Architecture

Team 2 - Billy’s Amazing Team

Member Names: Mitchel Bekink, Alyx Bruno-Bamford, Ashley Bryan, Rajul Chawla, Bosco Lau, Kacey Van Der Walt, Chris Grzywacz

Languages and Tools:

For almost every architectural diagram, we used Unified Modelling Language (UML) through the coding language PlantUML. UML is the industry standard diagrammatic language, and using the text-based PlantUML language allowed us to transform lines of code into fully drawn diagrams far quicker than if we had drawn the diagrams ourselves. The 'sample' feature also gave us examples of code, which was useful for figuring out how to represent features in our diagrams. Also, as the architecture went through many iterations: having the ability to update the diagrams in almost an instant, rather than having to redraw them, was extremely valuable. We used the website https://www.planttext.com/ to execute our PlantUML code, generating the final diagrams. We stored both a PDF and text files containing the code for each diagram. The only diagram not using UML was our problem decomposition diagram, which was designed using the Figma software. This allowed us to collaboratively arrange the diagram based on the requirements. Finally, we used the Whiteboard functionality in Zoom meetings to draft our initial UML diagrams, like Figma, this let us arrange the diagrams both collaboratively and without wasting physical resources such as paper.

Design Process:

To design the architecture, we followed the Responsibility-Driven Design (RDD) approach. This allowed us to manage what would normally be a large, complex problem by breaking it down into clear, iterative steps. We first took our user requirements tables and used them to generate some candidate objects for our project. By using an Object-Oriented approach to the project, architectural diagrams can be generated much more easily, and the project as a whole will be easier to both code and understand.

We then took these candidate objects and expanded upon them using the whiteboard functionality on Zoom. Each candidate object was written up as a post-it note, allowing us to freely move and edit each object in a fully collaborative environment. With these objects fully fleshed out, we grouped related objects and discussed whether each object should remain as a unique object, or should be combined with others. After we had eliminated unnecessary objects, we identified each one's responsibilities and connections to other objects. In other words, how the objects work together to enable our game to function. With these steps, our first iteration was finished.

We iterated through the above steps multiple times as a team, each one refining and improving upon our previous architecture. Once we were complete, we had a final diagram that would represent how our game's architecture would be laid out. The main iterations are linked below, as well as our final RDD diagram:

- Link to iteration 1: 3.1 RDD Diagram 1
- Link to iteration 2: 3.2 RDD Diagram 2
- Link to Final Architecture: 3.3 RDD Diagram 3

Once our RDD process had concluded, we took the above architecture and began to translate this into the appropriate UML diagrams. We made sure to cover both structural diagrams and behavioural diagrams, to ensure complete transparency between the implementation and the expected outcomes of our system.

UML Diagrams:

The first diagram we made was the problem decomposition diagram. We decided this would be useful to sort the user requirements and begin to plan the structure of the game. We colour-coded the nodes according to the priority of their corresponding URs (purple for shall, yellow for should, green for may). The bullet points linked below correspond to the nodes, and the dashed ones are notes on the corresponding nodes. These notes are directly related to the problem decomposition diagram, and provide extra information or clarification.

Link to problem decomposition points: 3.4 Problem decomposition points
Link to problem decomposition diagram: 3.5 Problem decomposition diagram

After researching various UML diagram types, we decided to draw 4 separate diagrams: two behavioural and two structural. This gave us a wide range of different diagrams to help us view different aspects of the system as a whole, rather than focusing on just one view. The four diagrams we decided upon were class and object for structural, with sequence and state for behavioural.

As a team, we sketched an unofficial structure diagram to understand what classes and relationships we would use. This allowed us to discuss the implementation as a group. Using this, we made a UML structural class diagram. We used this to map out the relationships between classes, including inheritance, reliance, and usage between them. We also planned some preliminary methods and variables.

Link to the initial class diagram: 3.6 Initial class diagram

We then moved on to creating the UML structural object diagram. This diagram acts as an extension of the class diagram. While the class diagram showed the overall structure, layout and relationships between the various classes in our system, the object diagram gives us an insight into how these objects look after they have been instantiated. This provided us with a snapshot of the inner workings at runtime. Both of these structural diagrams work hand-in-hand with one another, to provide the implementation team with the groundwork to build off of, as well as provide traceability to the whole project.

Link to the object diagram: 3.7 Object diagram

The sequence diagram was the first behavioural UML diagram we made. This diagram is in contrast to our previous two as it focuses on the user and how they will interact with the system. It shows the relationship between the actions performed by the user and the reactions the system has, which will help us plan the implementation in a more abstract way.

Links to the sequence diagrams: 3.8 Implemented version and 3.9 Full version

The final diagram we made was a UML behavioural state diagram. We used it to model the behaviours of classes in response to changing conditions and events. This again, extended on the structural diagrams by showing how the system would respond in different scenarios, allowing greater transparency and traceability within our final implementation.

Links to the state diagrams: 3.10 Implemented version and 3.11 Full version

Linking the Architecture to the Requirements

As previously mentioned, our architectural creation process was directly driven by the requirements generated from our customer meeting. Because of this, it is easy to see which objects on our diagrams relate to which requirement, and some of these links are shown below:

ID	User Requirements	Architectural Link
FR_MAP	UR_PLACE_BUILDING, UR_MAP	World class
FR_PLOTS	UR_PLACE_BUILDING, UR_MAP, UR_LAND_TRAITS	A list of abstract buildings in the World class
FR_BUILDINGS	UR_PLACE_BUILDING	addBuilding method in World class
FR_TIMER	UR_FIVE_MINUTES	Timer class
FR_PAUSE_GAME	UR_FIVE_MINUTES, UR PAUSE_GAME	pause method in the Timer class
FR_PAUSE_TIMEFLOW	UR_PAUSE_TIME, UR_FIVE_MINUTES	pause method in the Timer class
FR_MONEY	UR_MONEY	cost in the BuildingType class
FR_EVENTS	UR_EVENTS, UR_DIFFICULTY	Events Interface
FR_WARNING	UR_NO_NEGATIVE_FEEDBACK, UR_SATISFACTION	setWarningMessage in the UI class
FR_TITLE_SCREEN	UR_TITLE_SCREEN	Title Screen class
FR_CONSTRUCTION_TIMER	UR_CONSTRUCTION_TIMER	timeRemaining method in IncompleteBuilding class

Not all of our requirements are listed in the above table, and this is for two reasons. The first is that additional requirements were added to our table throughout the project, so our initial architecture diagrams do not reflect these additions, however, the final ones do. The second reason is that not all requirements can be given an explicit method or class, as some are implemented through various classes/methods that work together, however, none directly relate to the requirement.

On top of the links listed above between the requirements and the architecture, we wanted to ensure code traceability and ease of collaboration. Class, method and attribute names within both the architecture and implementation were consistent with the wording used in the user requirements. For example, our requirements list both a timer and pause functionality, and in the architecture these are represented in a class called Timer and a method called pause. These implicit links aid all team members, by increasing implementation and architectural traceability.

Justification of the Final Architecture

Our final architecture is a culmination of different programming and structural techniques we have chosen to produce an optimal solution. The architectural decisions can be broken down into three main choices: using an Object-Oriented Approach, using JSON files to store data and using abstract classes.

The primary reason for choosing an Object-Oriented approach was that our assigned programming language (Java) was designed with OOP structures in mind. It allows us to create different classes, packages and methods with ease and massively reduces the need for code duplication. This saves a lot of time, which was our most important resource during this project, so choosing a programming style that would help us reduce time wasted was essential.

We also made use of a somewhat data-driven architecture for assets and game content, by describing things such as building types in a single class, and then creating different instances using parameters loaded from JSON files in the game's assets. This allows for easy addition of new content, as most objects do not need variation in their code behaviour, just values used in calculations and display, such as names of textures and areas they can be placed on. This architecture also supports dependencies between objects, such as world layouts containing references to features such as lakes and roads, which helps reduce duplication of parameters, and allocates unique identifier strings to each object to allow them to be easily referenced.

The final architectural choice made was to use abstract classes in our architecture. These cannot be instantiated on their own, however, they provide the blueprints on how other subclasses should be written, and what information they must include. One big benefit of this is again reducing code duplication. Child classes inherit all code and methods from a parent class, so we can produce many subclasses with minimal rewriting of code. For example, we will use abstract classes in the creation of buildings, so that different building types can be added faster. Another benefit of abstract classes is that they make our code more readable, which is key in software development, as many different people need to understand our implementation.

Overall our architecture was created from the above decisions, as well as the RDD process we followed. The broad decisions gave us a direction to follow, and ensuring that our final architecture was as efficient, readable and logical as possible was down to the RDD steps. These steps ensured that we removed duplicate objects, links between objects were logical, each object had a specific purpose; was not redundant; and the architecture we created was linked to both our client’s needs and the user's requirements.

Evolution of our Architecture

While we initially designed multiple iterations of initial architectures, none of these iterations matched our final architecture by the end of the project. The final RDD-created architecture was used to commence our implementation, but as our coding team worked, they noticed ways that our design could be improved.

When changes were needed, the coding team brought these proposed changes to a group meeting and we discussed the proposed changes. One main example of this was the use of JSON files. Some members of our team thought that these would be a good addition, while others thought against it. After a lengthy discussion, we agreed that the pros outweighed the cons, and our architecture was updated accordingly.

These dynamic changes occurred throughout the allotted implementation timeframe, and by the end of the implementation, a few changes had been made. The underlying architecture was still the same, but there were a few small tweaks and alterations to a better final product. This is why we ensured we kept the architecture dynamic, we anticipated that there may be changes needed, so we left our diagrams easy to edit and allowed time at group meetings for changes to be discussed, and we feel that these decisions were massively beneficial to the outcome.

Link to final class diagram: 3.12 Final class diagram

Bibliography

[1] Figma, “Figma,” Figma, 2024. [Online]. Available: https://www.figma.com/.

[2] PlantText, “PlantText,” PlantText, 2024. [Online]. Available: https://www.planttext.com/.

[3] PlantUML, “PlantUML at a Glance,” PlantUML, 2024. [Online]. Available: https://plantuml.com/.

[4] tutorialspoint, “UML - Quick Guide,” tutorialspoint, 2024. [Online]. Available: https://www.tutorialspoint.com/uml/uml_quick_guide.htm.