Architecture

Team 15

Joe Wrieden

Benji Garment

Marcin Mleczko

Kingsley Edore

Abir Rizwanullah

Sal Ahmed

Team 13

Will Dunlop

Samin Alborzi Movahhed

Nam Duong

Jakub Grzmil

Darcy Adams

Davids Kacs

Architecture v0.6

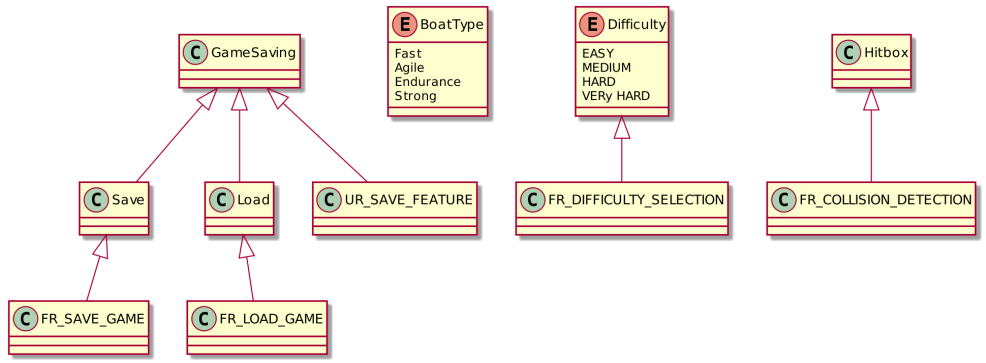
Preface

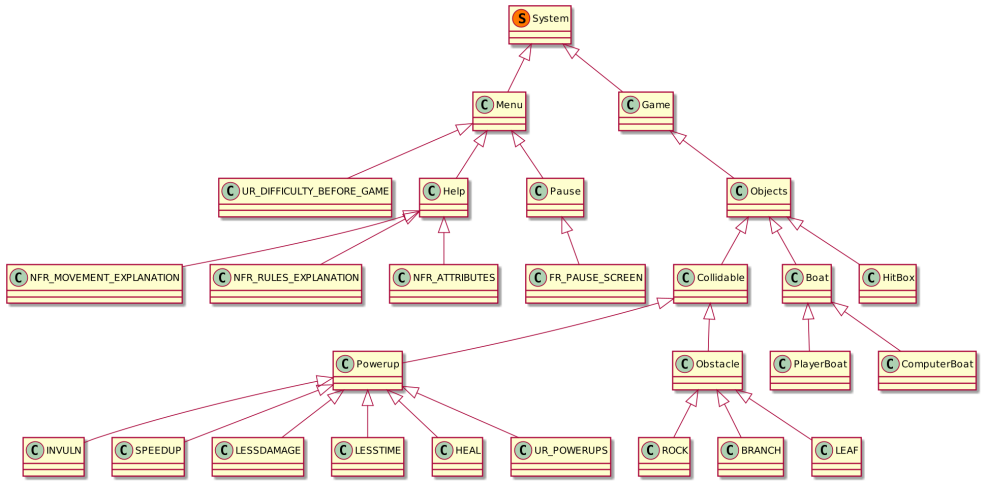
This document documents our various architectural iterations over the course of the project schedule. The document is added to as new requirements arise (refer to version history for this document's various iterations).

- v0.1 Creation of the document, and adding our abstract class diagram architectural representation.
- v0.2 Justifications for the abstract representation have been added but require revision.
- v0.3 Justifications for the abstract representation have been revised. Concrete class diagram added. Behavioural diagram may be added. Justifications for concrete diagram(s) needed.
- v0.4 Concrete diagram justifications added and revised. A behavioural state diagram may still be added and justified.
- v0.5 Last edits for final edition. Team decision to not add behavioural diagram due to time restraints for deliverables.
- v0.6 Edited version of this document, updating all the changes made to the project from Team 13

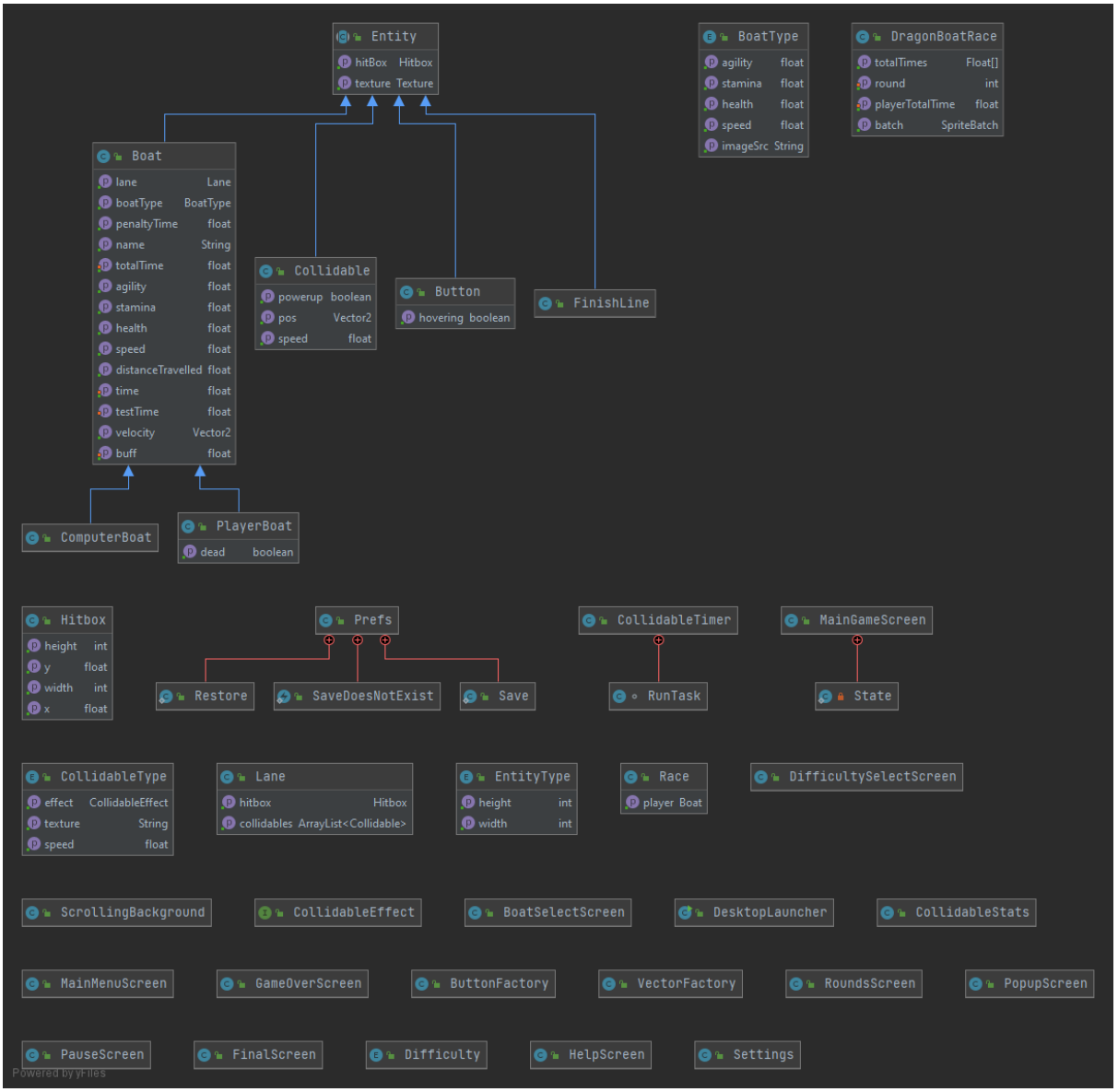
Abstract Representation of Software Architecture

Note: The diagram was split into 2 parts so it is easier to read





Concrete Representation of Software Architecture



In our team's Requirements Engineering, we used the PlantUML tool to create the UML class diagram above as a very high level, abstract representation of the game's Software Architecture. UML was our choice of modelling language, as using natural language to describe architecture can be imprecise and verbose since there are many different ways of doing the same thing, however UML graphical modelling allows people from all backgrounds, technical or nontechnical, to grasp the gist of complex concepts that code aims to carry out. As well as this, it is an industry standard and not language nor technology dependent. These are updated versions of the project's abstract and concrete architectures, changed to include new classes added to satisfy new requirements.

Justification for (original) Abstract Representation of Software Architecture

This architecture builds on the original and includes requirements provided for assessment 2. It reflects our decisions made prior to actual implementation, and serves as a basis for our lower level design. The architecture shows the classes and enums that the game will consist of, and we checked that every component in this abstract model relates back to the requirements.

This high level overview of the architecture allowed us to visualize the finished project and hence enabled us to plan when and how all of the features will be implemented. Had it been a lower level design closer to the detail of the code, making necessary design adjustments would have been costly and difficult on top of already having invested considerable time, resources and effort. Not only is it useful for bridging the communication gap between system stakeholders and software engineers, but also it aids project planning by allowing us to make decisions such as on allocating work or design problems concerning trade offs amongst potentially conflicting quality attributes before actual implementation. Thus it was advantageous to spend some time using a higher level design as we did.

Justification for Concrete Representation of Software Architecture

Further on in our Software Development Lifecycle, we developed a concrete representation of what we have planned for the Software Architecture of the game. This concrete representation is composed of a structural diagram representing the static features of the system. We checked that every system requirement relates forward to at least one component in this architectural model, in order to make sure everyone's understanding was thorough and up to standard before implementation. The components of the concrete architectural diagrams' relation to system requirements (in turn derived from user requirements, so we are making sure that we are still following through with the requirements we elicited) are justified under "Justification".

We used a class diagram form of structural modelling, as it is most applicable to the object oriented style used in our programming solution. It is clear that the class diagram builds from the abstract software architecture above and looks at the classes that are more specific to the code in more detail. The naming convention between the updated abstract architecture and the concrete architecture has changed slightly (details can be found here: https://team-13-rlc.github.io/pdfs/Impl2.pdf)1. The tool used in order to make this diagram was the UML Class Diagram tool provided by the IntelliJ IDE.

The diagram provides a critical link between the requirements engineering and the actual design of the software we implemented. It identifies the main structural components in the system and the relationships between them. By generating the concrete architecture from the abstract architecture (the precursor of which in turn was the functional and nonfunctional requirements), it is ensured that we keep to the requirements set out by the stakeholders, and as a result the concrete architecture helps reinforce these requirements into our

software implementation. A key difference in the two representations is that this representation has moved away from explicitly referring to the nonfunctional requirements, as they have been incorporated into the design, and since this one is more technical and closer to the code. Having this concrete representation of the design we have settled on will allow it to be easier for us to map the necessary components into the actual code in the organisational way we decided upon.

We tried to reuse existing paradigms and patterns as much as possible. Broadly this meant using an object oriented approach, as converting the project to any other structure would be infeasible within the time constraints. We also tried to follow patterns such as creating and extending an enum “Type” to keep track of certain complex variables. As well as closely following the Screen creation pattern.

Justification for how the concrete architecture builds from system requirements:
Note: justification for original requirements are omitted and can be found here: https://spanishforsalt.github.io/pdfs/Arch1.pdf

- Collidable: This class used to be called Obstacle and still fulfils all of those requirements. However it is now also responsible spawning powerups and hence UR_POWERUPS and FR_POWERUP_RATE (see https://team-13-rlc.github.io/pdfs/Impl2.pdf for further explanation of this change)

- CollidableType: replaced ObstacleType and also supports UR_POWERUPS and FR_POWERUP_RATE, by defining both obstacles and powerups. This enum stores the actions which the collidable take when they are collided with.

- DifficultySelectScreen: Used to allow the selection of the difficulty, UR_DIFFICULTY_BEFORE_GAME.

- Difficulty: This enum class stores the settings for each difficulty, it also landles setting all of the settings for the correct difficulty, FR_DIFFICULTY_SELECTION.

- PauseScreen: This is where FR_PAUSE_SCREEN is satisfied and where FR_SAVE_GAME (and more broadly UR_SAVE_FEATURE) can begin.

- Prefs: This class houses the Save and Restore subclasses used for FR_SAVE_GAME and FR_LOAD_GAME respectively.

Bibliography

- “Software Engineering”, Ian Sommerville, Chapter 6
- ”UML Online Training”, Tutorials Point
- “Use Case Models and State Models”, Binary Terms
- Software Architecture: Foundations, Theory, and Practice, R.N. Taylor, N. Medvidovic, and E.M. Dashofy John Wiley & Sons, 2008.
- Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. Documenting Software Architectures: Views and Beyond. Addison- Wesley, Boston, 2002.