

Description: This contains all the code to assist in integrating the python back-end code with the UI.

The general folder contains the parent classes and interfaces that RTS specific classes can inherit from. For now we only have SC2 specific classes.



### User Interface

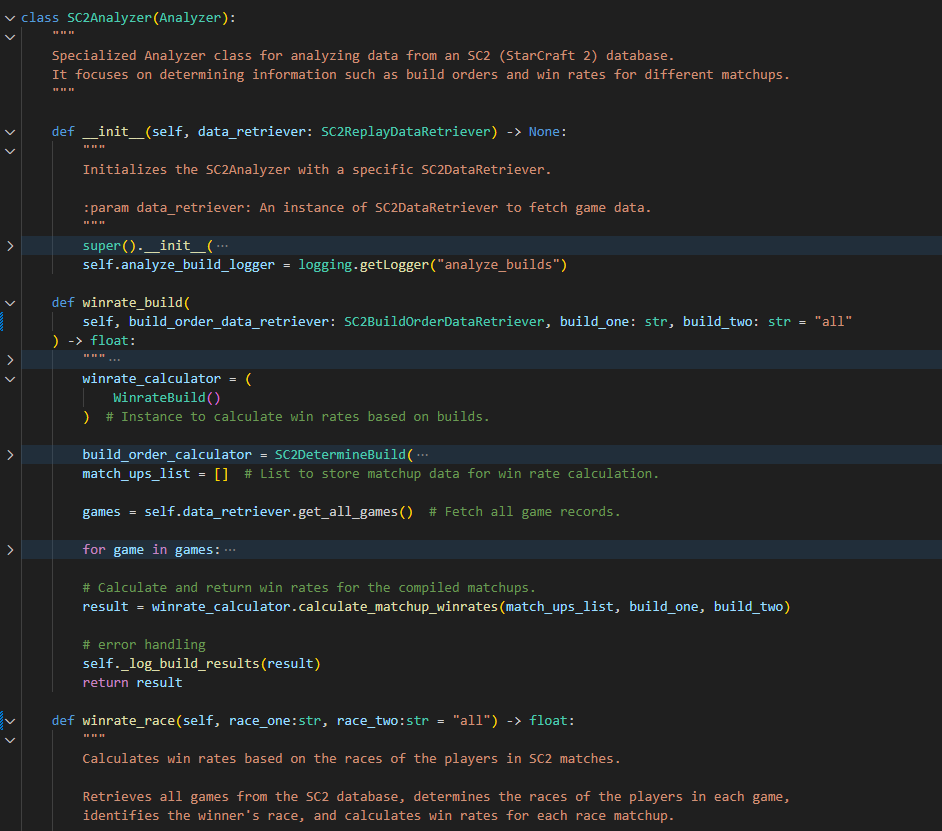
Description: This is the flutter code that builds the user interface, each of the .dart files are a page in the UI. However currently only the stats.dart is up and running.

A screenshot of a computer

Description automatically generated

## Code Snippets

### Sc2\_analyzer.py



### Sc2\_extractor.py

A screen shot of a computer program

Description automatically generated

### Sc2\_replay\_database.py

A screenshot of a computer program

Description automatically generated

### Stats.dart

A screenshot of a computer program

Description automatically generated

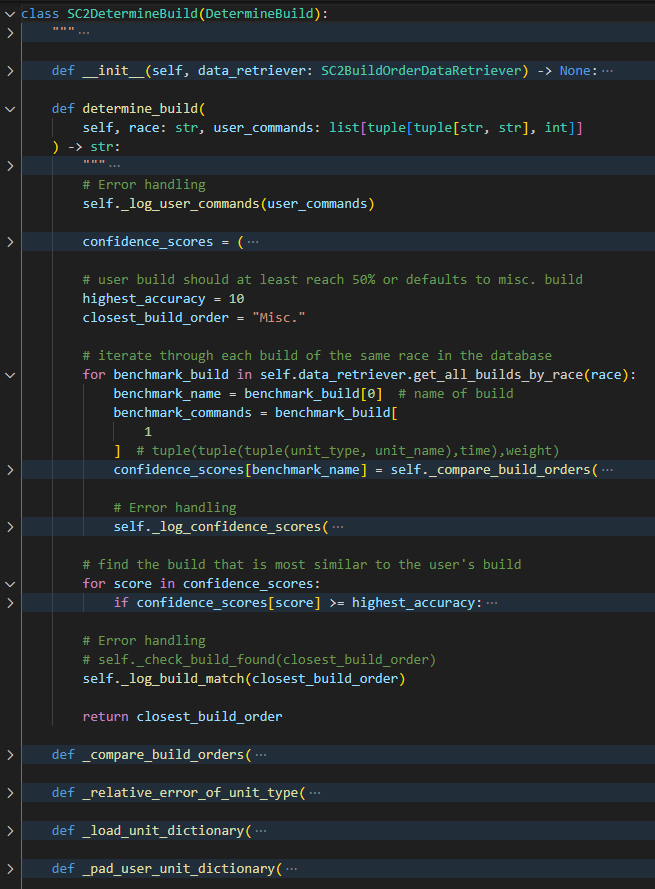
## Determine Build Orders

### Algorithm

The goal of the algorithm is to find out build order that the user is attempting to go in their game by analyzing the commands the user input in the game (note that each command consists of (TypeOfCommand, TimeCommandWasGiven). This is accomplished by:

1. Creating a database of build orders that we will call benchmark builds
2. Compare the user commands against each benchmark build and get a similarity score, this is done by
   1. Sorting all commands into containers by type, e.g. ‘scv’ commands get grouped with other ‘scv’ commands
   2. Sort each container in ascending order by when each command was given
   3. Compare how similar user’s containers are to benchmarks by
      1. Compare the size of each container
      2. If the user’s container is smaller pad it with **large** numbers until it is the same size as benchmark’s container
      3. Take the relative error between command 1’s time in benchmark to command 1’s time in user
      4. Repeat for all commands in benchmark’s container
      5. Get the average relative error of all the commands between the two containers
   4. Repeat for all containers getting the average relative error of each container
3. Get the average relative error of all the containers
4. Repeat this for each build
5. Whichever build has the smallest average relative error is considered the most similar

### Code



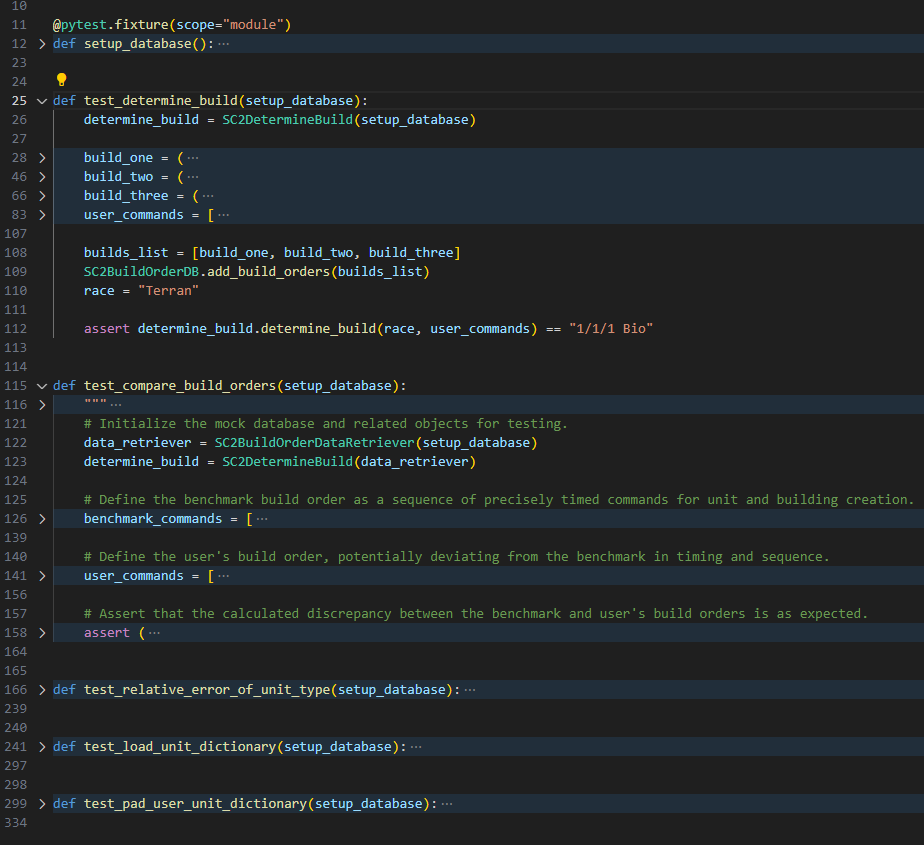
# Testing

## Pytest

Description: We created unit tests for all the methods in most of the python programming files.

Rather than a formal testing plan we just created a test file for all relevant python files

### Code



### Results

A black screen with white text

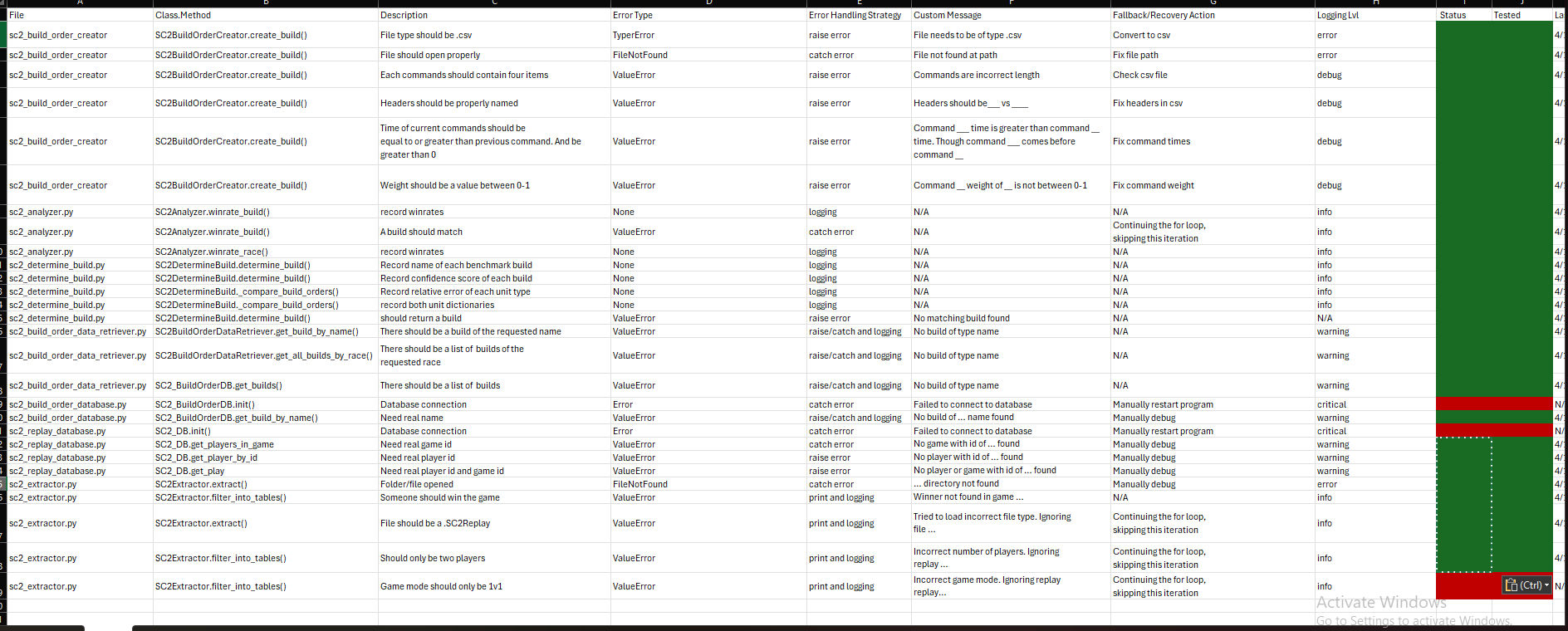
Description automatically generated

## Error Handling + Logging

Description: This is the error handling and logging plan, we went through each file searching for likely exceptions and errors that could occur. We documented where these were at later we came back at wrote error handling for the exceptions that we could. We also setup python logging so that each error will be logged, on top of this we added additional logging in certain areas to make it easier to debug in the future.

Document located at RTSanalytics\documentation\finished\error\_handling\_plan.xlsx

### Plan



### Logging Snippets

A screen shot of a computer screen

Description automatically generated

A screen shot of a computer screen

Description automatically generated

### Logging Config

A screen shot of a computer program

Description automatically generated

## Determine Build Accuracies

Description: This is used to track the accuracy of our ‘build order determining algorithm’ currently we only have anecdotal evidence for its success. Of the 10 builds we ran through the algorithm 8 were determined correctly however not a high enough variety of games were played to confidently say the algorithm is 80% accurate.

Control build refers to whether we played the build or if a random person did.

A screenshot of a computer

Description automatically generated