REPORT

HALF STACK City Builder

Group Members:

Shinn-Ru Hung u23576996

Sashen Inder Gajai u23526310

Devan De Wet u05169098

Yuthika Harripersad u23562732

Keanu Ellary u22893459

Micheal Stone u21497682

Jamean Groenewald u23524121

Link to this document: Report: Half Stack City-Builder

Task 4.1: Research Brief

Urban development, as defined by GSDRC, is "the social, cultural, economic, and physical development of cities, and the underlying causes of these processes." This development is essential for cities as they adapt to growing populations, aiming to meet and satisfy the diverse needs of residents and industries. Successful urban development focuses on integrating critical areas, including transportation networks, public services, and zoning for residential, commercial, and industrial spaces. Cohesiveness in these areas is fundamental, allowing a city to function efficiently and effectively to meet current and future demands.

Principles of City Management

City management involves coordinating different urban sectors to support steady and sustainable growth. Key principles include:

- **Productivity**: Efficient use of city resources is essential for driving urban progress. Productivity-focused policies help cities build a strong economic future.
- **Sustainability**: Responsible, sustainable resource use extends the city's lifespan and promotes long-term prosperity. This principle encourages urban development that respects environmental limits and preserves resources for future generations.
- **Adaptability**: A city must be prepared for a range of challenges, from climate shifts to economic changes. For instance, South African cities must plan for seasonal droughts that impact water supply and influence various city services and sectors.

- **Health**: A city must be a healthy environment for its residents. Ensuring access to clean air, green spaces, and healthcare services improves residents' quality of life and overall well-being.
- **Inclusion**: Involving residents in decision-making processes strengthens the city’s community foundation. Opportunities for civic engagement, such as voting on policies and government representation, empower citizens and promote an inclusive urban culture.
- **Safety**: Ensuring public safety is a cornerstone of city management. This involves enforcing building codes, improving emergency response systems, and providing safe public spaces for residents.

The Role of Key Urban Components

To achieve cohesive urban development, several critical components play unique roles:

- **Transportation**: A well-planned transportation system improves mobility, reduces congestion, and connects residents to jobs, services, and recreation, enhancing the city’s accessibility and efficiency.
- **Government**: Effective governance oversees the city’s strategic planning, resource allocation, and public policies. Governmental institutions coordinate and implement urban development plans to align with community needs and goals.
- **Resource Management**: Managing natural resources, such as water and energy, ensures that essential services are available sustainably. Efficient resource management also helps cities withstand environmental challenges.
- **Residential**: Housing development and zoning provide residents with safe, affordable living spaces. Residential areas must be well-integrated with services, amenities, and transportation for a balanced quality of life.
- **Commercial**: Commercial zones drive economic activity, providing jobs and essential services. These areas are vital for urban prosperity and often serve as social and cultural hubs within a city.
- **Industrial**: Industrial areas support manufacturing and logistics, fostering economic growth and employment. These zones are typically separated from residential areas to mitigate pollution and congestion.
- **Public Services**: Public services, including healthcare, education, and emergency services, are foundational to city life, ensuring residents have access to essential resources and support.

Together, these principles and components support a city’s growth and adaptability, forming a framework that helps urban environments thrive amidst evolving needs and challenges.

Design and Assumptions:

Our system design draws on these urban principles and components to reflect real-life city management processes. We divided the system into distinct departments that manage different city sectors, each utilising commands specific to their area. This compartmentalization allows for isolated operation of different areas and processes, with interactions controlled as needed.

Understanding that resources are limited and specific to each city, we implemented a resource management system to centralise the management of supply and demand across government, departments, and other areas. This approach ensures efficient resource distribution and usage.

The assumption that tax rates and utility provision must adapt dynamically to meet changing demands informed our choice of design patterns, leading us to utilise Command and Strategy patterns for these components. Additionally, we assumed that citizen satisfaction would be affected by multiple factors, including tax rates, living standards (e.g., adding amenities), and basic utilities like water and electricity. This understanding guided the integration of these elements to capture the real-world dynamics of urban living.

References

1. GSDRC - Governance, social development, conflict and humanitarian knowledge services. (2016). *Key definitions - GSDRC*. [online] Available at: https://gsdrc.org/. [Accessed 12 Oct. 2024]

2. Lai, S.-K. (2017). City Management: Theories, Methods, and Applications (Book Proposal). *ResearchGate*. [online] Available at:https://doi.org/. [Accessed 12 Oct. 2024]

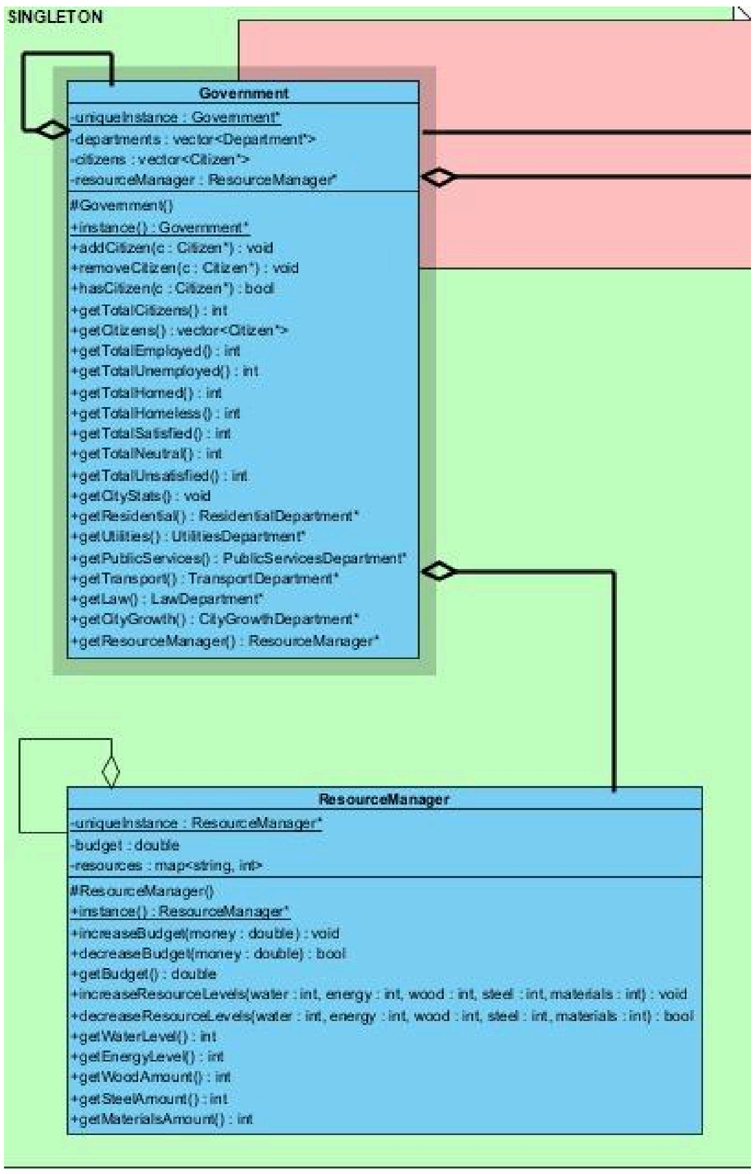
3. Unhabitat.org. (2024). *UN-Habitat - A Better Urban Future | UN-Habitat*. [online] Available at: https://unhabitat.org/. [Accessed 12 Oct. 2024]

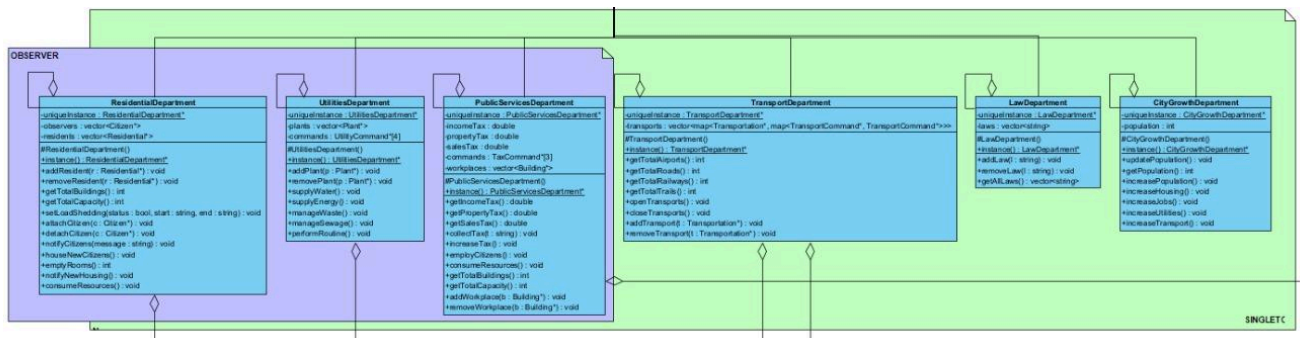
Task 4.2: Design Pattern Application

Now we delve into how each design pattern has been used to address the functionalities of the system.

Singleton Design Pattern

Singleton was used to accompany for the Government and the various functionalities such as taxation, city growth, resource management and allocation, and all the other various functionalities they provide.


SINGLETON
Government
-uniqueInstance : Government*
-departments : vector<Department*>
-citizens : vector<Citizen*>
-resourceManager : ResourceManager*
#Government()
+instance() : Government*
+addCitizen(c : Citizen*) : void
+removeCitizen(c : Citizen*) : void
+hasCitizen(c : Citizen*) : bool
+getTotalCitizens() : int
+getCitizens() : vector<Citizen*>
+getTotalEmployed() : int
+getTotalUnemployed() : int
+getTotalHomed() : int
+getTotalHomeless() : int
+getTotalSatisfied() : int
+getTotalNeutral() : int
+getTotalUnsatisfied() : int
+getCityStats() : void
+getResidential() : ResidentialDepartment*
+getUtilities() : UtilitiesDepartment*
+getPublicServices() : PublicServicesDepartment*
+getTransport() : TransportDepartment*
+getLaw() : LawDepartment*
+getCityGrowth() : CityGrowthDepartment*
+getResourceManager() : ResourceManager*
ResourceManager
-uniqueInstance : ResourceManager*
-budget : double
-resources : map<string, int>
#ResourceManager()
+instance() : ResourceManager*
+increaseBudget(money : double) : void
+decreaseBudget(money : double) : bool
+getBudget() : double
+increaseResourceLevels(water : int, energy : int, wood : int, steel : int, materials : int) : void
+decreaseResourceLevels(water : int, energy : int, wood : int, steel : int, materials : int) : bool
+getWaterLevel() : int
+getEnergyLevel() : int
+getWoodAmount() : int
+getSteelAmount() : int
+getMaterialsAmount() : int

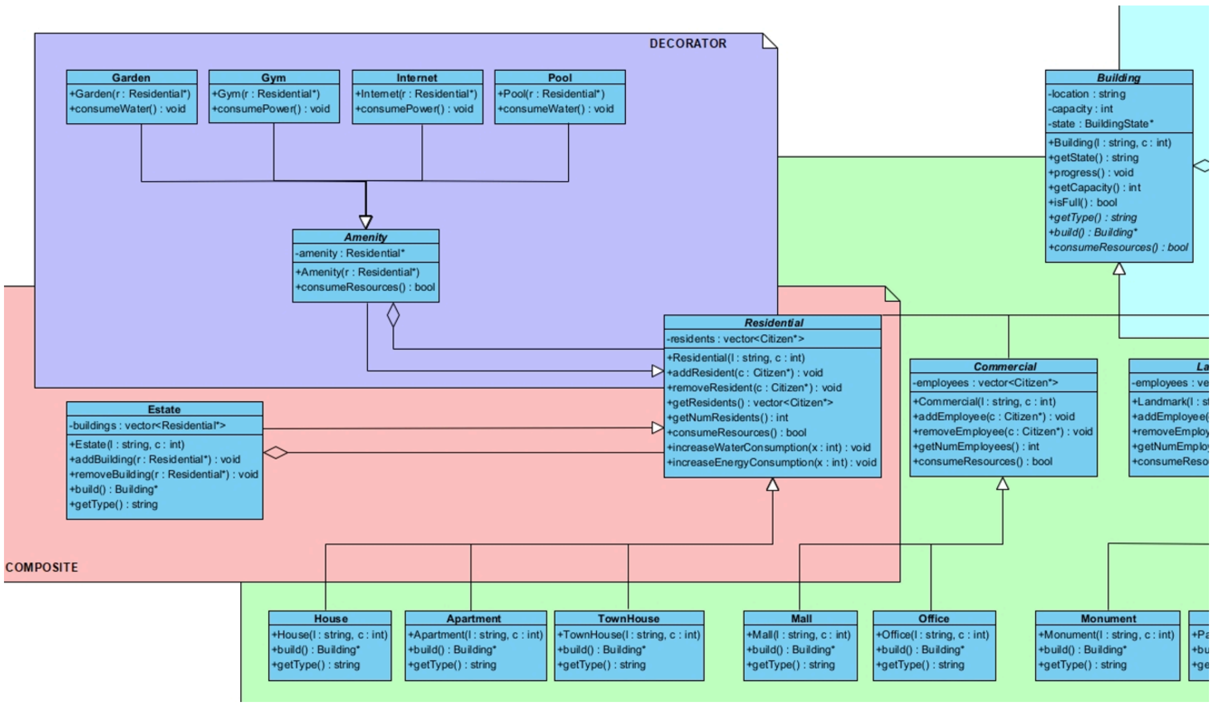

OBSERVER
ResidentialDepartment
-uniqueInstance : ResidentialDepartment*
-observers : vector<Citizen*>
-residents : vector<Residential*>
#ResidentialDepartment()
+instance() : ResidentialDepartment*
+addResident(r : Residential*) : void
+removeResident(r : Residential*) : void
+getTotalBuildings() : int
+getTotalCapacity() : int
+setLoadShedding(status : bool, start : string, end : string) : void
+attachCitizen(c : Citizen*) : void
+detachCitizen(c : Citizen*) : void
+notifyCitizens(message : string) : void
+houseNewCitizens() : void
+emptyRooms() : int
+notifyNewHousing() : void
+consumeResources() : void
UtilitiesDepartment
-uniqueInstance : UtilitiesDepartment*
-plants : vector<Plant*>
-commands : UtilityCommand*[4]
#UtilitiesDepartment()
+instance() : UtilitiesDepartment*
+addPlant(p : Plant*) : void
+removePlant(p : Plant*) : void
+supplyWater() : void
+supplyEnergy() : void
+manageWaste() : void
+manageSewage() : void
+performRoutine() : void
PublicServicesDepartment
-uniqueInstance : PublicServicesDepartment*
-incomeTax : double
-propertyTax : double
-salesTax : double
-commands : TaxCommand*[3]
-workplaces : vector<Building*>
#PublicServicesDepartment()
+instance() : PublicServicesDepartment*
+getIncomeTax() : double
+getPropertyTax() : double
+getSalesTax() : double
+collectTax(t : string) : void
+increaseTax() : void
+employCitizens() : void
+consumeResources() : void
+getTotalBuildings() : int
+getTotalCapacity() : int
+addWorkplace(b : Building*) : void
+removeWorkplace(b : Building*) : void
TransportDepartment
-uniqueInstance : TransportDepartment*
-transports : vector<map<Transportation*, map<TransportCommand*, TransportCommand*>>>
#TransportDepartment()
+instance() : TransportDepartment*
+getTotalAirports() : int
+getTotalRoads() : int
+getTotalRailways() : int
+getTotalTrails() : int
+openTransports() : void
+closeTransports() : void
+addTransport(t : Transportation*) : void
+removeTransport(t : Transportation*) : void
LawDepartment
-uniqueInstance : LawDepartment*
-laws : vector<string>
#LawDepartment()
+instance() : LawDepartment*
+addLaw(l : string) : void
+removeLaw(l : string) : void
+getAllLaws() : vector<string>
CityGrowthDepartment
-uniqueInstance : CityGrowthDepartment*
-population : int
#CityGrowthDepartment()
+instance() : CityGrowthDepartment*
+updatePopulation() : void
+getPopulation() : int
+increasePopulation() : void
+increaseHousing() : void
+increaseJobs() : void
+increaseUtilities() : void
+increaseTransport() : void
SINGLETC

We used singleton because we only need one instance of these. The Government singleton will control and manage the different requirements of the system such as tax collection for the various citizens, citizen registration and deregistration, allocation of the various resources that is needed for construction of buildings.

Each different department is its own singleton. They handle the different requirements for tax collection (specifying what type of tax to collect), utilities (setting the utilities for the buildings like waste management, water supply, energy supply and sewage management), public services (for employment, adding workplaces, collecting tax and managing citizen satisfaction), transport (having means of travelling around the city and a way to open and close businesses to accompany for the realistics of an actual city life), laws (add laws and make the place feel like your own), and city growth (this accompanies for population growth, making sure that the city cannot exceed set maximums and a way to notify the user on how to rectify passing these maximums, adding a complex layer on the game).

The resource manager singleton accompanies for the allocation of resources that is contained within the system. The user needs to have enough resources if they want to build a new building and this will keep track of when more resources are needed. If new resources are needed, then the user is notified and they can purchase more resources. This also holds a budget to add another layer of complexity. This budget is essentially the user's wallet. If they have no money, then they will not be able to build any buildings.

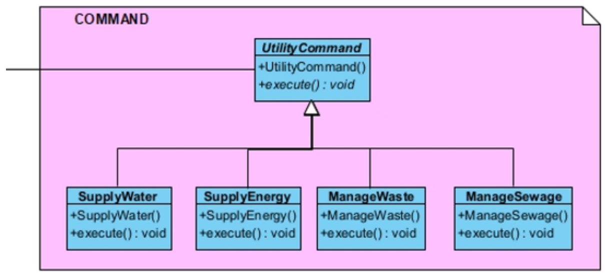
Decorator Design Pattern:

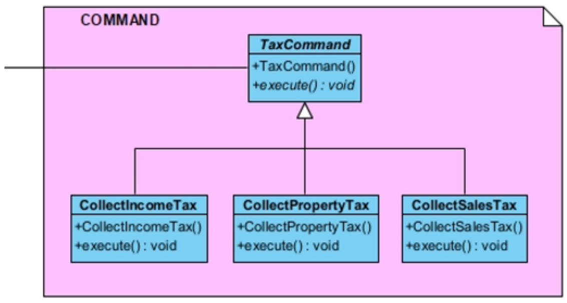

DECORATOR
Garden
+Garden(r : Residential*)
+consumeWater() : void
Gym
+Gym(r : Residential*)
+consumePower() : void
Internet
+Internet(r : Residential*)
+consumePower() : void
Pool
+Pool(r : Residential*)
+consumeWater() : void
Amenity
-amenity : Residential*
+Amenity(r : Residential*)
+consumeResources() : bool
Building
-location : string
-capacity : int
-state : BuildingState*
+Building(l : string, c : int)
+getState() : string
+progress() : void
+getCapacity() : int
+isFull() : bool
+getType() : string
+build() : Building*
+consumeResources() : bool
Residential
-residents : vector<Citizen*>
+Residential(l : string, c : int)
+addResident(c : Citizen*) : void
+removeResident(c : Citizen*) : void
+getResidents() : vector<Citizen*>
+getNumResidents() : int
+consumeResources() : bool
+increaseWaterConsumption(x : int) : void
+increaseEnergyConsumption(x : int) : void
Commercial
-employees : vector<Citizen*>
+Commercial(l : string, c : int)
+addEmployee(c : Citizen*) : void
+removeEmployee(c : Citizen*) : void
+getNumEmployees() : int
+consumeResources() : bool
Estate
-buildings : vector<Residential*>
+Estate(l : string, c : int)
+addBuilding(r : Residential*) : void
+removeBuilding(r : Residential*) : void
+build() : Building*
+getType() : string
COMPOSITE
House
+House(l : string, c : int)
+build() : Building*
+getType() : string
Apartment
+Apartment(l : string, c : int)
+build() : Building*
+getType() : string
TownHouse
+TownHouse(l : string, c : int)
+build() : Building*
+getType() : string
Mall
+Mall(l : string, c : int)
+build() : Building*
+getType() : string
Office
+Office(l : string, c : int)
+build() : Building*
+getType() : string
Monument
+Monument(l : string, c : int)
+build() : Building*
+getType() : string

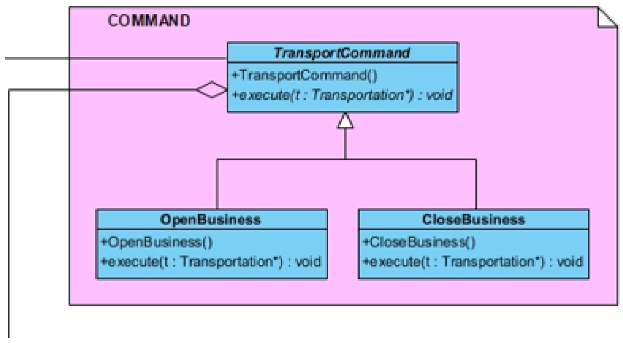
The Decorator Pattern is used to extend the functionality of residential buildings by adding various amenities without modifying their existing structure. This is particularly useful for dynamically enriching buildings with additional features such as a *Garden*, *Gym*, *Internet*, or *Pool*.

By using the decorator pattern, residential buildings can be wrapped with multiple amenities, adding features on demand. This approach promotes flexibility, allowing buildings to gain or lose capabilities dynamically without altering their fundamental design.

Command Design Pattern






COMMAND
TransportCommand
+TransportCommand()
+execute(t : Transportation*) : void
OpenBusiness
+OpenBusiness()
+execute(t : Transportation*) : void
CloseBusiness
+CloseBusiness()
+execute(t : Transportation*) : void

1. Utilities Management Commands

The UtilitiesDepartment is responsible for initiating and managing utility-related commands. It triggers specific commands based on city management needs, like supplying resources or handling waste. The actual department responsible for executing utility operations.

These are specific implementations of UtilityCommand:

- SupplyWater: Manages water supply throughout the city.
- SupplyEnergy: Handles the energy supply, like electricity or gas distribution.
- ManageWaste: Oversees waste management, such as garbage collection.
- ManageSewage: Controls sewage system management and maintenance.

2. Tax Collection Commands

The PublicServicesDepartment initiates tax-related commands, which handle the collection of various taxes from citizens.The Citizen class or entity that the tax commands act upon. The receiver processes the tax payment and adjusts the citizen’s financial status accordingly.
Specific implementations of TaxCommand:

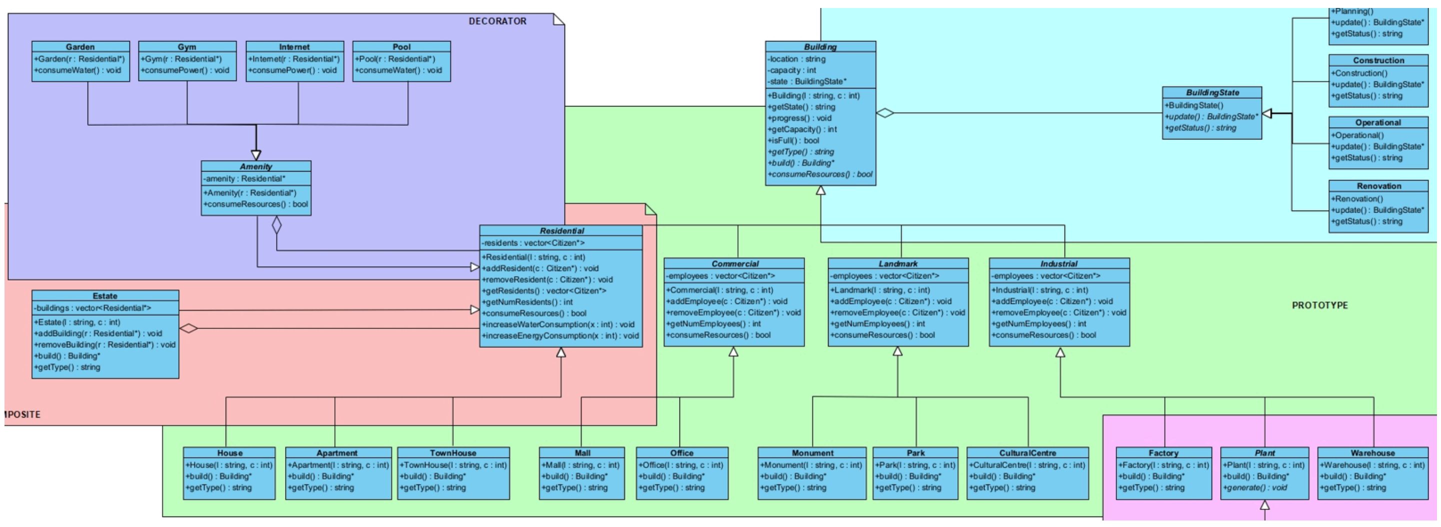
- CollectIncomeTax: Handles the collection of income taxes from citizens.
- CollectPropertyTax: Manages property tax collection for real estate owned by citizens.
- CollectSalesTax: Oversees the collection of sales taxes on purchases.

3. Transportation Management Commands

The TransportDepartment initiates commands related to transportation, specifically opening or closing transportation-related businesses.The Transportation class or entity responsible for managing the transport infrastructure. This receiver handles the opening or closing of businesses and related logistics
Specific implementations of TransportCommand:

- OpenBusiness: Activates or opens transportation-related businesses, such as bus lines or subway stations.
- CloseBusiness: Shuts down or suspends operations of transportation services.

Prototype Pattern



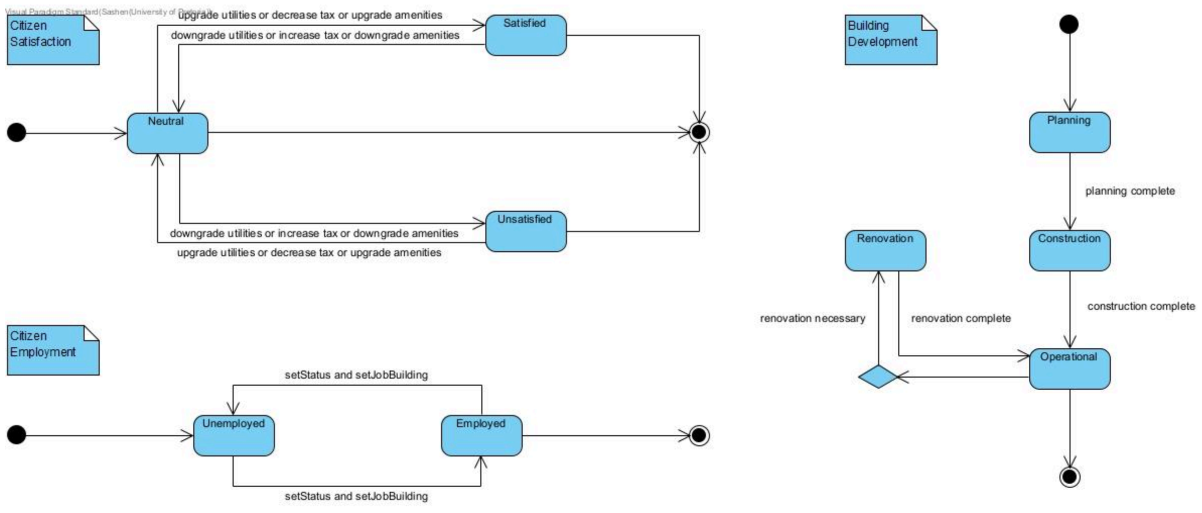
DECORATOR
Garden
+Garden(r : Residential*)
+consumeWater() : void
Gym
+Gym(r : Residential*)
+consumePower() : void
Internet
+Internet(r : Residential*)
+consumePower() : void
Pool
+Pool(r : Residential*)
+consumeWater() : void
Amenity
-amenity : Residential*
+Amenity(r : Residential*)
+consumeResources() : bool
Residential
-residents : vector<Citizen*>
+Residential(l : string, c : int)
+addResident(c : Citizen*) : void
+removeResident(c : Citizen*) : void
+getResidents() : vector<Citizen*>
+getNumResidents() : int
+consumeResources() : bool
+increaseWaterConsumption(x : int) : void
+increaseEnergyConsumption(x : int) : void
Estate
-buildings : vector<Residential*>
+Estate(l : string, c : int)
+addBuilding(r : Residential*) : void
+removeBuilding(r : Residential*) : void
+build() : Building*
+getType() : string
IPOSITE
Building
-location : string
-capacity : int
-state : BuildingState*
+Building(l : string, c : int)
+getState() : string
+progress() : void
+getCapacity() : int
+isFull() : bool
+getType() : string
+build() : Building*
+consumeResources() : bool
BuildingState
+BuildingState()
+update() : BuildingState*
+getStatus() : string
+Planning()
+update() : BuildingState*
+getStatus() : string
Construction
+Construction()
+update() : BuildingState*
+getStatus() : string
Operational
+Operational()
+update() : BuildingState*
+getStatus() : string
Renovation
+Renovation()
+update() : BuildingState*
+getStatus() : string
Commercial
-employees : vector<Citizen*>
+Commercial(l : string, c : int)
+addEmployee(c : Citizen*) : void
+removeEmployee(c : Citizen*) : void
+getNumEmployees() : int
+consumeResources() : bool
Landmark
-employees : vector<Citizen*>
+Landmark(l : string, c : int)
+addEmployee(c : Citizen*) : void
+removeEmployee(c : Citizen*) : void
+getNumEmployees() : int
+consumeResources() : bool
Industrial
-employees : vector<Citizen*>
+Industrial(l : string, c : int)
+addEmployee(c : Citizen*) : void
+removeEmployee(c : Citizen*) : void
+getNumEmployees() : int
+consumeResources() : bool
PROTOTYPE
House
+House(l : string, c : int)
+build() : Building*
+getType() : string
Apartment
+Apartment(l : string, c : int)
+build() : Building*
+getType() : string
TownHouse
+TownHouse(l : string, c : int)
+build() : Building*
+getType() : string
Mall
+Mall(l : string, c : int)
+build() : Building*
+getType() : string
Office
+Office(l : string, c : int)
+build() : Building*
+getType() : string
Monument
+Monument(l : string, c : int)
+build() : Building*
+getType() : string
Park
+Park(l : string, c : int)
+build() : Building*
+getType() : string
CulturalCentre
+CulturalCentre(l : string, c : int)
+build() : Building*
+getType() : string
Factory
+Factory(l : string, c : int)
+build() : Building*
+getType() : string
Plant
+Plant(l : string, c : int)
+build() : Building*
+generate() : void
Warehouse
+Warehouse(l : string, c : int)
+build() : Building*
+getType() : string

Each BuildingState subclass can be cloned to produce identical copies. This is beneficial when multiple buildings need to share similar states with the same initial parameters, enabling quick duplication.

The build() method creates new instances by copying the prototype building object. This approach reduces the overhead of setting up buildings from scratch, particularly when many buildings of the same type are needed.

For example, if you need multiple House objects with the same layout and configuration, the build() function can quickly replicate the prototype House instance.

State Design Pattern


Visual Paradigm Standard(Sashen(University of Pretoria))
Citizen Satisfaction
upgrade utilities or decrease tax or upgrade amenities
downgrade utilities or increase tax or downgrade amenities
Satisfied
Neutral
Unsatisfied
downgrade utilities or increase tax or downgrade amenities
upgrade utilities or decrease tax or upgrade amenities
Citizen Employment
setStatus and setJobBuilding
Unemployed
Employed
setStatus and setJobBuilding
Building Development
Planning
planning complete
Construction
construction complete
Renovation
renovation necessary
renovation complete
Operational

1.Citizen Satisfaction

States: The citizen satisfaction model consists of three primary states: Neutral, Satisfied, and Unsatisfied.

Transitions:

Moving from Neutral to Satisfied occurs when utilities are upgraded, taxes are decreased, or amenities are enhanced.
Shifting from Neutral to Unsatisfied happens when utilities are downgraded, taxes are increased, or amenities are reduced.
Citizens can return to a Neutral state from either Satisfied or Unsatisfied if conditions change back (such as reversing tax adjustments or utility modifications).

2. Citizen Employment

States: Employment has two states: Unemployed and Employed.

Transitions: The state of a citizen changes based on functions like hiring or job loss. The employment state is also linked to building status, influencing whether job roles are available.

3. Building Development

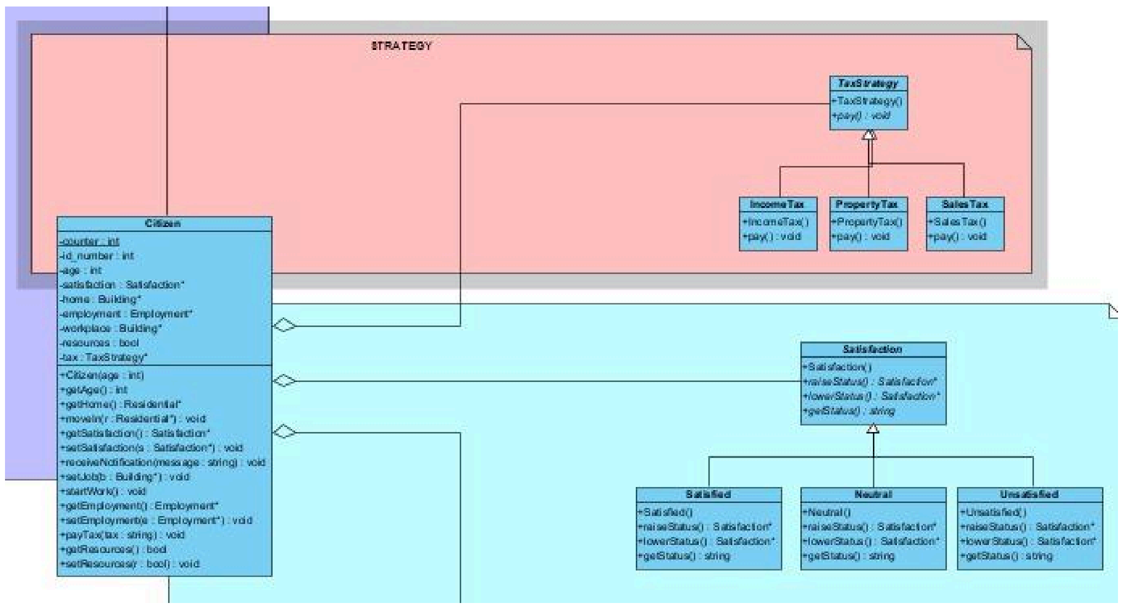
Stages: The building development process has several stages: Planning, Construction, Renovation, and Operational.

Flow: The process begins at Planning, then progresses to Construction.

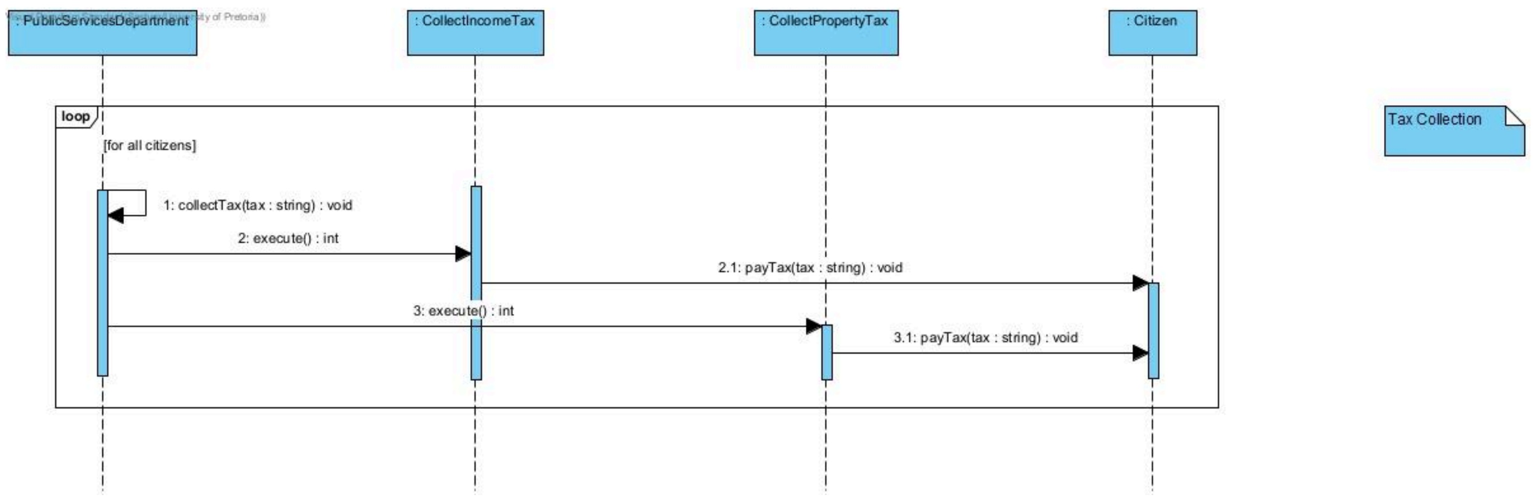
Once construction is complete, the building becomes Operational.

If renovations are needed, the building enters the Renovation stage before returning to the Operational state.

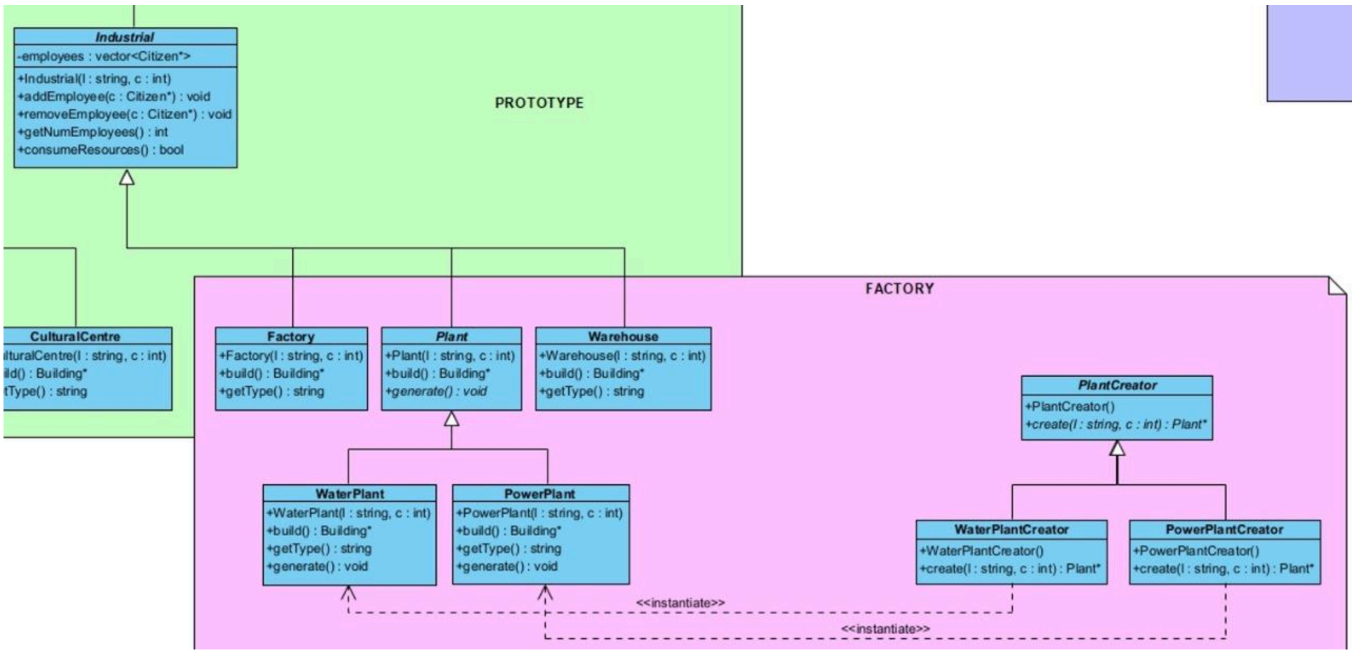
STRATEGY


STRATEGY
TaxStrategy
+TaxStrategy()
+pay() : void
IncomeTax
+IncomeTax()
+pay() : void
PropertyTax
+PropertyTax()
+pay() : void
SalesTax
+SalesTax()
+pay() : void
Citizen
-counter : int
-id_number : int
-age : int
-satisfaction : Satisfaction*
-home : Building*
-employment : Employment*
-workplace : Building*
-resources : bool
-tax : TaxStrategy*
+Citizen(age : int)
+getAge() : int
+getHome() : Residential*
+moveIn(r : Residential*) : void
+getSatisfaction() : Satisfaction*
+setSatisfaction(s : Satisfaction*) : void
+receiveNotification(message : string) : void
+setJob(b : Building*) : void
+startWork() : void
+getEmployment() : Employment*
+setEmployment(e : Employment*) : void
+payTax(tax : string) : void
+getResources() : bool
+setResources(r : bool) : void
Satisfaction
+Satisfaction()
+raiseStatus() : Satisfaction*
+lowerStatus() : Satisfaction*
+getStatus() : string
Satisfied
+Satisfied()
+raiseStatus() : Satisfaction*
+lowerStatus() : Satisfaction*
+getStatus() : string
Neutral
+Neutral()
+raiseStatus() : Satisfaction*
+lowerStatus() : Satisfaction*
+getStatus() : string
Unsatisfied
+Unsatisfied()
+raiseStatus() : Satisfaction*
+lowerStatus() : Satisfaction*
+getStatus() : string

The strategy design pattern was used to implement how citizens would pay taxes based on the type of tax that the user/government would like to collect. This design makes the system flexible as it is easy to implement various versions of an algorithm.
It simplifies coding as there is no need for multiple conditional statements as would be required if the strategy pattern was not used. By encapsulating each tax type (Income,Property and Sales) into their own separate classes, it is easier to make modifications to the code without needing to edit the core of the system. It is also easier to allow the addition of more taxes that could be introduced in the future.


: PublicServicesDepartment
: CollectIncomeTax
: CollectPropertyTax
: Citizen
Tax Collection
loop
[for all citizens]
1: collectTax(tax : string) : void
2: execute() : int
2.1: payTax(tax : string) : void
3: execute() : int
3.1: payTax(tax : string) : void

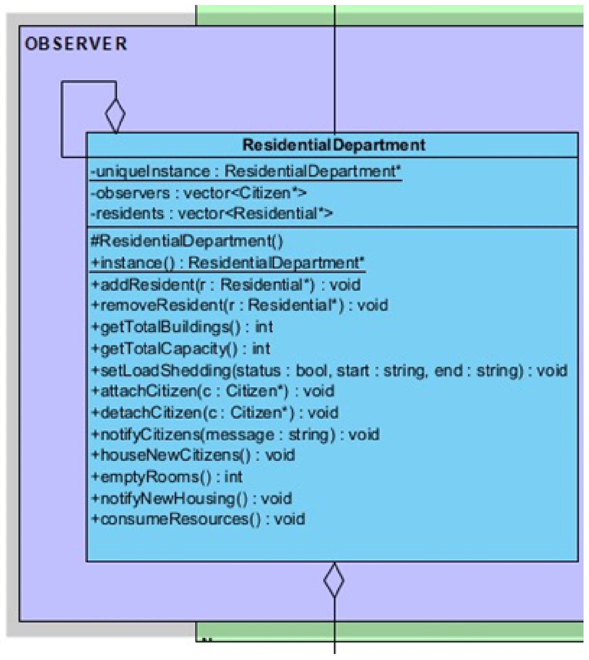
Factory Design Pattern

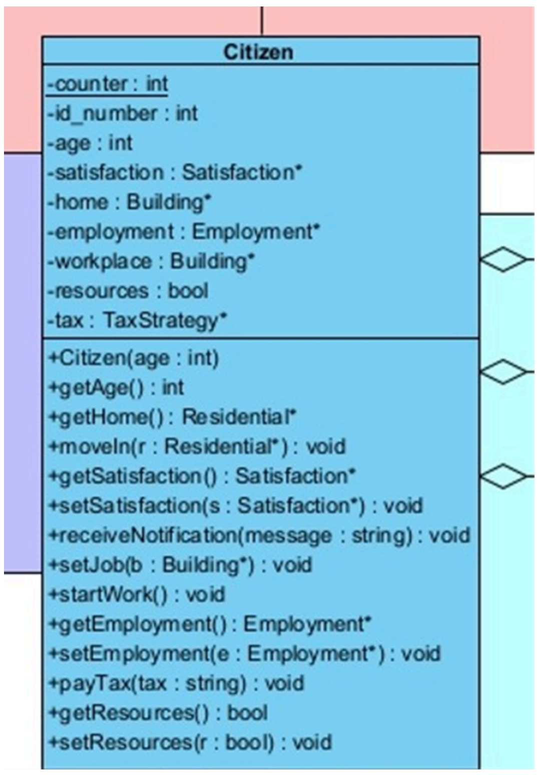


The Factory Method pattern is used to create different types of concrete plant classes, specifically WaterPlant and PowerPlant. A generic factory class, PlantCreator, defines the interface for creating a plant without specifying the exact type. Then, concrete creator classes, WaterPlantCreator and PowerPlantCreator, implement this interface to instantiate their specific plant types. For instance, WaterPlantCreator creates instances of WaterPlant, while PowerPlantCreator creates instances of PowerPlant.

This design pattern was chosen because it encapsulates the object creation process, making the code more modular and easier to extend. Instead of modifying the factory or product classes when a new plant type is added, an additional concrete creator class and concrete product class can simply be introduced. This makes the system highly flexible, supporting future growth without disrupting the existing architecture. The Factory Method was used for this functionality because it offers a structured and scalable solution for managing the creation of different plant types.

Observer Design Pattern


OBSERVER
ResidentialDepartment
-uniqueInstance : ResidentialDepartment*
-observers : vector<Citizen*>
-residents : vector<Residential*>
#ResidentialDepartment()
+instance() : ResidentialDepartment*
+addResident(r : Residential*) : void
+removeResident(r : Residential*) : void
+getTotalBuildings() : int
+getTotalCapacity() : int
+setLoadShedding(status : bool, start : string, end : string) : void
+attachCitizen(c : Citizen*) : void
+detachCitizen(c : Citizen*) : void
+notifyCitizens(message : string) : void
+houseNewCitizens() : void
+emptyRooms() : int
+notifyNewHousing() : void
+consumeResources() : void

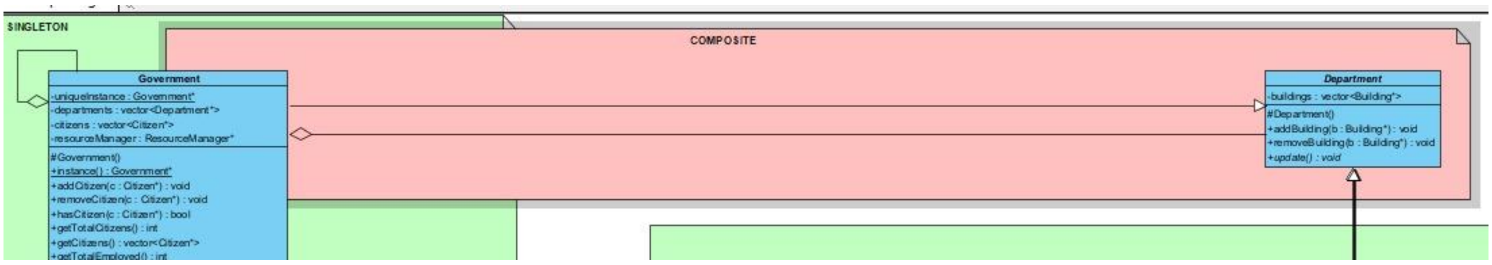

Citizen
-counter : int
-id_number : int
-age : int
-satisfaction : Satisfaction*
-home : Building*
-employment : Employment*
-workplace : Building*
-resources : bool
-tax : TaxStrategy*
+Citizen(age : int)
+getAge() : int
+getHome() : Residential*
+moveIn(r : Residential*) : void
+getSatisfaction() : Satisfaction*
+setSatisfaction(s : Satisfaction*) : void
+receiveNotification(message : string) : void
+setJob(b : Building*) : void
+startWork() : void
+getEmployment() : Employment*
+setEmployment(e : Employment*) : void
+payTax(tax : string) : void
+getResources() : bool
+setResources(r : bool) : void

The Observer pattern is used to manage notifications to citizens regarding events such as load-shedding. The ResidentialDepartment class acts as the subject, responsible for notifying its observers (in this case, citizens) about any changes. Citizens who are interested in receiving notifications register themselves with the ResidentialDepartment by using the attachCitizen method, while those who no longer wish to receive notifications can detach themselves using detachCitizen. When a load-shedding event is triggered, the setLoadShedding method changes the load-shedding status and, if there is an update, it calls the notifyCitizens method. This method iterates over the registered citizens and sends a notification message to each one, ensuring they are informed of the event.

The Observer pattern was chosen here because it decouples the ResidentialDepartment class from the specific implementation of citizen notification, making it easy to manage a dynamic list of observers. If new types of citizens or notification methods are introduced, the ResidentialDepartment can continue notifying all observers without requiring changes to its internal structure. This pattern provides a flexible way to handle one-to-many relationships and allows for changes in the state of the ResidentialDepartment to be reflected across multiple objects that depend on it. As a result, the Observer pattern enhances modularity, making the codebase more maintainable and scalable.

Composite Design Pattern

1.Department Composite

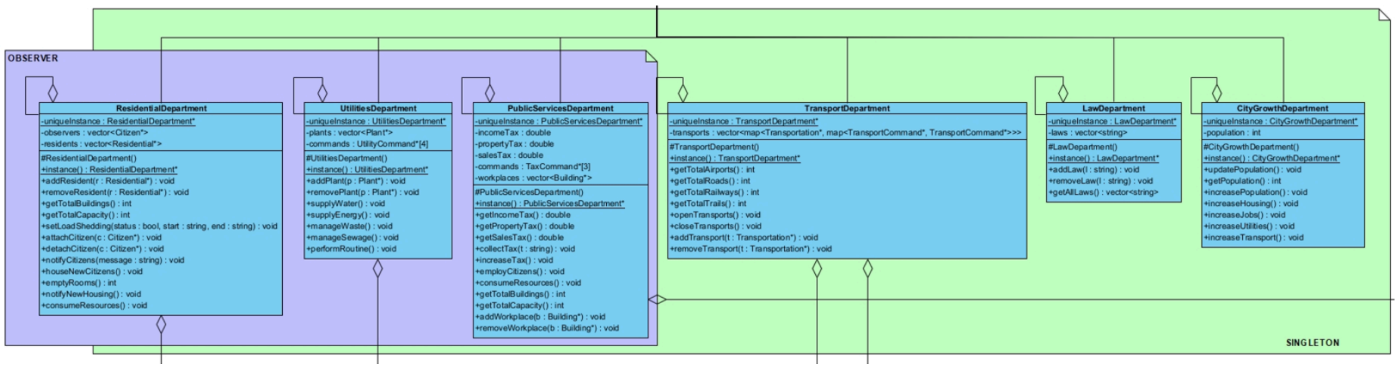

SINGLETON
COMPOSITE
Government
-uniqueInstance : Government*
-departments : vector<Department*>
-citizens : vector<Citizen*>
-resourceManager : ResourceManager*
#Government()
+instance() : Government*
+addCitizen(c : Citizen*) : void
+removeCitizen(c : Citizen*) : void
+hasCitizen(c : Citizen*) : bool
+getTotalCitizens() : int
+getCitizens() : vector<Citizen*>
+getTotalEmployed() : int
Department
-buildings : vector<Building*>
#Department()
+addBuilding(b : Building*) : void
+removeBuilding(b : Building*) : void
+update() : void

The Government class serves as the composite that can manage multiple Department objects. It extends the Department class, which allows it to function as a department itself, but with added capabilities for managing a collection of other departments.

The Leaf components are the individual, specialised departments that inherit from the Department class. Each department focuses on a specific area of city management. Examples include:

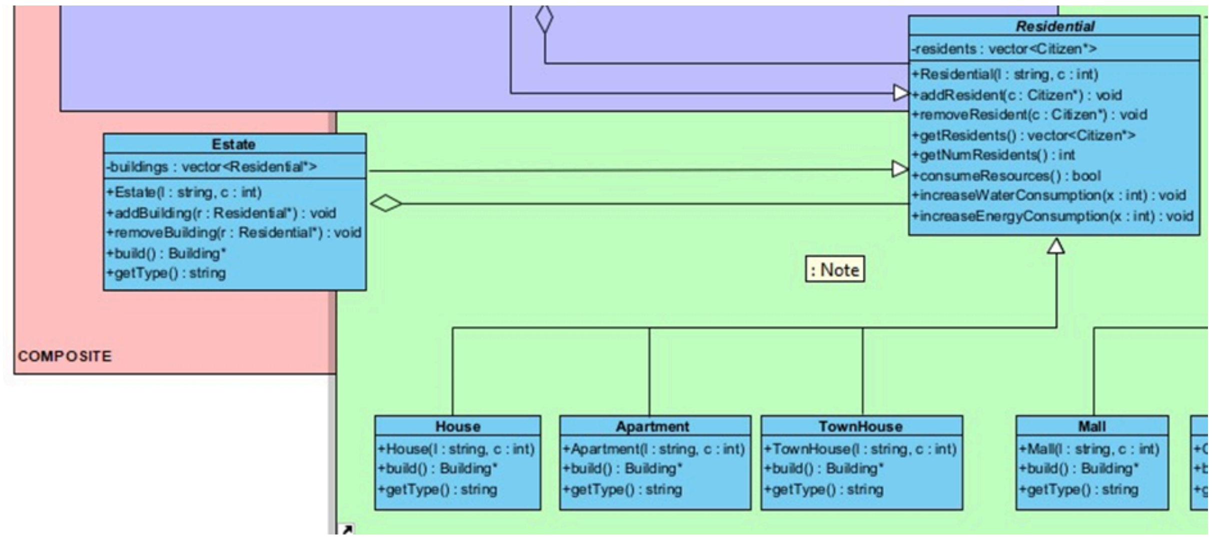
- **ResidentialDepartment**: Manages residential areas and buildings.
- **TransportDepartment**: Handles transportation and logistics.
- **UtilitiesDepartment**: Oversees utility services like water, energy, and waste management.
- **PublicServicesDepartment**: Manages public services, including tax collection
- **LawDepartment**: Manages legal and law enforcement operations.
- **CityGrowthDepartment**: Focuses on urban development and city planning.

The Department class serves as the base class for all specific department types within the government. It defines core functionality that all departments share, such as managing buildings.The Department class ensures that all specific department classes can uniformly manage their buildings.


OBSERVER
SINGLETON
ResidentialDepartment
-uniqueInstance : ResidentialDepartment*
-observers : vector<Citizen*>
-residents : vector<Residential*>
#ResidentialDepartment()
+instance() : ResidentialDepartment*
+addResident(r : Residential*) : void
+removeResident(r : Residential*) : void
+getTotalBuildings() : int
+getTotalCapacity() : int
+setLoadShedding(status : bool, start : string, end : string) : void
+attachCitizen(c : Citizen*) : void
+detachCitizen(c : Citizen*) : void
+notifyCitizens(message : string) : void
+houseNewCitizens() : void
+emptyRooms() : int
+notifyNewHousing() : void
+consumeResources() : void
UtilitiesDepartment
-uniqueInstance : UtilitiesDepartment*
-plants : vector<Plant*>
-commands : UtilityCommand*[4]
#UtilitiesDepartment()
+instance() : UtilitiesDepartment*
+addPlant(p : Plant*) : void
+removePlant(p : Plant*) : void
+supplyWater() : void
+supplyEnergy() : void
+manageWaste() : void
+manageSewage() : void
+performRoutine() : void
PublicServicesDepartment
-uniqueInstance : PublicServicesDepartment*
-incomeTax : double
-propertyTax : double
-salesTax : double
-commands : TaxCommand*[3]
-workplaces : vector<Building*>
#PublicServicesDepartment()
+instance() : PublicServicesDepartment*
+getIncomeTax() : double
+getPropertyTax() : double
+getSalesTax() : double
+collectTax(t : string) : void
+increaseTax() : void
+employCitizens() : void
+consumeResources() : void
+getTotalBuildings() : int
+getTotalCapacity() : int
+addWorkplace(b : Building*) : void
+removeWorkplace(b : Building*) : void
TransportDepartment
-uniqueInstance : TransportDepartment*
-transports : vector<map<Transportation*, map<TransportCommand*, TransportCommand*>>>
#TransportDepartment()
+instance() : TransportDepartment*
+getTotalAirports() : int
+getTotalRoads() : int
+getTotalRailways() : int
+getTotalTrails() : int
+openTransports() : void
+closeTransports() : void
+addTransport(t : Transportation*) : void
+removeTransport(t : Transportation*) : void
LawDepartment
-uniqueInstance : LawDepartment*
-laws : vector<string>
#LawDepartment()
+instance() : LawDepartment*
+addLaw(l : string) : void
+removeLaw(l : string) : void
+getAllLaws() : vector<string>
CityGrowthDepartment
-uniqueInstance : CityGrowthDepartment*
-population : int
#CityGrowthDepartment()
+instance() : CityGrowthDepartment*
+updatePopulation() : void
+getPopulation() : int
+increasePopulation() : void
+increaseHousing() : void
+increaseJobs() : void
+increaseUtilities() : void
+increaseTransport() : void

The Different Departments

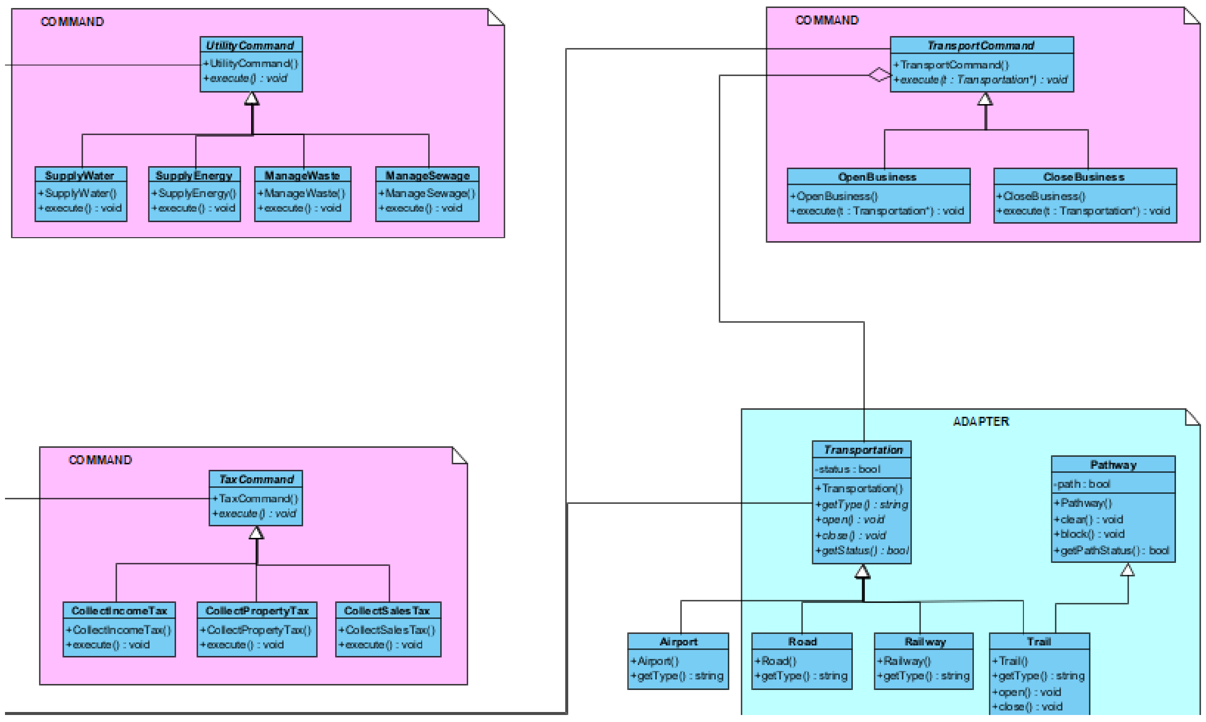
2. Residential Composite


Residential
-residents : vector<Citizen*>
+Residential(l : string, c : int)
+addResident(c : Citizen*) : void
+removeResident(c : Citizen*) : void
+getResidents() : vector<Citizen*>
+getNumResidents() : int
+consumeResources() : bool
+increaseWaterConsumption(x : int) : void
+increaseEnergyConsumption(x : int) : void
Estate
-buildings : vector<Residential*>
+Estate(l : string, c : int)
+addBuilding(r : Residential*) : void
+removeBuilding(r : Residential*) : void
+build() : Building*
+getType() : string
: Note
COMPOSITE
House
+House(l : string, c : int)
+build() : Building*
+getType() : string
Apartment
+Apartment(l : string, c : int)
+build() : Building*
+getType() : string
TownHouse
+TownHouse(l : string, c : int)
+build() : Building*
+getType() : string
Mall
+Mall(l : string, c : int)
+build() : Building*
+getType() : string

The Composite pattern is used to manage different types of residential buildings within a single unified structure. The Residential class serves as the Component interface, defining common operations for both individual buildings (Leaf nodes) and groups of buildings (Composite nodes). The specific types of residential buildings—such as House, Apartment, and TownHouse—are implemented as Leaf nodes, meaning they represent individual, non-divisible units. The Estate class functions as the Composite, aggregating multiple Residential buildings and enabling them to be treated as a single entity. Through the Composite pattern, Estate can hold both individual buildings and groups of buildings, allowing operations like adding, removing, or managing buildings to be applied to both single units and collections.

The decision to use the Composite pattern here allows for flexibility in handling hierarchical building structures while maintaining consistency in operations. By using this method, an Estate can have a variety of residential buildings and sub-groups, including other Estates, which enables flexible and adaptable handling of intricate building layouts. The pattern ensures that all residential entities, whether individual or grouped, can be treated uniformly. This design makes it easy to expand with additional types of residential buildings or estates, supporting maintenance and scalability within the codebase.

Adapter Design Pattern:



The Adapter Pattern is utilised in our scenario to integrate traditional pathways with modern transport systems, making them compatible with new forms of eco-friendly transportation like bicycles. Our previous infrastructure consists of pathways that can either be cleared or blocked. The modern transportation system requires trails that can either be opened or closed and can support multiple transport types, including pedestrians and cyclists. When the modern transport system requests a pathway that supports bicycles, the Trail adapts the existing traditional Pathway, adding the necessary features for ease of use.