Architecture Diagrams to Make / Architecture Process

- Identify initial components:
- Identify actors who perform activities with the application and the actions those actors may perform
- Discover typical users of the system and what kinds of things they might do with the system
- **Use UML for component diagrams**
- Assign requirements to components
- Analyse roles and responsibilities
- Analyse architecture characteristics
- Restructure components

UML Diagrams:

- Component diagrams for high-level decomposition
- Class diagrams to capture core entities, their features and relationships (no constructors, getters or setters, no dependencies on implementation language / library classes, visibilities, composition / aggregation relationships not important at this stage)
- Sequence / activity diagrams for object communication
- State diagrams to capture different system states
- Component-entity-system diagrams
- OOP entity hierarchy diagrams - these are just class diagrams!! (Sophia will do this)

Sally first half, Sophia second half

Classes

Info holder

UI

- Main
- Title
- End
- Menu
- Instructions
- Buildings

Buildings

Contains the blueprint for all of the buildings in the game. Including attribute such as satisfaction, cost, grid cords, sprite, num placed

Info holder

Screen

Has background image, various UI elements, and various methods to render itself, resize itself etc

User Interface

Designer story

We are developing a video game involving designing a university campus to maximise student satisfaction, to be run on a desktop. The game will last 5 minutes, which will represent several years in the game. The player will place buildings on placeable tiles from a selection of categories (e.g., accommodation, recreational, food, and education) around the campus. The placement and type of building will affect the student satisfaction score. They will also have to contend with events which can affect satisfaction throughout the game. At the end of the game, the player will receive their student satisfaction score.

Themes/candidate objects

- Grid considering placeable and non placeable tiles + building location
- Timer (for out of game and in game time)
- Score counter
- Screen / interface
- Buildings - counter etc
- Events
- Building placing phase?
- Camera
- Sound
- Screen
- Mouse
- Keyboard?
- Where building is being placed

Components

- User
- Buildings
- StudentSatisfaction
- Events
- UI
    - Pause menu
    - Building menu
    - Game screen
- Timer

Actual classes in implementation:

- DragAndDropManager
- GameOverScreen
- InstructionScreen
- MainGameScreen
- MainMenuScreen
- Money
- MoneyClass
- Reputation
- ReputationClass
- Settings
- TimerClass

Beginning of design process section of write-up: (will need to change some stuff in later part based on updated diagrams)

We began the architecture design process by looking at the requirements for the game and identifying initial components that may be needed. To do this, we used an actor / actions approach. We decided to use this approach as we thought it fit an object-oriented programming style the best, as opposed to event storming, which is inappropriate as the system will not only be built on messages or events to communicate between components, or a workflow approach, which is more suited to something less open-ended than a game.

The actor / actions approach we took involved identifying actors and the actions they may take, as well as potential users and the way they might use the system. An example of an actor in our system would be a camera, which can move up, down, right and left, showing a different view of the map depending on its position. We established the user of the game to be a player, who will use the game by playing it. In addition to this, they may also want to pause the game, or read the instructions, for example.

Through this process of considering the actors and users of the system, we determined some initial components. One of these would be the user - a key component of the system. As well as this, there are different types of buildings that would each be a component. We included these buildings as requirements UR_BLD and UR_TYP specify that the user must be able to place buildings on a map, and that there will be a selection of different types of buildings. Similarly, UR_TIM, FUN_CNT, FUN_TIM and FUN_SCR describe the need for timers for both in-game and out-of-game time, and some sort of score. Therefore, we added a Timer component to handle both timers, and a StudentSatisfaction component for the score, which is based on student satisfaction, to our plan.

We also considered the requirements related to UI elements, such as UR_PAU, FUN_BLD_BTN, FUN_BLD_TAB, FUN_PAU_BTN, FUN_PAU_MEN, and more, and decided to include various screens as components. Some of the screens we identified as potential components include a pause screen, a building screen and a game screen.

Although it is not a requirement of part one of the assessment, events will have to be implemented in the final game, so we also included an Events component in our plan to prepare for this.

Moreover, we made sure to incorporate responsibility-driven design in our design approach, since the implementation of the game uses object-oriented programming principles. As the first step for this, we had a requirements meeting with the stakeholder, from which we came up with a list of requirements for the system. We also listened to how the stakeholder suggested users might use the system, and developed a holistic view of various use cases from this. Considering these requirements and user stories, we created a designer story, which summarises what we learned and suggests a typical use case for the system. This can be read on our website.

Next, we thought of various possible themes and candidate objects for the implementation of the game. In this process, we were influenced by our consideration of the actors and users of the system, in addition to our designer story and requirements. Examples of candidate objects we identified include a grid with placeable and non-placeable tiles, holding the

locations of buildings (as required by UR_BLD and NFR_OBS), timers for in-game and out-of-game time (FUN_TIM and UR_TIM, respectively), a score counter (FUN_SCR), screens (FUN_STA, FUN_PAU_MEN, FUN_END_SCR), types of buildings (UR_BLD, UR_TYP), events, a camera (UR_CAM, FUN_CAM), sound (UR_SND, FUN_SND_EFF, FUN_SND_MUS), a mouse and a keyboard.

As we iterated on these ideas, we removed sound, a mouse and a keyboard from our list of candidate objects. This was not because we no longer needed sound, or mouse and keyboard inputs, but because we felt that it was not necessary for entire objects to be dedicated to these functionalities. Rather, they could be part of other objects, or we could make use of functionality provided by pre-written libraries to implement them.

Following this, we used CRC cards for each of these concepts, describing their purpose and picking a role stereotype from information holder, structurer, coordinator, controller, interfacer or service provider. The CRC cards we created can be viewed on our website. We proceeded to group these cards into groups that may be able to inherit from the same class, such as MainGameScreen, MainMenuScreen, BuildingScreen and EndingScreen, which would all be able to inherit from a common Screen class. We also grouped some similar classes, such as Timer and ScoreCounter.

We then proceeded to add responsibilities to each CRC card. These responsibilities include information that the objects need to know, and actions they need to be able to perform, eventually becoming the attributes and methods that a class will have. We also added collaborators to each CRC card, trying to imagine what other objects they would need to interact with in our system.

For example, the ScoreCounter object should know the current student satisfaction score, in line with requirement FUN_SCR. In addition to this, requirement FUN_CAM dictates that the camera should be moveable using the arrow keys, so the player can have a different view of the map. Thus, we added the responsibility to the Camera object that it should be able to update its position based on player input.

Subsequently, we were able to turn these CRC cards into actual designs for classes, and construct various UML diagrams to represent the architecture of the system. The final versions of these diagrams can be viewed below, whilst previous iterations are available on our website.

Further along in the project, the actual architecture implemented changed somewhat from our original design. We first created UML diagrams from our original design, then iterated upon these when we realised another design would be better suited to the game during the implementation stage. The UML diagrams below consequently differ from the CRC cards we created and described above, and rather reflect the final implementation more closely. To reiterate, versions of the UML diagrams that more directly match the CRC cards and what we designed originally are available on the website.

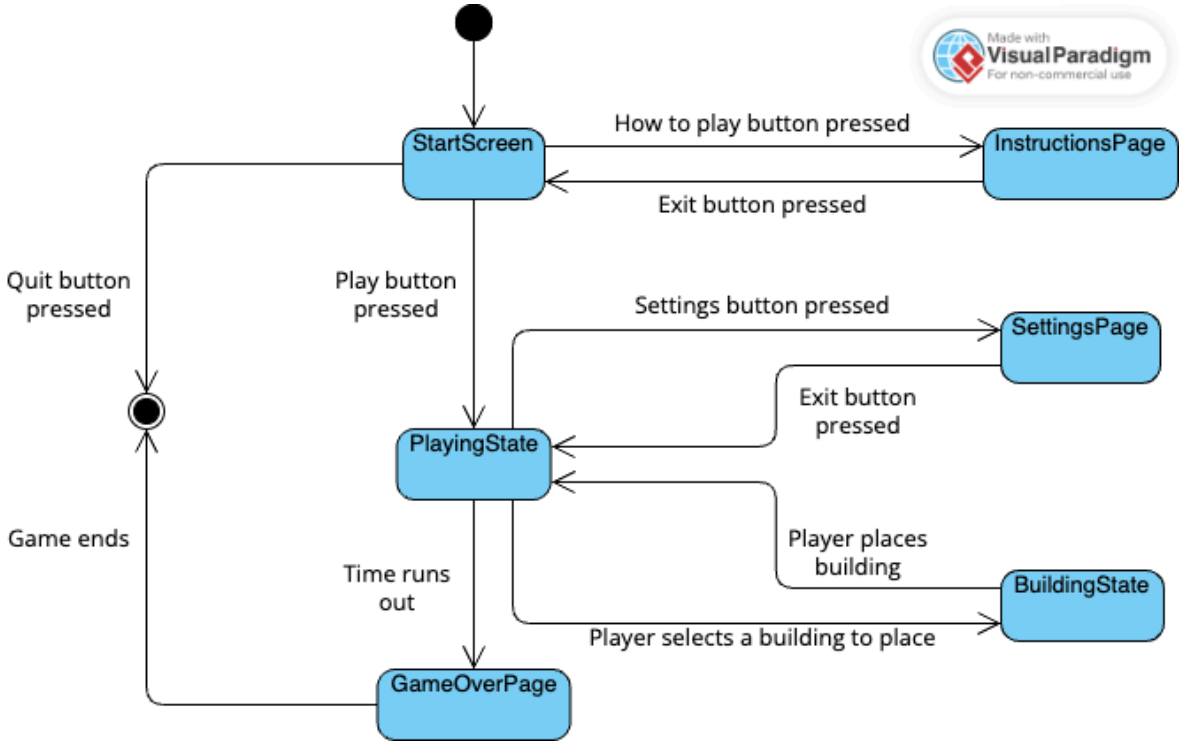
Individual Diagram Write-Up Outline:

State Diagram:

We created a UML state diagram to represent the different states that the system could be in at any given time, and how the system will transition from one state to another.

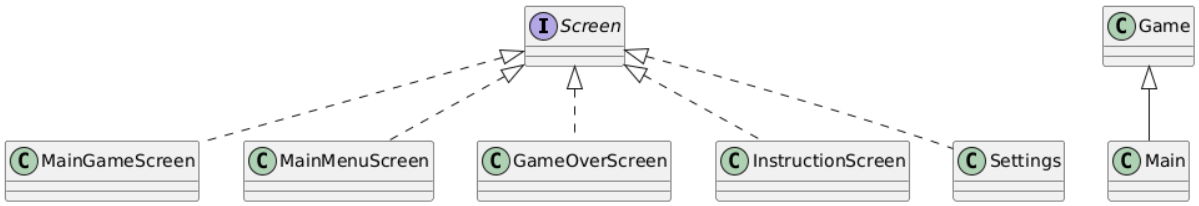
For this diagram, we used a tool called Visual Paradigm Online, which allows you to drag and drop boxes for states and circles for the starting point and ending point. As well as this, you are able to draw arrows between states and place text boxes.

We had originally used PlantUML on the website PlantText to create the state diagram, but the arrows and states overlapped so much that it was unreadable. We therefore chose to use a tool that allows you to drag and drop elements of the diagram instead, allowing us to choose where to place each state and the arrows between them.



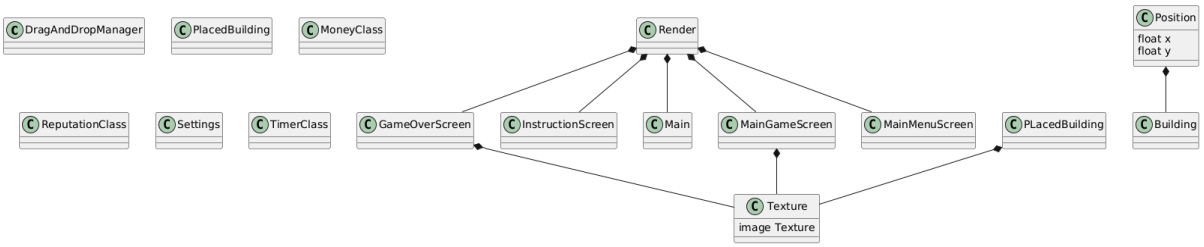
OOP Entity Hierarchy:

We included a diagram to represent the object-oriented programming related hierarchies in our system. We represented this through a UML class diagram, using PlantUML on the website PlantText. Some of the commands we used to create this diagram are, for example, “interface Screen,” to make a box for the interface Screen, and, “Screen <|.. MainGameScreen,” to draw an implementation relation arrow pointing from MainGameScreen to Screen. This shows that the class MainGameScreen will implement the interface Screen.

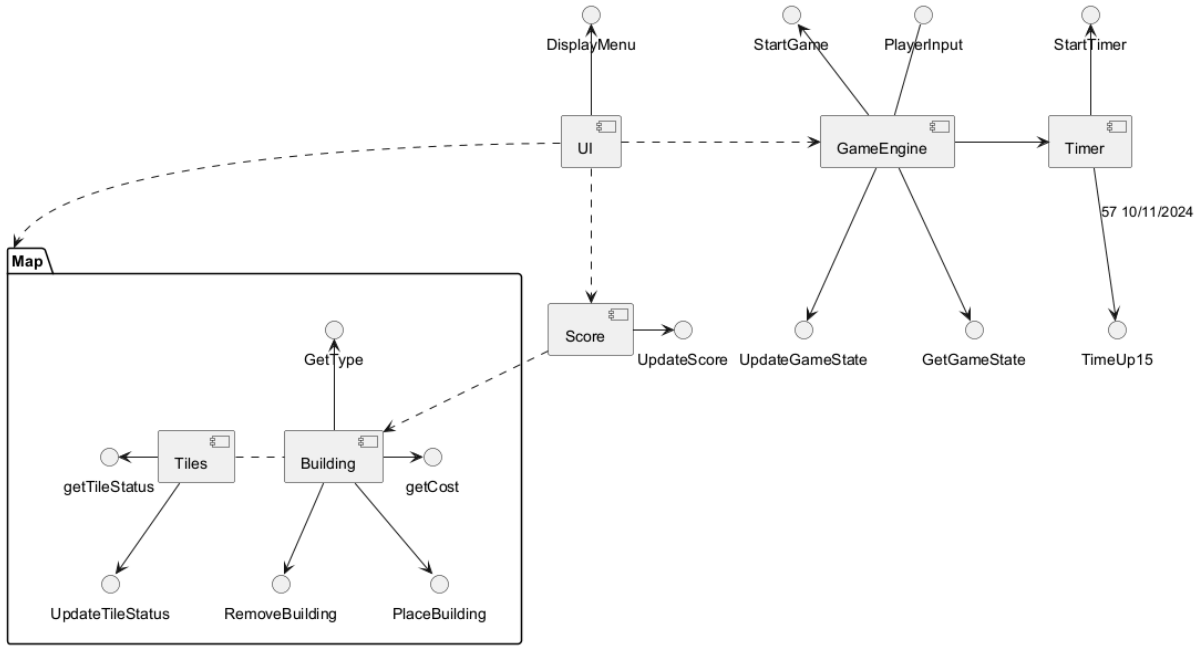


Component-Entity-System:

We also used a UML class diagram to show the entity-component-systems style in our game. We again used PlantText, for PlantUML. One PlantUML command we used specifically for this diagram includes, “Position : float x,” to specify that the component Position has a float attribute named ‘x’. Furthermore, we used commands such as, “MainGameScreen *-- Texture,” to draw a composition relation arrow from the component Size to the entity Screen.



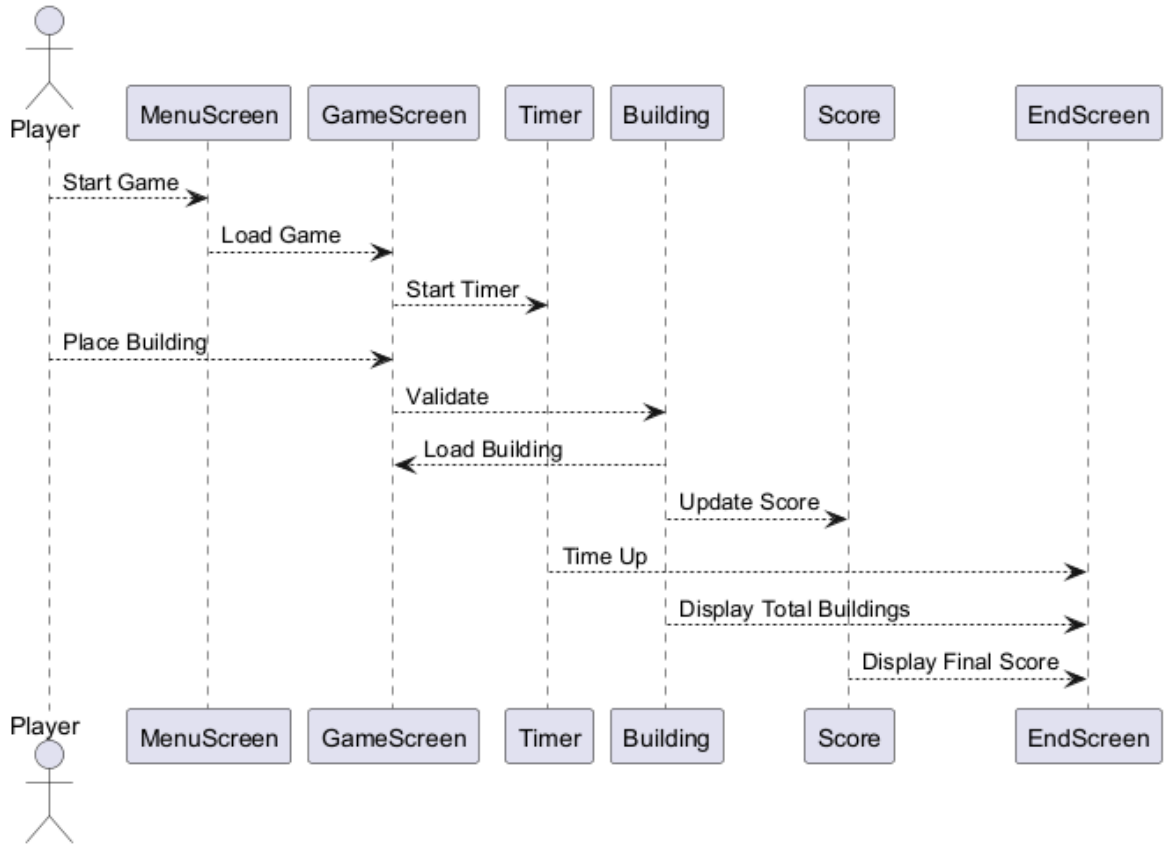
Component Diagram:



Similarly using PlantText, for PlantUML we have constructed a component diagram. This illustrates the relationships between different components of our system and their interfaces. Specifically, the keyword component is used to initiate a new component. When defining relationships, components stand out from interfaces as they are within []. A dependent

relationship between components uses the command “...>” and an associative relationship uses “--->”. Furthermore, by replacing the middle dot or dash in the arrow with a direction (up, down, left or right) you can reorientate the diagram. For example, [Score] -right-> UpdateScore. Defines the association between component Score and interface, UpdateScore.

Sequence Diagram:



Finally we have constructed a sequence diagram to illustrate the chain of messages passed between objects in the system. In this case, it shows the flow of messages required to allow a player to play and interact with the UniSim game. For ease and familiarity, we have also used PlantText, for PlantUML for this diagram. We can initialise objects using the participant keyword. However, the keyword actor is used for the player, hence the different symbol. Lifelines are automatic in PlantUML, saving time. To represent messages, arrows are drawn using the “--->” command. The order in which these are typed out determines the order they are shown on the screen. For instance, the first message in the list is “Player --> MenuScreen: Start Game”. As shown in the diagram, this becomes the top arrow in the diagram.