Ultimate City Manager - Game Requirements & Implementation Specifications

User Requirements

The Ultimate City Manager system allows the Player to create and manage a virtual city. At the start of the game, the Player must initiate a new city, laying the foundation for all subsequent activities, such as management and development. The Player can construct various buildings within the city, including residential housing, industrial workplaces, and recreational spaces. Once a building is created, the system automatically assigns residents to any available spots within that structure.

For each building constructed, the Player must choose its type, specifying whether it serves as housing, a workplace, or a recreational facility. Every workplace needs to have at least one type of obtained resource assigned (Exported, Imported, Natural, ManMade). The type of the resource a certain building creates can be changed any time by the Player. The buildings can be upgraded by the Player to earn a larger amount of the resource.

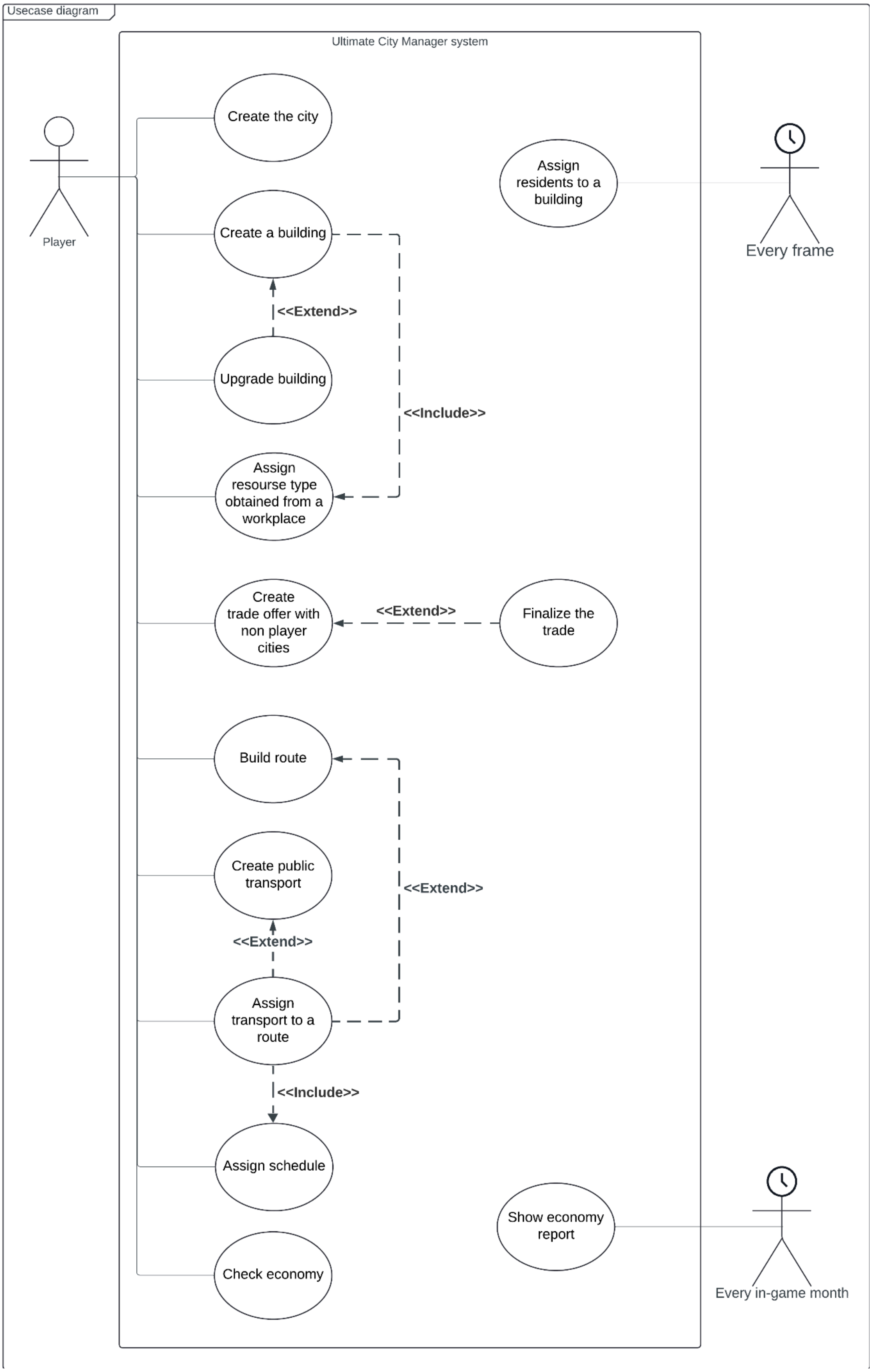
The Player is also able to engage in trading activities with non-player cities. These trades involve a process where the Player creates a trade offer and then finalizes it, confirming the resources or items to be exchanged.

In addition to buildings, the Player can construct essential infrastructure such as routes. Routes enable transportation within the city and contribute to its expansion. For routes to work, the player will need to create public transportation such as buses, trams, or metro systems. After creating these public transport vehicles, they can be assigned to specific routes to serve the city's residents. When assigning a vehicle to a route, the player also needs to specify the schedule on which the particular vehicle will operate. The schedule can be then changed any time by the Player.

Managing the economy is a crucial part of the gameplay. The Player has access to an economic overview, where they can check the city's resources and ensure efficient allocation and management. At the end of every in-game month, the system generates an economic report, summarizing the city's performance and providing insights into resource status and usage.

The system includes various actors. The Player is the primary actor, responsible for playing the game, therefore managing the city's growth and development. There are also system-level triggers, such as "Every Frame," which automatically handle tasks like assigning residents to available buildings, and "Every In-Game Month," which generates regular economic reports. These elements work together to create a dynamic and immersive experience, where the Player must balance city development, resource management, and economic growth.

Use case diagram



Use case scenario: “Assign schedule to vehicle”

UC Scenario: *"Assign Schedule to Vehicle"*

Actor: Player
Purpose and Context:
The player wants to assign a schedule to a public transport vehicle (e.g., bus, tram, metro) to optimize its operating times and frequency in the game.
Assumption:
1. The vehicle already has an assigned route, and the player is managing the transportation network.

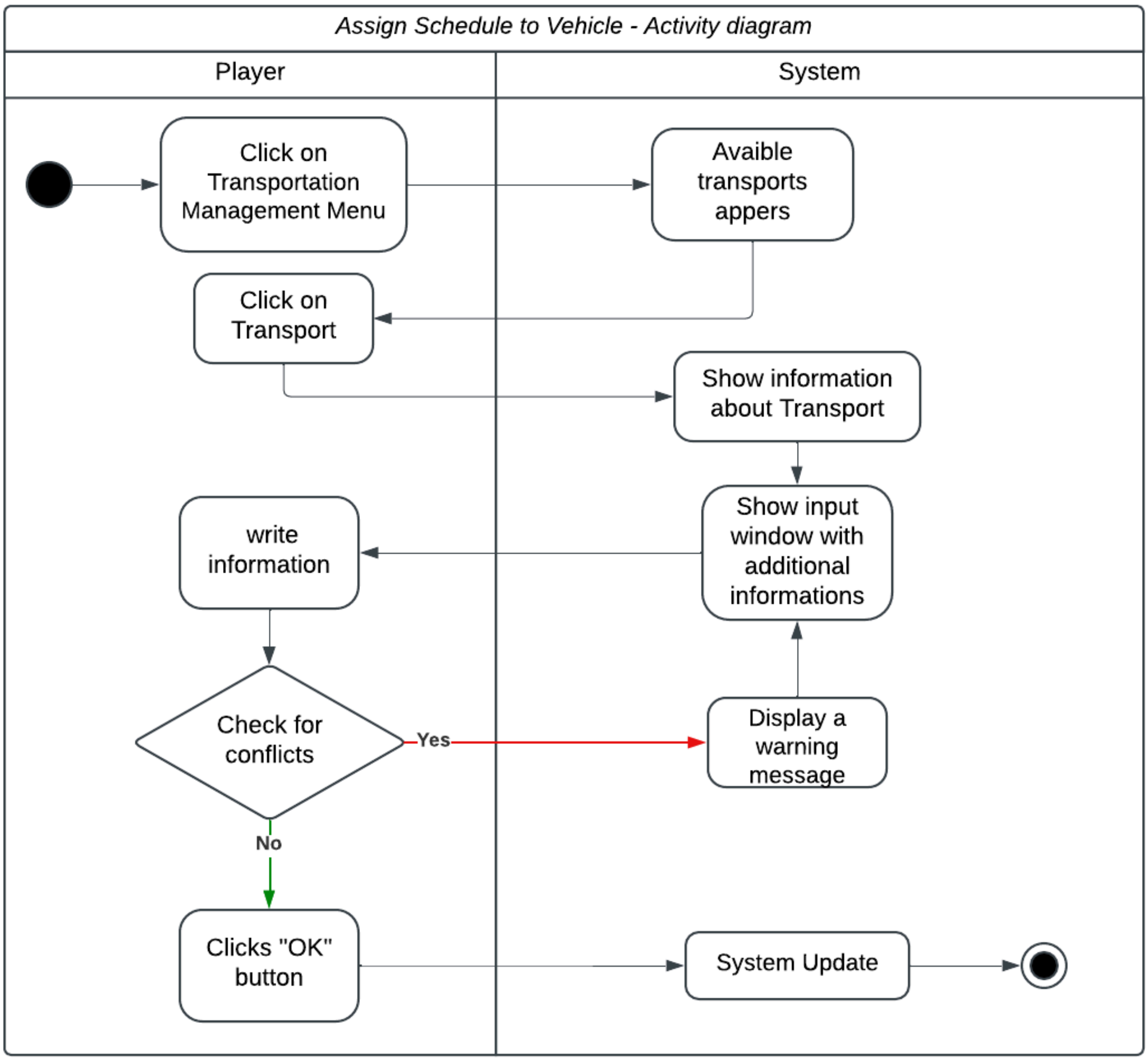
Precondition:
1. The public transport vehicle has been created, and a route is assigned to it.

Basic Flow of Events:
1. The player opens the transportation management menu and selects the desired vehicle.
2. The system displays the vehicle's current route information and scheduling options.
3. The player inputs schedule details, including start time, end time, and frequency.
4. The system validates the schedule to ensure no conflicts with other vehicles or city resources.
5. The player confirms the schedule by clicking "Assign Schedule."
6. The system updates the vehicle with the new schedule and reflects changes in the game simulation.

Alternative Flow of Events:
1. Invalid Schedule Input:
4a1 - The player enters a schedule that conflicts with another vehicle or route.
4a2 - The system displays a warning message: "Schedule conflicts with an existing route. Please adjust timing or frequency."
4a3 - Return to point 3 of the basic flow.

Postcondition:
Basic: The schedule is successfully assigned to the vehicle, and its operations begin following the new schedule.

Activity diagram: “Assign schedule to vehicle”

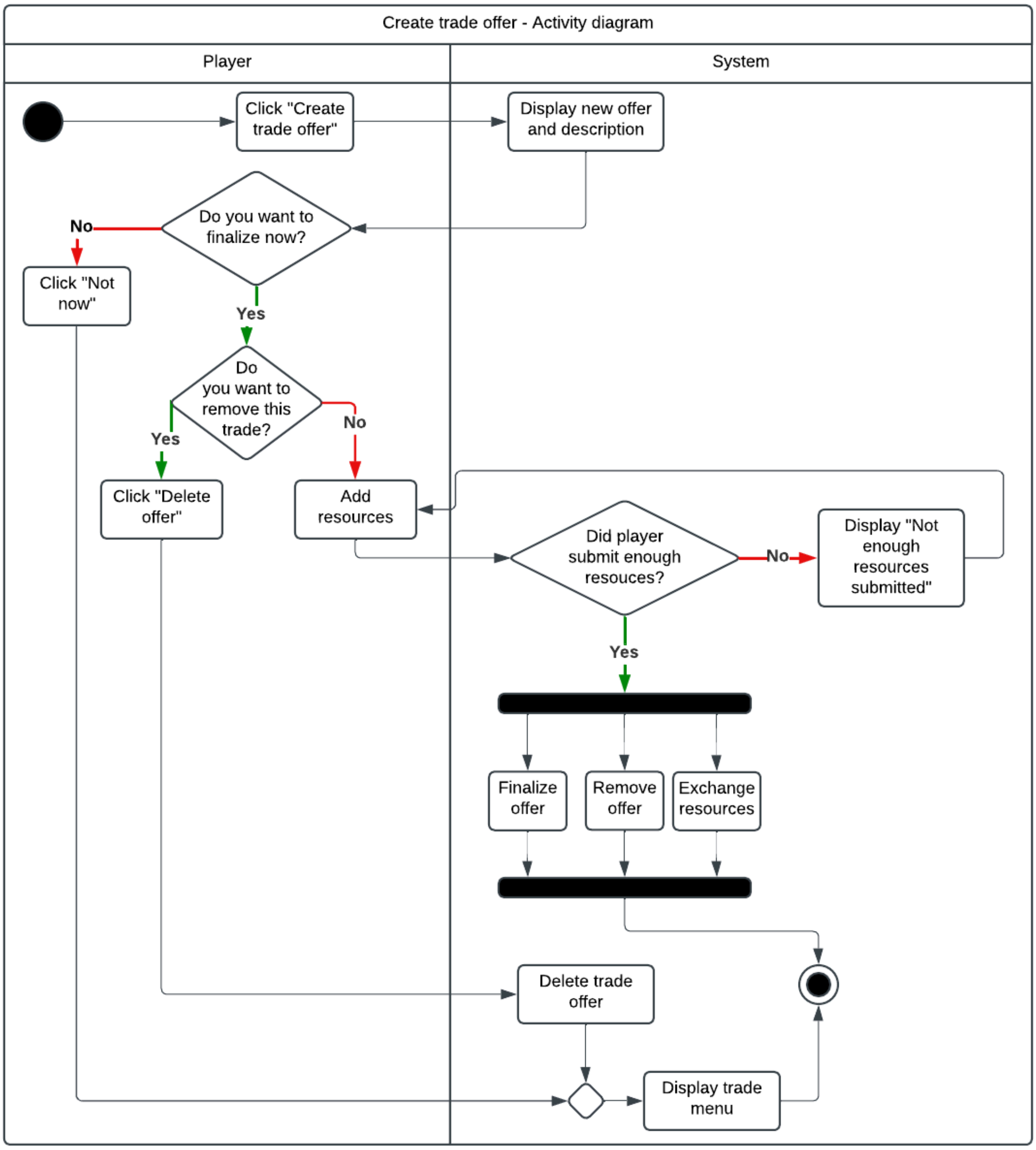


Use case scenario: “Create trade offer”

Create trade offer - Use case scenario

Actor: Player.
Purpose and context: Player wants to trade resources with another city.
Assumption:
1 - Player has achieved the opening level needed to trade.
Pre-condition:
1 - Player has opened the trade menu.
Basic flow of events:
1 - Player clicks on the "Create trade offer" button.
2 - System displays the new offer with resources needed to complete it and rewards for it, as well as a cross on top ("Not now") and "Delete offer" button.
3 - Player adds existing resources and clicks finish trade.
4 - System triggers "Finalize trade", removes the offer and exchanges player's resources.
Alternative flow of events:
1. Player reconsiders finishing the trade:
3a1 - Player clicks the "Delete offer" button.
3a2 - System deletes the offer from the list and returns to the trade menu.
2. Player wants to finalize later:
3a1 - Player clicks the "Not now" button.
3a2 - System returns to the trade menu.
3. Player clicks finish without submitting enough resources:
4a1 - System shows "Not enough resources submitted" error.
4a2 - Player returns to step 3 of basic flow.
Post-conditions:
Basic: The player got new resources in the result of exchange.
Player reconsiders finishing the trade: No changes occur.
Player wants to finalize later: The trade offer stays in the trade menu.

Activity diagram: “Create trade offer”



Use case scenario: “Assign transport to a route”

UC Scenario: "Assign transport to the road"

Actor: Player

Purpose and Context:

The player wants to assign a public transport vehicle (e.g., bus, tram, metro) to a specific route to set up or optimize the transport system across the city.

Assumption:

1. The public transport vehicle (bus, tram, or metro) has already been created.
2. Created Roads(or rails) support the selected vehicle.

Precondition:

1. A public transport route has been created in the city.

Basic Flow of Events:

1. The player opens the transportation management menu and selects a created public transport vehicle.
2. The system displays the list of available routes within the city that the vehicle can be assigned to.
3. The player selects the route where the vehicle will operate.
4. The player confirms the assignment of the transport vehicle to the chosen route.
5. The system updates the vehicle's schedule and displays changes in the transport simulation (e.g., the vehicle begins moving along the route).

Alternative Flow of Events:

1. Alternative way. Assign vehicle to a road:
   - 1a1. The player chooses a route on map.
   - 1a2. The system displays contextual menu of the selected route.
   - 1a3. Player navigates to the transportation managment sub-menu.
   - 1a4. The system displays the list of available vehicles that the selected route can be assigned to.
   - 1a5. The player selects a vehicle to operate on the selected route.
   - 1a6. The player returns to step 4 of the basic flow.
2. Invalid Route Assignment:
   - 4a1. The player selects a route that is not compatible, or not complete.
   - 4a2. The system displays an error message: "Selected transport vehicle can't be assigned to this route. Please select a different one."
   - 4a3. The player returns to step 2 of the basic flow.

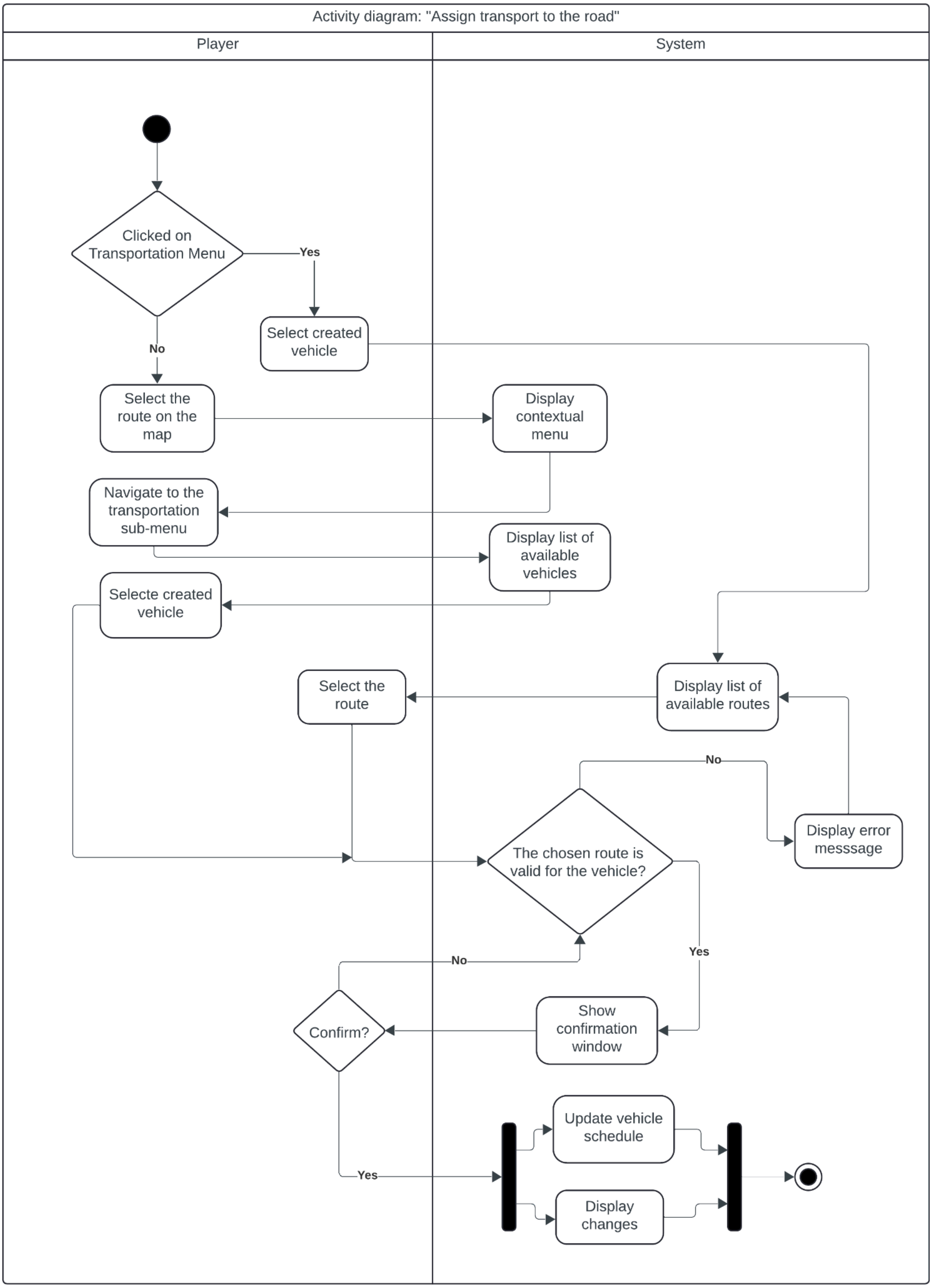
Post-condition:

Basic: The vehicle is successfully assigned to the selected route and begins operating along that route as per the city’s transport schedule.

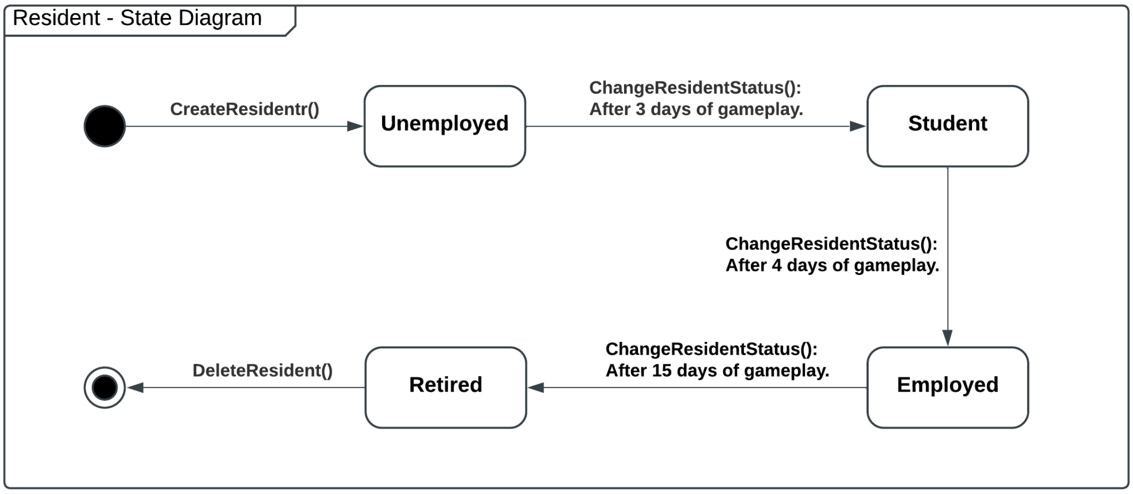
Alternative way: The vehicle is successfully assigned to the selected route and begins operating along that route as per the city’s transport schedule.

Invalid Route: The system prevents the assignment of a vehicle to an incompatible route, allowing the player to reassign it correctly.

Activity diagram: “Assign transport to a route”



State Diagram



State Diagram - Design Description

The state diagram describes the lifecycle of a Resident object in the system. When using the **CreateResident()** method, a new Resident object is created with the initial status of Unemployed. The state of the resident evolves based on game events and specific time periods, as described below.

States and Transitions:

Unemployed:
The initial state of a resident after creation.
Next step requirement: Occurs via **ChangeResidentStatus()** after 3 days of gameplay. This transition signifies the resident enrolling in an educational program.

Student:
Indicates that the resident is currently pursuing education.
Next step requirement: Occurs via **ChangeResidentStatus()** after 4 days of gameplay, signifying the resident completing education and obtaining employment.

Employed:
Represents the state where the resident is actively working.
Next step requirement: Occurs via **ChangeResidentStatus()** after 15 days of gameplay. This transition reflects the resident reaching retirement age.

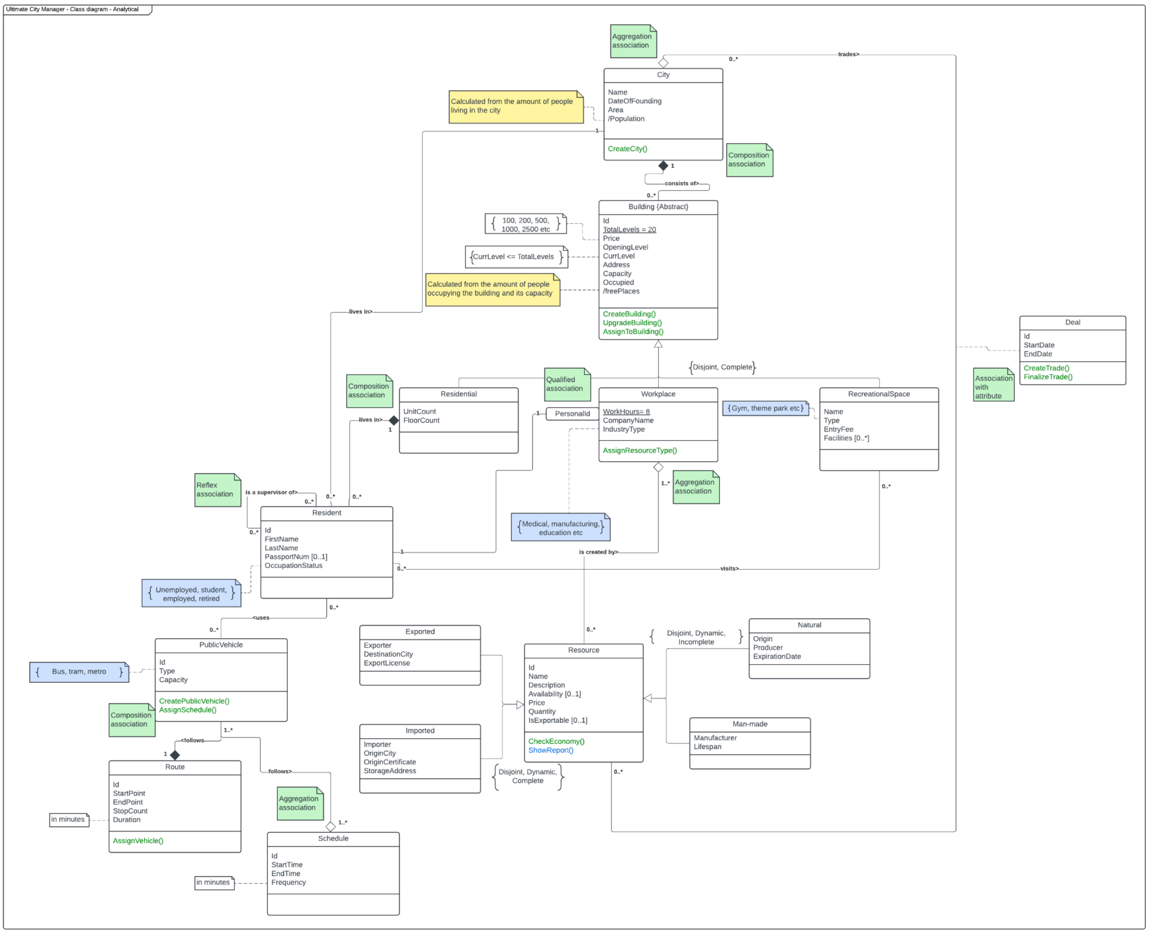
Retired:
Indicates the resident has exited the workforce and is no longer employed.

Next step requirement: The resident can be removed from the system by invoking the **DeleteResident()** method, which ends the lifecycle of the Resident.

Important Notes

The lifecycle starts with the **CreateResident()** method and ends with the **DeleteResident()** method.
Transitions are triggered by the **ChangeResidentStatus()** method, based on time elapsed in gameplay (3, 4, or 15 days).
The state of the resident object is essential for determining available actions and interactions within the system.

Class diagram - Analytical



Analytical Class diagram - Design Description

The analytical class diagram above shows the core structure of the city management game. The game simulates a process of city development, resource management, and transportation systems. By managing buildings, residents, workplaces, and public services—players learn the principles of planning, economics, and specifically management.

At the heart of the system is the City class, the player creates only one at the start of the game, providing details like the city's name, founding date, area, and population size. In the city different types of Buildings can be erupted. These buildings include Residential housing, which are defined by the number of units and the number of floors, Workplaces for employment, and RecreationalSpaces like gyms or parks and list the facilities and the entry fee they have. Each type of building has distinct attributes, such as unique ID, address, total levels, current level, capacity, and information about occupancy. Players can upgrade buildings to accommodate more people or raise their efficiency.

The Resident class represents the city's population. Each resident has attributes like FirstName, LastName, OccupationStatus, and PassportNum. A resident's employment status is also tracked—students, employed, unemployed, or retired.

Workplaces are critical for the city's economy, characterized by an 8-hour working day, the businesses operating within them and come in different forms, such as Medical, Manufacturing, or Education sectors. These workplaces employ residents and utilize city Resources. For instance, a hospital may require medical supplies or a manufacturing plant may need raw materials. Resources are trucked by their id, name, availability, price, quantity and are divided into Natural, which detail their origin, producer, and expiration date, and Man-made types, which detail their manufacturer and lifespan. The game allows players to either produce them locally in Workplaces or import/export them via the Imported and Exported classes. In case of export, we must know information about an exporter, destination city and export license. In case of import, we must know information about an importer, origin city, origin certificate and storage address. This introduces a layer of economic management, where students must balance resource production with trade and supply chain management.

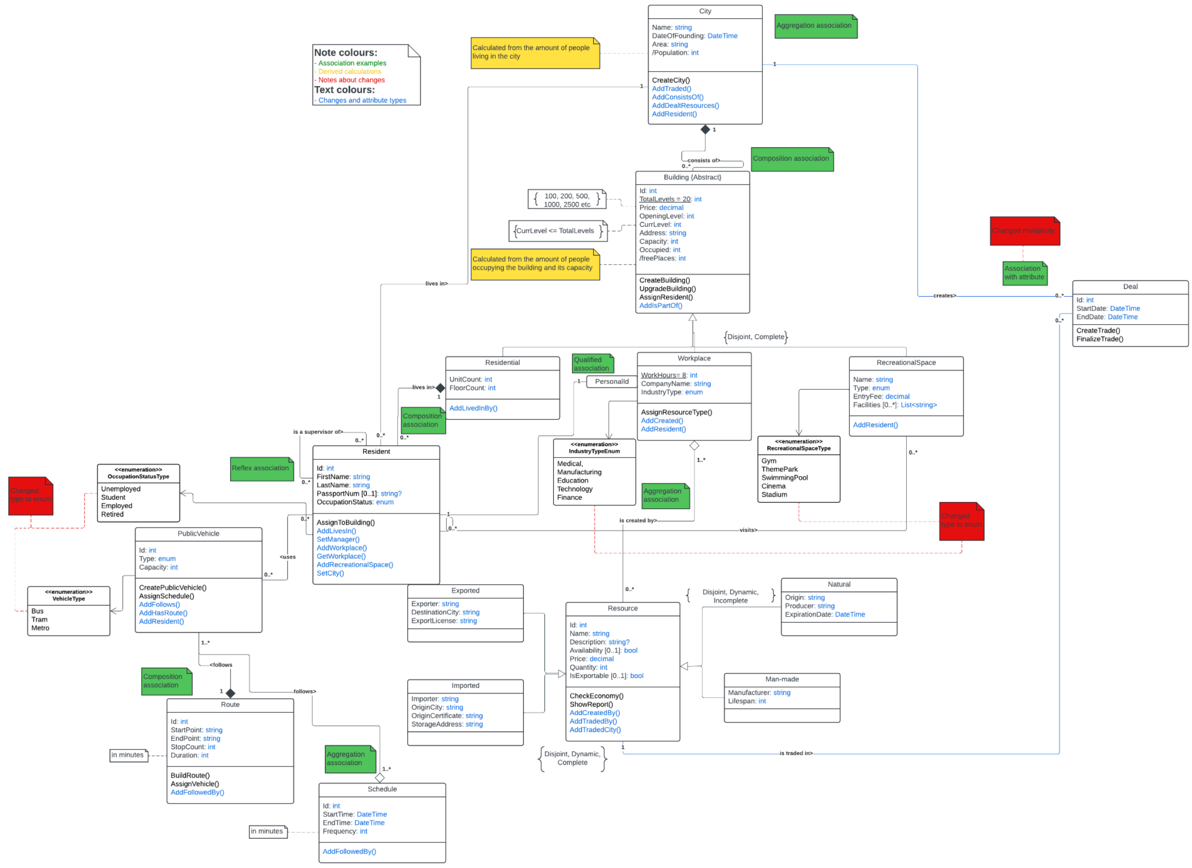
Transportation is another key aspect of the game, managed through the PublicVehicle, Route, and Schedule classes. Players must assign PublicVehicles (like buses or trains) to specific routes, determining start points and stops. The Schedule class allows players to set the frequency of these routes, optimizing public transit to keep the city running efficiently. Mismanagement here could result in congestion or inefficient resource allocation, teaching students the importance of planning.

The Deal class in the diagram introduces a layer of negotiation and transaction management, where players can engage in deals for resource trading. Deals track when they are created and when they are finalized, allowing players to engage in strategic economic planning.

This diagram includes plenty of UML constructs, which do not exist in modern programming languages. Certain parts of the diagram that are not clear are explained in notes, displayed on the diagram, as well as in the Design Decisions section.

In the next section, the analytical class diagram will be transformed into a design class diagram, showcasing the information regarding the attribute implementation.

Design diagram - Associations



Design Decisions

The aforementioned design class diagram showcases changes, needed to adjust the UML constructs to the C# language, as some of them cannot be represented there directly. These changes are highlighted in blue. The final version of the project will use XML to handle data storage and retrieval. This approach was chosen for its simplicity, ease of use, and ability to securely and reliably persist data across sessions. Following section will describe all necessary transformations, regarding various attribute types, used in the project, and managing associations.

Data validation

.NET System.ComponentModel.DataAnnotations namespace was mainly used for specifying validation rules via using different kinds of annotations. These annotations provide a structured standardized way of checking data validity, as well as having high-level of customizability and being secure and consistent.

Tagged values

While note notation for tagged values is possible in UML, the most efficient way of implementing them in C# is using enumeration. Each enum is a class that stores possible values for the attribute. It ensures type safety, meaning that only valid values can be passed. Classes created for enums are highlighted in the design diagram (e.g., OccupationStatusType for the attribute OccupationStatus in Resident class).

Complex attributes

Complex attributes may hold multiple values of different types and have a certain structure. They might need a separate class for structural items representation, however, for this class diagram, they all could be replaced by default C# structures, such as DateTime (a moment in time, typically expressed as a date and time of the day). For example, it was used for setting StartTime and EndTime attributes in Schedule class.

Derived attributes

The method of implementing derived attributes via getters and expression bodies was chosen. These attributes are read-only (cannot be modified), therefore don't have setters and are set automatically.

For instance, attribute FreePlaces in Building class (showing how many places are not occupied by people yet) is set like this using Capacity and Occupied attributes of the same class:

```
public int FreePlaces => Capacity - Occupied;
```

Optional attributes

Showcased in the class diagram with the multiplicity range of [0..1], these attributes may hold no value. They are marked nullable (e.g., with a type of "string?") in C#, meaning they can either contain a null or hold a value of a certain type

Aggregation

Methods for creating the connection using reverse connection were implemented. Objects can exist separately (e.g. classes PublicVehicle and Schedule).

Composition

Methods for creating the connection using reverse connection were implemented. Object that is a "part" cannot exist without the "whole" (e.g. classes Residential and Resident, as deletion of a residential building should delete the Resident objects that are connected to it).

Association class

Methods for creating the connection using reverse connection were implemented. The multiplicity of association was changed (e.g.: classes City and Resource create an instance of Deal class and have two [1..0*] associations with it).

Basic association

Methods for creating the connection using reverse connection were implemented.

Qualified association

Methods for creating the connection using reverse connection were implemented. Dictionary for storing the qualifier and reference to the related object was introduced (e.g. in class Resident Dictionary<int, Workplace> _workplaces was created, as well as a GetWorkplace() method).

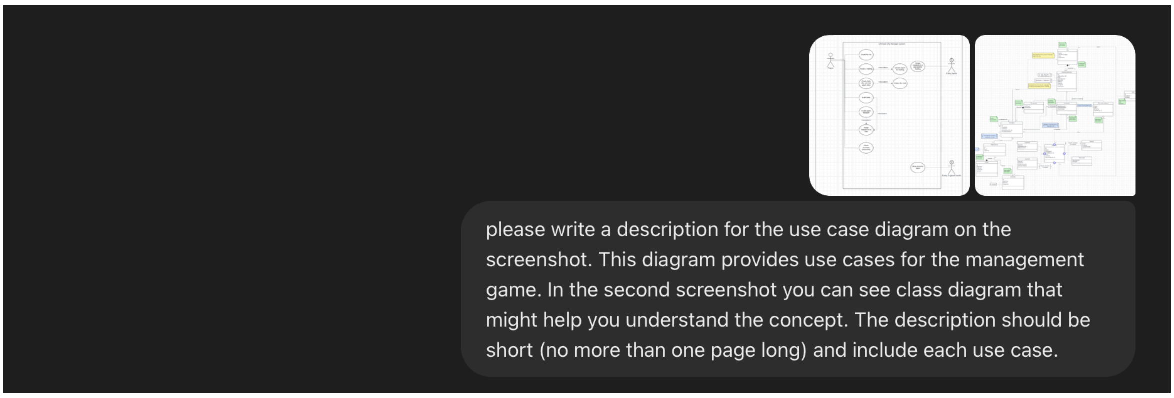
Reflex association

Methods for creating the connection using reverse connection were implemented (e.g. SetManager() in Resident class).

AI Usages:

Use case diagram description:

Prompt:



Output:

"

Use Case Diagram Description for "Ultimate City Manager System"

The use case diagram illustrates the primary functionalities and interactions available to the **Player** in the "Ultimate City Manager System," a city management simulation game. It includes the following use cases:

1. **Create the City**:
The Player can initiate a new city within the game, forming the foundation for subsequent management and development activities.

2. **Create a Building**:
The Player can construct various types of buildings. This process includes selecting the type of building (e.g., Residential, Workplace, or Recreational) and assigning residents to suitable buildings.

o **Choose Type of the Building** (included):
The Player must specify the type of building during its creation, such as residential housing, industrial workplaces, or recreational spaces.

3. **Create Trade Offer with Non-Player Cities**:
The Player can engage in trading activities with other cities to exchange resources. This process involves two steps:

o **Finalize the Trade** (included):
The Player must confirm the details of the trade, including items or resources to be exchanged.

4. **Build Roads:**
The Player can lay down infrastructure by constructing roads, enabling transportation and city expansion.

5. **Create Public Transport:**
The Player can establish public transportation systems, such as buses, trams, or metro lines.

o **Assign Transport to a Road** (included):
After creating public transport vehicles, the Player must allocate them to specific routes for effective operations.

6. **Check Economy (Resources):**
The Player can review the city's economic status, including resource availability and distribution, to ensure efficient management and planning.

7. **Show Economy Report:**
At the end of every in-game month, the system automatically generates an economic report summarizing the city's performance and resource status.

Actor Interactions:

- **Player:** The primary actor, responsible for managing all city-related activities, including construction, trading, transportation, and resource management.

- **Every Frame:** A system-level trigger that handles real-time events, such as assigning residents to available buildings.

- **Every In-Game Month:** A time-based system trigger for generating periodic economic reports.

This diagram provides an overview of the Player's capabilities in managing and growing the city while highlighting dependencies between use cases for building, trading, and public services.

"

The AI output helped write the monotonous core of the text that was modified after to better describe the system. The idea of describing actor interactions was also taken and modified as it is a good way to describe and translate the operation process of the system actors on paper.