Architecture

Cohort 1 Team 11

Freddie Aberdeen
Mutaz Ghandour
James Given
Jasper Owen
Jungwan Park
Joe Reece
Ivan Shestakov

Architecture

We decided to use an Entity-Component-System (ECS) architecture as it keeps all systems independent and our codebase modular. Because each system is separate and does not have references to other systems, we can test or change one without worrying about side effects. In practice, this allowed us to develop systems independently of one another and add or remove code without breaking the gameplay loop.

This modularity allows us to reuse components across entities, share systems (where it is sensible to do so), and develop features quickly without broader impacts on other areas of the code. This approach has both dramatically increased productivity and facilitated good readability and long-term maintainability of code. In conclusion, this structure makes teamwork easier, allowing team members to work in parallel and smoothly combine their work later.

All diagrams linked in this document were made in PlantUML.

Entities

A core piece of the architecture is an entity which represents an object in the world. All entities are instances of the entity class and are, in essence, an ID and an unordered collection of components (discussed later). This design encouraged us to use composition rather than inheritance to create our objects, allowing us to use disparate pieces of code within a single entity. For example, we were able to give both the player and coffee entities a physics component without making the player a type of coffee or vice versa. This would not be easily possible using inheritance without code duplication.
A few examples of entity instances are:

- The player
- The coffee collectable
- The goose that attacks the player

Components

A component is a unit of data storage. There are different types of components that represent different data. The components that compose an entity determine the data it needs to function. However, the components themselves are behaviourless: their only responsibility is to store data for associated systems (detailed below).

The components in our game are responsible for storing the data needed for the game to run. Specific systems use the components to run the game.
For example, the transform component records the position and scale of an object. Any object that exists in a specific location in the world has a transform component.

The responsibilities each component has and the systems they collaborate with are shown in the following CRC card table: CRC card table | ENG1-C1G11

Components within entities

The Components contained in each entity are:

Entity	Components
Player	● PlayerComponent ● TransformComponent ● SpriteComponent ● PhysicsComponent ● CameraFollowComponent ● AnimationComponent ● AudioListenerComponent
Coffee	● InteractableComponent ● TransformComponent ● PhysicsComponent ● SpriteComponent
Goose	● InteractableComponent ● GooseComponent ● PhysicsComponent ● TransformComponent ● AnimationComponent ● SpriteComponent
Timer	● TimerComponent
Walls	● PhysicsComponent ● TransformComponent
Pressure Plate	● PhysicsComponent ● InteractableComponent ● SpriteComponent ● TransformComponent
Hidden Wall	● PhysicsComponent ● HiddenWallComponent ● SpriteComponent ● TransformComponent

As shown in the UML Structural diagram in the following list of diagrams: [Architecture Diagrams | ENG1-C1G11]

Systems

Systems contain behaviour associated with entities. Systems activate in sequence, scanning over all the entities in the game. If the entity contains the components the system is interested in, it processes that entity before moving on to the next. Systems can modify the entity's components and publish messages via the messaging system described later. Some systems provide the player with output. For example, the Rendering System displays the game on the screen.

For example, the Rendering System takes all entities with TransformComponents and SpriteComponents and draws them onto the screen, ignoring other entities.

Currently, the systems in our game and their responsibilities are:

- **The Player System:** Takes the directional input, moves the player on the screen, and tells the audio system when to play footsteps.
- **The Physics System:** Simulates and updates the velocity and position of entities with Physics Components.
- **The Physics Sync System:** Synchronises the Physics and Transform Components before the Physics System is updated.
- **The Physics To Transform System:** Takes the positions of all objects on the screen and synchronises their transformations so they are all up to date after the physics simulation is updated.
- **The World Camera System:** Makes the camera follow certain entities in the game. Currently, the only entity the camera tracks is the player, but it can track more than one.
- **The Rendering System:** Renders everything on the screen, such as our tiles and sprites, in the correct render order.
- **The Timer System:** Keeps the timer updated and accurate throughout the game.
- **Goose System:** Controls the goose entity's actions.
- **The Interactable System:** Makes any entities that contain interactable components send messages to indicate they have been interacted with.
- **The Game State System:** When the game state system receives events, it passes them to an event counter that keeps track of how many times each event has happened. Also, the Game State System ends the game when the end condition is met.
- **Audio System:** Responsible for playing sound effects and music during the game.
- **Hidden Wall System:** Makes the Hidden Wall disappear when the player steps on the corresponding Pressure Plate.

A design choice that may seem counterintuitive at first is that we used three systems for physics instead of one. Using three different physics systems helped us abstract synchronisation, simulation and output updates correctly.

Requirements met by Systems

The requirements each system will meet are:

System	Requirements	Justification
Player System	● UR_ESCAPE_MAZE	By using the Player System to control their sprite, the player shall be able to escape the maze
Physics System	● FR_BOUNDARIES ● FR_EVENTS	The Physics System will make the player's sprite unable to pass through walls it collides with, and by indicating that a collision has happened, it will cause collision-based events to happen

World Camera System	● FR_MAZE	The camera shall show the player where they are in the maze.
Rendering System	● UR_THEME ● UR_BOUNDARIES ● FR_TIMER	By making the university building appear on the screen, the rendering system shall create the university theme we are looking for, as well. By making the timer appear on the screen, the Rendering System also contributes to the requirement NFR_TIMER.
Timer System	● NFR_TIMER	The Timer System will keep track of how long the player has, fulfilling the requirement NFR_TIMER
Goose System	● UR_NEGATIVE_EVENTS ● FR_NEGATIVE_EVENTS	The Goose System operates the goose that bites the player and knocks them back, causing their score to decrease. It serves as a negative event.
Interactable System	● UR_POSITIVE_EVENTS ● UR_NEGATIVE_EVENTS ● NFR_EVENT ● NFR_SCORE ● NFR_END_GAME	Having interactable objects in our game allows us to create objects that increase the player's score in a positive event and decrease it in negative events. Also, the player currently has to interact with an object to win the game, so having an interactable system contributes to the NFR_END_GAME requirement, too.
Game State System	● FR_SCORE ● NFR_SCORE	By keeping track of the player's score, the Game State System meets the FR_SCORE requirement. Also, the Game State System keeps the score updated, which means it fulfils the NFR_SCORE requirement as well.
Audio System	● UR_AUDIO ● FR_THEME	By playing sound effects that feel relevant to a university setting, the audio system shall enhance the university theme
Hidden Walls System	● UR_HIDDEN_EVENTS ● FR_HIDDEN_EVENTS ● NFR_EVENTS	A wall that disappears when the player steps on a pressure plate is a hidden event because we have not told them about the walls disappearing or the pressure plate. This means there is a hidden event in the game.

System interaction

To enable communication between systems while keeping them separate from each other, we decided to allow the systems to send messages across the game. We did this by creating a message publisher that the systems hold references to. When a system tells the publisher to publish a message, the publisher sends it to every MessageListener that subscribed to it. Each system that receives messages contains a MessageListener that keeps track of all the messages it has received. The systems can then query the MessageListener to get unread messages whenever it is their turn to process.

Any message a MessageListener receives is added to a queue for processing when the system is updated. When the system looks through the message listener, if the current message it is reading is

relevant to the system, it is processed. Otherwise, it is discarded. While in theory we could have multiple publishers, we only need one to meet our requirements. This has the added benefit of requiring only one publisher to manage, keeping the design simple.

The messages themselves are objects that carry data. An example message could be a notification that the player has collided with a coffee power-up. When the interactable system receives and processes this message, it can publish another message in response to the one it received. The PlayerSystem can then receive that and apply the power-up. This is shown in the sequence diagram below:

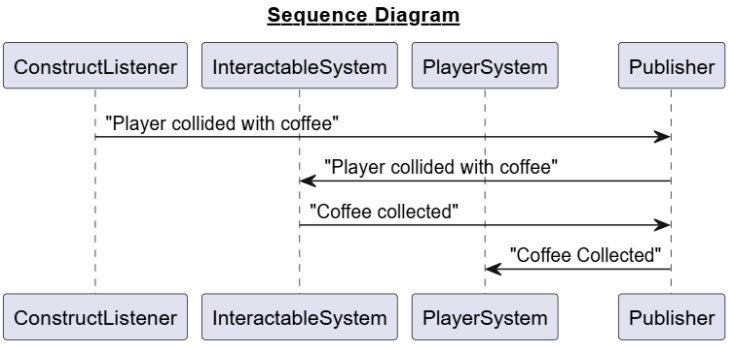

Figure 1: A Sequence diagram showing an example of messages being sent between systems. Shown above

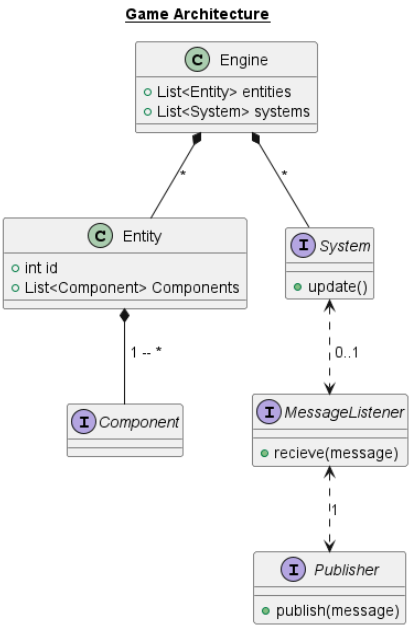

Figure 2: A component diagram showing the generic layout of our ECS architecture shown to the right.

System updates

Initially, we had each system update at variable time steps, where the computer would start rendering a new frame as soon as it finished rendering the old one and display it on the screen.

As seen in the Prototype Activity Diagram at the bottom of this list of diagrams: [Previous behaviour diagram | ENG1-C1G11]

This meant that the number of frames made per second could vary a lot depending on how powerful the computer is. While this approach did suit our *RenderingSystem*, it was unsuitable for our *PhysicsSystem* because each computer has a different frame rate, so using variable time steps would mean that the *PhysicsSystem* would behave differently for every computer on which the game was played, which is not helpful for a consistent user experience and could lead to bugs.

We did consider using fixed time steps instead, where frames are rendered in fixed time intervals. This approach would have also been more suitable for the *PhysicsSystem*. However, this would have meant that, for whichever frame rate we chose, if a computer could not run at that rate, it could not run the game, which would not meet our requirement NFR_PERFORMANCE. This meant that neither variable nor fixed time steps could be used for both the *PhysicsSystem* and *RenderingSystem*.

To solve this problem, we separated the visual parts of the game from the simulation-based parts, putting the PhysicsSystem and game simulation on fixed time steps, and the RenderingSystem and systems that control the graphics on variable time steps, so each system could be updated in the way it works best. Now we have met the NFR_PERFORMANCE requirement because intensive calculations have been moved away from fixed-step updates, allowing low-end machines to run the game.

We also created a class called Fixed Stepper, which determines when to fire the next fixed update. This ensures that there are always 60 fixed updates per second. Therefore, the Physics System always runs at 60fps.

When a system is updated, it performs its responsibilities: reading and writing messages, and acting on entities (as described in the Systems section above). Then the next system is updated, and so on. In every visible frame, we perform the number of fixed updates corresponding to the time between visible frames, then we fire a variable update that handles rendering. As shown in the Behavioural Activity diagram in the link: [Architecture Diagrams | ENG1-C1G11]

Asset management

Initially, when creating the game, we used the built-in asset manager from LibGDX to manage our assets. However, this turned out to be a naive approach because for every asset we used, we had to specify a path to the folder it was stored in whenever we used or loaded it. While this worked in the initial stages of development, when we had few assets, as we began adding more assets to the game, many file paths were left scattered throughout our code.
This soon made our code cumbersome and difficult to maintain, as we had to re-read it all to find the file path to an asset. Editing the file path meant changing it in multiple places at random points throughout the codebase.

Eventually, to fix this problem, we decided to create our own wrapper class for AssetManager, which we called AssetLoader. This class allowed us to specify our assets using symbolic identifiers.
These identifiers were passed into the Asset Loader class to get the asset, instead of specifying the file path every time we used it. This also meant that if we changed the asset file path, we only had to change the file path in one place instead of many. The Asset Loader would then convert the symbolic identifier to the file path and pass it to the asset manager, returning the result.
Finally, our asset loader class is responsible for loading the game's assets when the game starts. Instead of iterating through everything to decide whether to load it, the asset loader will iterate through a list of pre-recorded assets the game needs when it starts and load them all into memory. This means we can store all our file paths in one file and avoid duplication.

Previous diagrams
Previous versions of our structural and behavioural diagrams are available below, showing the order in which we added some of our components and systems.
[Previous version of structural diagrams | ENG1-C1G11]
[Previous behaviour diagram | ENG1-C1G11]