Task 4: Report

The 7

Research Brief

Managing a city is a complex process, needing to take various factors into consideration to provide for the citizens. The needs of the citizens play a crucial role in urban development, building infrastructure, providing transportation, and providing goods and services. With all this comes the need for managing everything. Managing finances, collecting taxes, ensuring everyone is following the rules, and making sure resources are allocated efficiently. To ensure economic growth will require efficient management of the above mentioned and addressing the goals of the city.

Urban development includes all the buildings and infrastructure of the city. This is to accommodate the citizens. Expansion is required as the number of citizens increases. It is also important to accommodate the social needs of citizens such as having parks, shops, entertainment areas, etc. For our project we made sure to include all of these. We made sure to have various plants, to manage the sewage, water, and power of the city. We included factories for our resources and warehouses to store them. We keep track of all the buildings, such as their coordinates, resource consumption, the state of the buildings, and more.

City management is how the city is managed. Example, providing services, managing finances, and providing safety. This will determine the quality of life and is therefore important to manage the city efficiently. We made sure to provide resources such as water, power, and waste management. We keep track of housing space available, jobs, and income of citizens. We also make sure to manage the taxes of each citizen so that we can provide services, such as police stations and other infrastructure, and maintain the city.

The role of the components is very important as the various components are dependent on one another. For example, the urban development components are reliant on city management. And city management will not be possible without urban development. Taxes need to be collected and resources need to be managed to expand the city and provide the citizens with their needs. We therefore keep track and manage all the components in our city. The urban development will be dependent on how the population increases and decreases. We have a government to oversee all the taxes of the citizens and to keep order in the city.

Through this research we learned that it is important to incorporate the various roles of urban development, city management, and the components to have a successful, well-functioning city. We made sure to understand these components well, chose our design patterns carefully, and implemented the code to have a city that is up and running, catering for all our citizens' needs.

References

Almssad, A. A. (2019, 10 22). *City Phenomenon between Urban Structure and Composition*. Retrieved from https://www.intechopen.com/chapters/70567

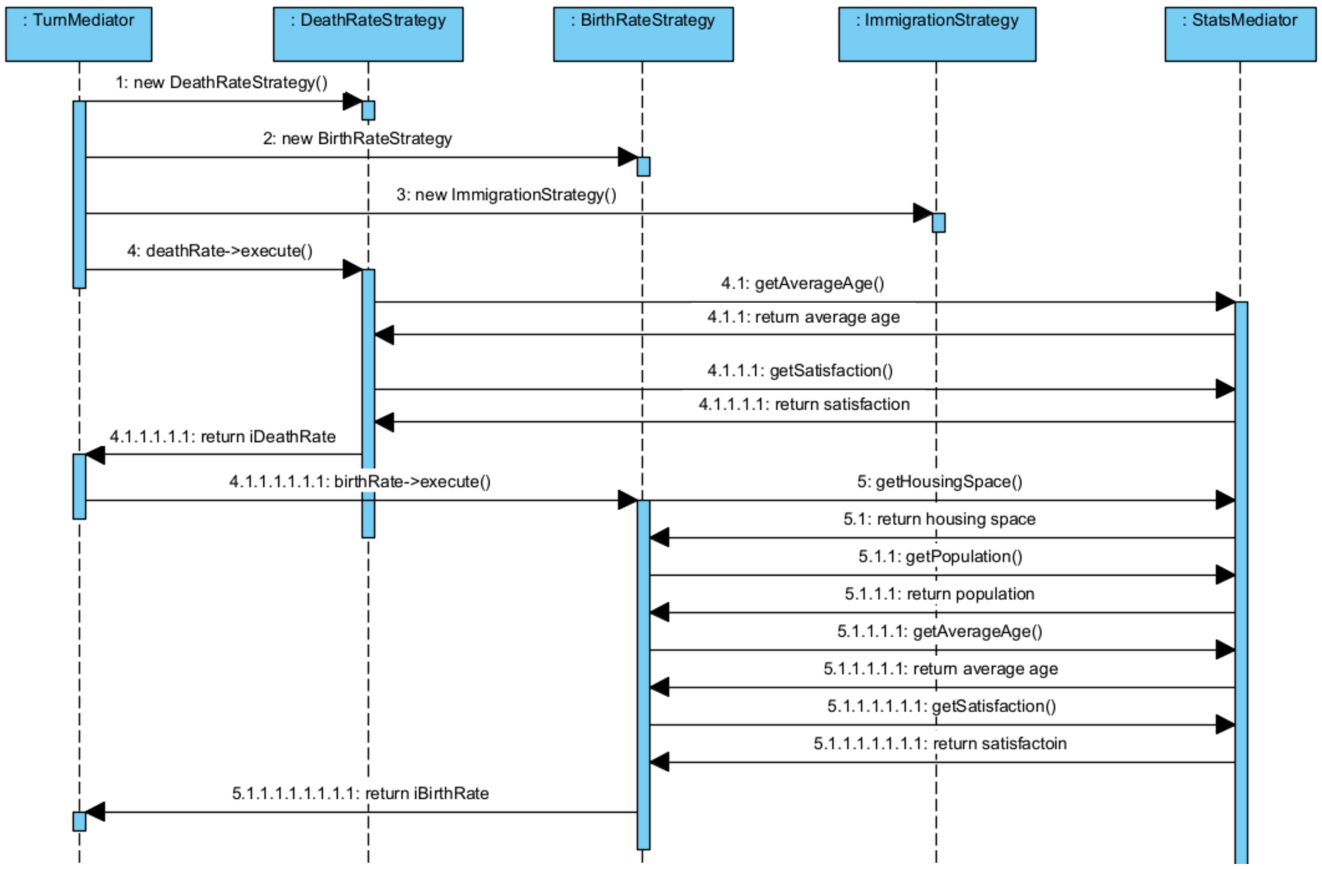
City Management. (n.d.). Retrieved from Basic Knowledge 101: https://www.basicknowledge101.com/subjects/citymanagement.html#goverment

Sailus, M. (2024, 09 23). *7 Types Of Urban Planning Concepts Explained*. Retrieved from ClearPoint Strategy: https://www.clearpointstrategy.com/blog/types-of-urban-planning

Design Pattern Application Report

Strategy Pattern:

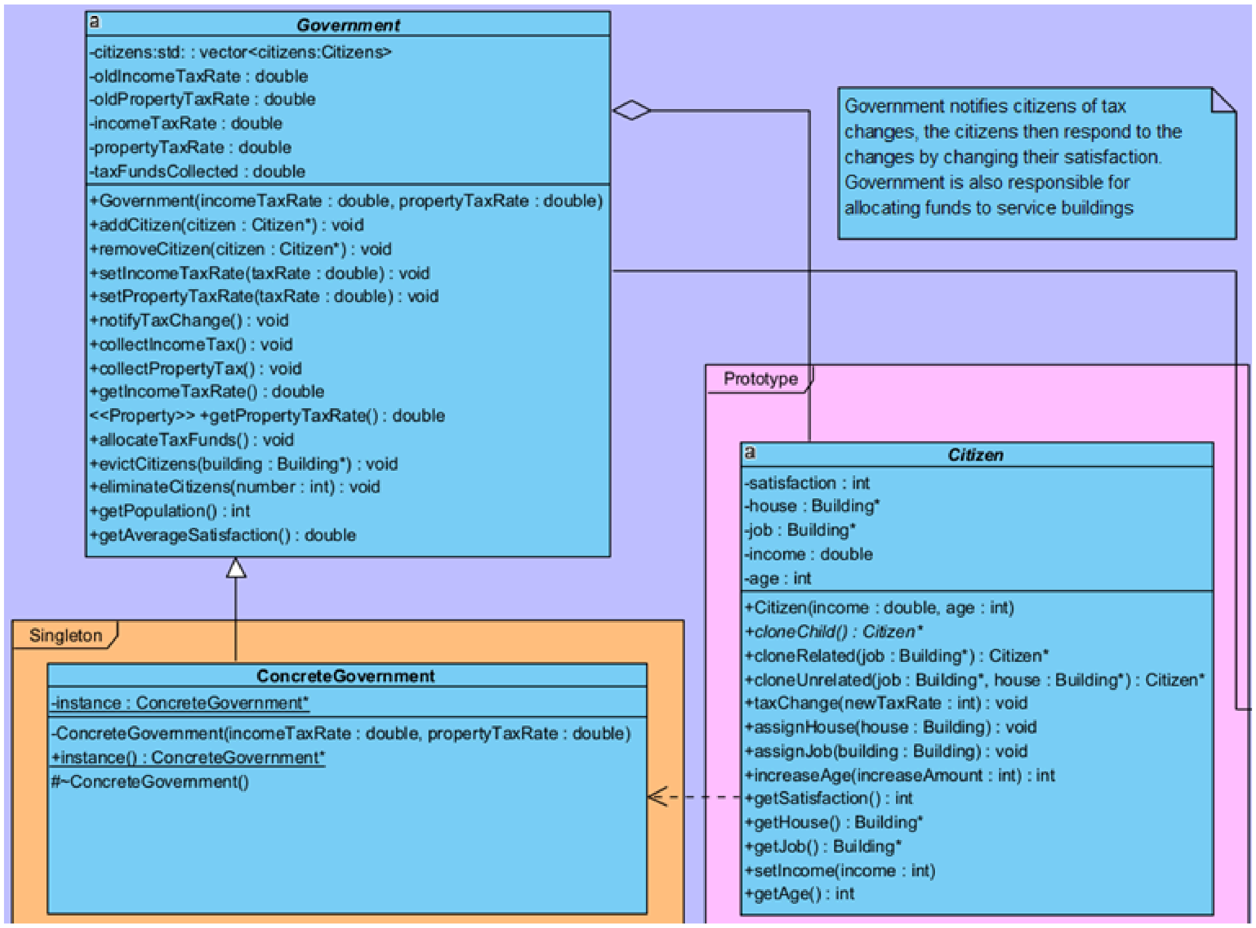
The strategy pattern is used to define a family of strategies to determine population changes in the city. The 'PopulationStrategy' class acts as the strategy, providing an abstract definition of the execute method. The concrete strategies used are: BirthRateStrategy, DeathRateStrategy, and ImmigrationStrategy. In each of these an algorithm is used to calculate the respective amount of people by which the population will change. We determine the algorithm using different statistics, such as, average age, satisfaction of the population, immigration limit, etc. These stats are provided by the mediator pattern (StatsMediator). The TurnMediator acts as the context of the strategy class, executing the execute() method of the relevant strategies. Below is a sequence diagram to illustrate the strategy pattern better.



: TurnMediator
: DeathRateStrategy
: BirthRateStrategy
: ImmigrationStrategy
: StatsMediator
1: new DeathRateStrategy()
2: new BirthRateStrategy
3: new ImmigrationStrategy()
4: deathRate->execute()
4.1: getAverageAge()
4.1.1: return average age
4.1.1.1: getSatisfaction()
4.1.1.1.1: return satisfaction
4.1.1.1.1.1: return iDeathRate
4.1.1.1.1.1.1: birthRate->execute()
5: getHousingSpace()
5.1: return housing space
5.1.1: getPopulation()
5.1.1.1: return population
5.1.1.1.1: getAverageAge()
5.1.1.1.1.1: return average age
5.1.1.1.1.1.1: getSatisfaction()
5.1.1.1.1.1.1.1: return satisfactoin
5.1.1.1.1.1.1.1.1: return iBirthRate

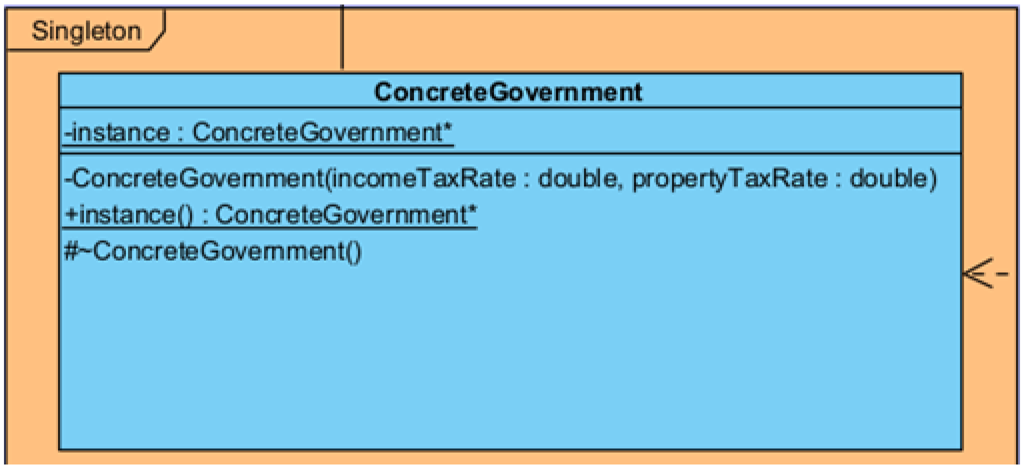
<u>Observer Pattern:</u>

The Observer Pattern is a behavioral design pattern that establishes a one-to-many relationship between objects, allowing one object (the subject) to notify multiple observers of any changes with the taxes. Such as the Income tax or the Property Tax. When notified by the government by the tax change the observers(citizens) will then alter their satisfaction with the change.


Government
-citizens:std: : vector<citizens:Citizens>
-oldIncomeTaxRate : double
-oldPropertyTaxRate : double
-incomeTaxRate : double
-propertyTaxRate : double
-taxFundsCollected : double
+Government(incomeTaxRate : double, propertyTaxRate : double)
+addCitizen(citizen : Citizen*) : void
+removeCitizen(citizen : Citizen*) : void
+setIncomeTaxRate(taxRate : double) : void
+setPropertyTaxRate(taxRate : double) : void
+notifyTaxChange() : void
+collectIncomeTax() : void
+collectPropertyTax() : void
+getIncomeTaxRate() : double
<<Property>> +getPropertyTaxRate() : double
+allocateTaxFunds() : void
+evictCitizens(building : Building*) : void
+eliminateCitizens(number : int) : void
+getPopulation() : int
+getAverageSatisfaction() : double
Government notifies citizens of tax changes, the citizens then respond to the changes by changing their satisfaction. Government is also responsible for allocating funds to service buildings
Prototype
Citizen
-satisfaction : int
-house : Building*
-job : Building*
-income : double
-age : int
+Citizen(income : double, age : int)
+cloneChild() : Citizen*
+cloneRelated(job : Building*) : Citizen*
+cloneUnrelated(job : Building*, house : Building*) : Citizen*
+taxChange(newTaxRate : int) : void
+assignHouse(house : Building) : void
+assignJob(building : Building) : void
+increaseAge(increaseAmount : int) : int
+getSatisfaction() : int
+getHouse() : Building*
+getJob() : Building*
+setIncome(income : int)
+getAge() : int
Singleton
ConcreteGovernment
-instance : ConcreteGovernment*
-ConcreteGovernment(incomeTaxRate : double, propertyTaxRate : double)
+instance() : ConcreteGovernment*
#~ConcreteGovernment()

<u>Singleton Pattern:</u>

The Singleton Pattern is a design pattern that restricts the instantiation of a class to a single instance and provides a global point of access to that instance. This pattern is useful in where exactly one object is needed such as one Government in our program.


Singleton
ConcreteGovernment
-instance : ConcreteGovernment*
-ConcreteGovernment(incomeTaxRate : double, propertyTaxRate : double)
+instance() : ConcreteGovernment*
#~ConcreteGovernment()

Private Static Pointer:

- static ConcreteGovernment *instancePtr;: This static member variable holds the single instance of ConcreteGovernment.

Private Constructor:

- The constructor ConcreteGovernment(double incomeTaxRate, double propertyTaxRate); is private, preventing other parts of the program from creating new instances of ConcreteGovernment directly.

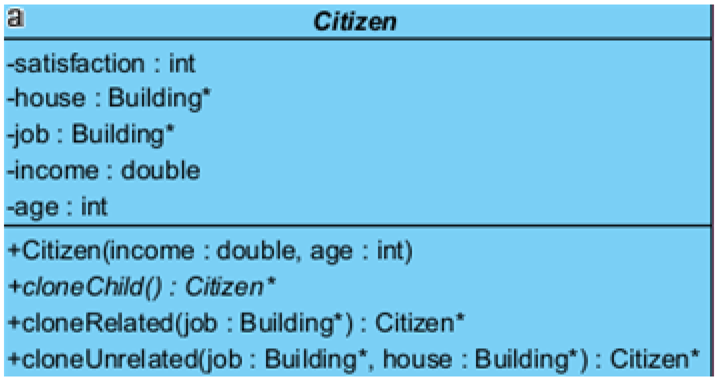
Static Instance Method:

- static ConcreteGovernment *instance();: This method provides a global point of access to the single instance. It typically checks if instancePtr is nullptr (i.e., if the instance has not been created yet) and, if so, creates it.

Prototype:

The Prototype Pattern is a creational design pattern that allows for creating new objects by copying an existing object, known as the prototype. This pattern is particularly useful when the cost of creating a new instance of an object is more expensive than copying an existing instance.

In the Citizen class we have clone functions namely cloneChild, cloneRelated, cloneUnrelated.


a
Citizen
-satisfaction : int
-house : Building*
-job : Building*
-income : double
-age : int
+Citizen(income : double, age : int)
+cloneChild() : Citizen*
+cloneRelated(job : Building*) : Citizen*
+cloneUnrelated(job : Building*, house : Building*) : Citizen*

cloneChild clones the citizen. They will have the same house but the job and income as well as the age will all be zero or nullptr.

cloneRelated clones the citizen with the same house but with a different job.

cloneUnrelated will have different houses and jobs.

Command Pattern:

The command pattern is a behavioral design pattern that is used to encapsulate the requests for various city operations, providing a flexible way to execute commands such as building or destruction without changing the client code.

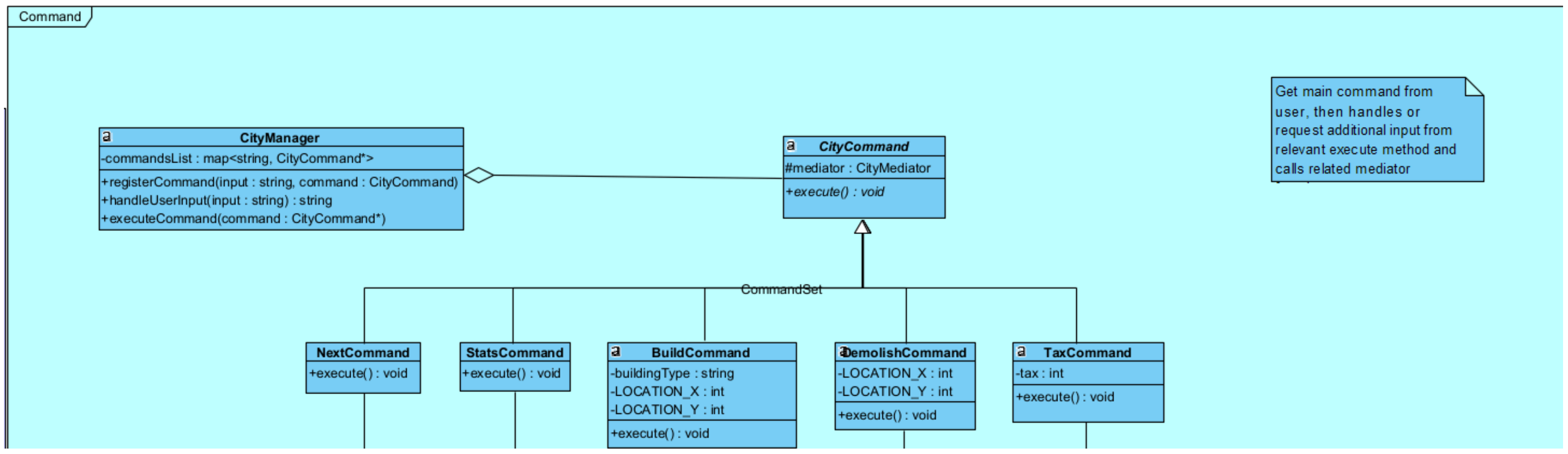
The participants for command design pattern include:

CityCommand: An abstract base class with a pure virtual execute() method. It serves as the interface for all concrete command classes.

ConcreteCommands:

- **BuildCommand:** This class implement the execute() function defined in CityCommand
- **DemolishCommand:** This class handles the demolition of a building at specific X and Y coordinates
- **TaxCommand:** This command sets and changes the tax rate by interacting with the Tax class, ensuring control within the city.
- **NextCommand:** Proceeds to the next phase in the city management workflow.
- **StatsCommand:** Retrieves and displays city statistics when the execute() method is called.

CityManager: Acts as the invoker that stores and executes the commands. CityManager holds a list of CityCommand objects and calls the execute() method on each object when necessary.



The command pattern promotes flexibility and scalability when handling different types of city operations. Overall the Command design pattern supports an easily extendable design for the city simulator.

Abstract Factory:

The Abstract Factory Pattern is a creational design pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes. In our city management system, this pattern is implemented to create different types of buildings and services that are required for urban development.

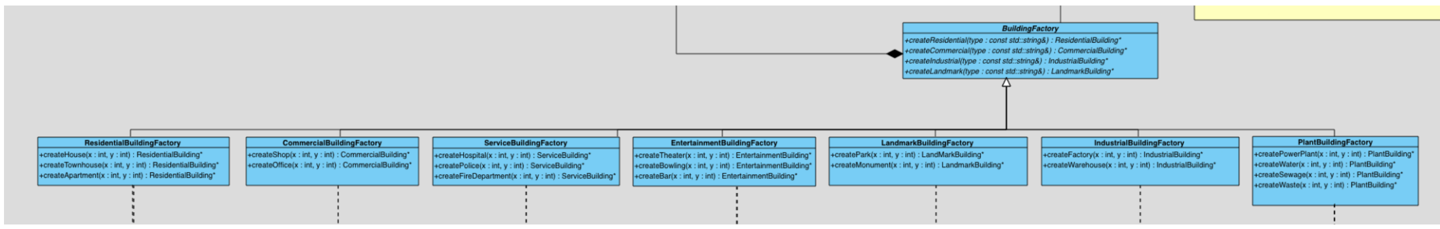
Purpose

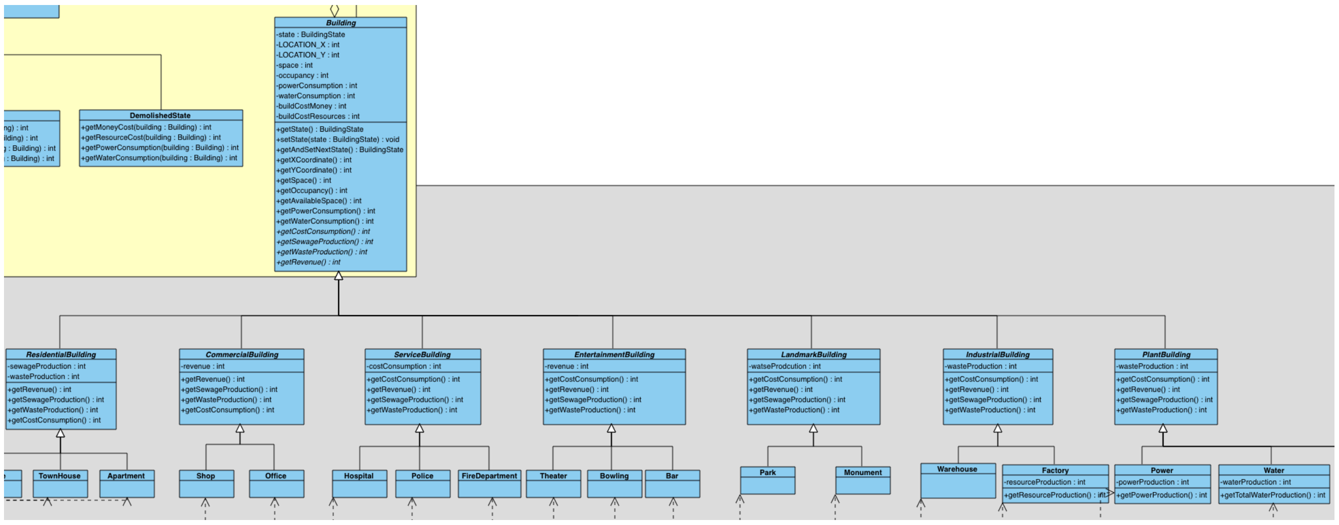
The Abstract Factory serves to encapsulate the creation of various components of the city, such as residential buildings, commercial buildings, industrial facilities, and essential services (e.g., police stations, hospitals). This approach allows us to ensure that all created components are compatible with each other and adhere to the requirements of the city's infrastructure.

Implementation:

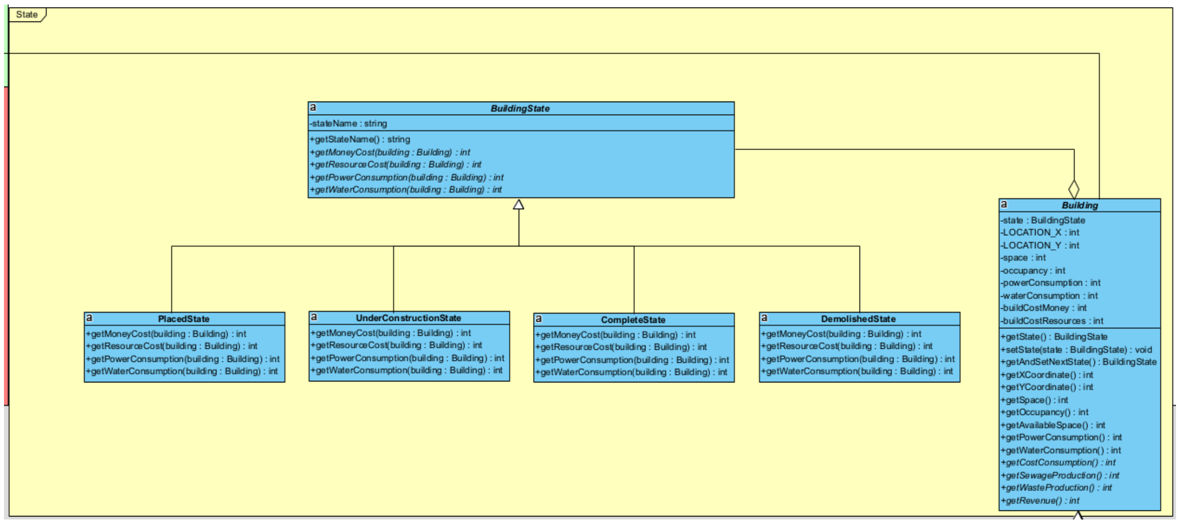
1. **Abstract Factory Interface:** We define an interface, BuildingFactory, with methods for creating various types of buildings and services.

2. **<u>Concrete Factories:</u>** We implement concrete factories that extend the BuildingFactory interface, such as ResidentialBuildingFactory, CommercialBuildingFactory, ServiceBuildingFactory, EntertainmentBuildingFactory, LandmarkBuildingFactory, IndustrialBuildingFactory, and PlantBuildingFactory
3. **<u>Abstract Products:</u>** We define an abstract interface, Building that defines basic attribute of a building, we then extend it to multiple other abstract classes: ResidentialBuilding, CommercialBuilding, ServiceBuilding, EntertainmentBuilding, LandmarkBuilding, IndustrialBuilding, and PlantBuilding, all these classes provide an interface for product objects.
4. **<u>Concrete Products:</u>** We extend every abstract product class into multiple concrete product classes which will be constructed by their respective concrete factory.


BuildingFactory
+createResidential(type : const std::string&) : ResidentialBuilding*
+createCommercial(type : const std::string&) : CommercialBuilding*
+createIndustrial(type : const std::string&) : IndustrialBuilding*
+createLandmark(type : const std::string&) : LandmarkBuilding*
ResidentialBuildingFactory
CommercialBuildingFactory
ServiceBuildingFactory
EntertainmentBuildingFactory
LandmarkBuildingFactory
IndustrialBuildingFactory
PlantBuildingFactory



<u>State Design Pattern:</u>


State
BuildingState
-stateName : string
+getStateName() : string
+getMoneyCost(building : Building) : int
+getResourceCost(building : Building) : int
+getPowerConsumption(building : Building) : int
+getWaterConsumption(building : Building) : int
PlacedState
+getMoneyCost(building : Building) : int
+getResourceCost(building : Building) : int
+getPowerConsumption(building : Building) : int
+getWaterConsumption(building : Building) : int
UnderConstructionState
+getMoneyCost(building : Building) : int
+getResourceCost(building : Building) : int
+getPowerConsumption(building : Building) : int
+getWaterConsumption(building : Building) : int
CompleteState
+getMoneyCost(building : Building) : int
+getResourceCost(building : Building) : int
+getPowerConsumption(building : Building) : int
+getWaterConsumption(building : Building) : int
DemolishedState
+getMoneyCost(building : Building) : int
+getResourceCost(building : Building) : int
+getPowerConsumption(building : Building) : int
+getWaterConsumption(building : Building) : int
Building
-state : BuildingState
-LOCATION_X : int
-LOCATION_Y : int
-space : int
-occupancy : int
-powerConsumption : int
-waterConsumption : int
-buildCostMoney : int
-buildCostResources : int
+getState() : BuildingState
+setState(state : BuildingState) : void
+getAndSetNextState() : BuildingState
+getXCoordinate() : int
+getYCoordinate() : int
+getSpace() : int
+getOccupancy() : int
+getAvailableSpace() : int
+getPowerConsumption() : int
+getWaterConsumption() : int
+getCostConsumption() : int
+getSewageProduction() : int
+getWasteProduction() : int
+getRevenue() : int

The State Design Pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. In the context of our city management system, this pattern is particularly useful for managing the different states of buildings and their transitions throughout the development lifecycle.

By encapsulating the various states of a building into separate classes, we can enhance the maintainability and clarity of our code while also making it easier to implement new states in the future.

Purpose

The primary purpose of the State Design Pattern in our city management system is to model the life cycle of buildings within the city, including their construction phases and eventual demolition. The states include:

Implementation

1. **Context Class (Building):** The building class acts as the context that maintains a reference to the current state. It delegates the state-specific behavior to the current state object, allowing for transitions between states without needing to modify the building class itself.
2. **State Interface (BuildingState):** An interface or abstract class defining the methods that all concrete state classes must implement. This interface ensures that all states adhere to a common protocol, allowing the building context to interact with them uniformly.
3. **Concrete State Classes:**

    - **PlacedState**: Implements the behavior for a building that has been placed but is not yet constructed. This class can handle requests to start construction.
    - **UnderConstructionState**: Manages the processes associated with building construction, such as resource consumption and transitioning to the complete state.
    - **CompleteState**: Represents the fully constructed building, allowing it to interact with occupants and other city systems.
    - **DemolishedState**: Represents the state of a building that has been demolished, restricting any further interactions.
4. **State Transition Logic:** The logic for transitioning between states is encapsulated within each state class. For instance, when a building is completed, the UnderConstructionState can transition to CompleteState, allowing the building to update its state seamlessly.

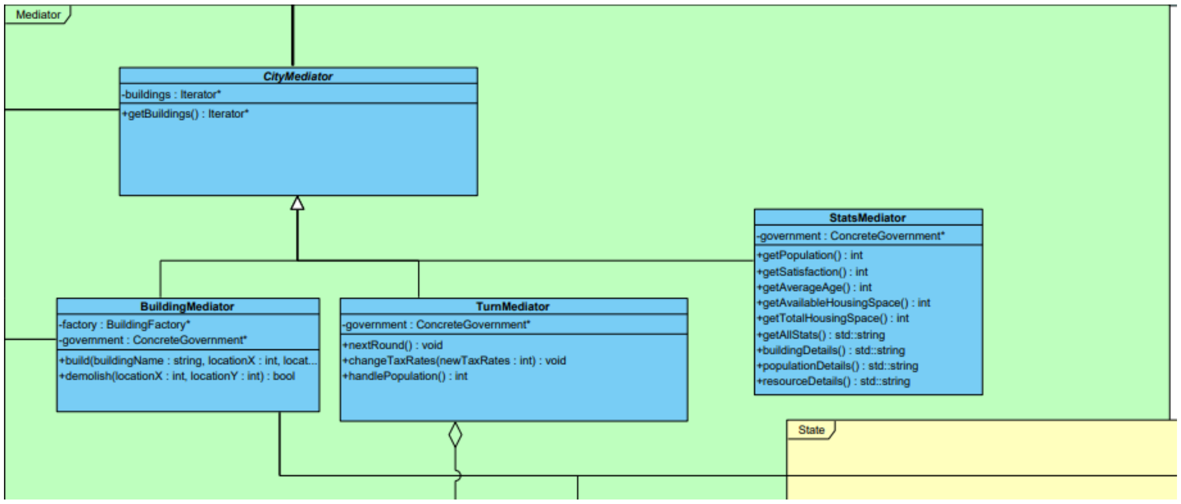
Mediator Pattern:

The Mediator enables loose coupling between objects by keeping the object from referring to each other explicitly and forces them to communicate via the mediator object. This allows for the program to vary the interactions between the objects independently.

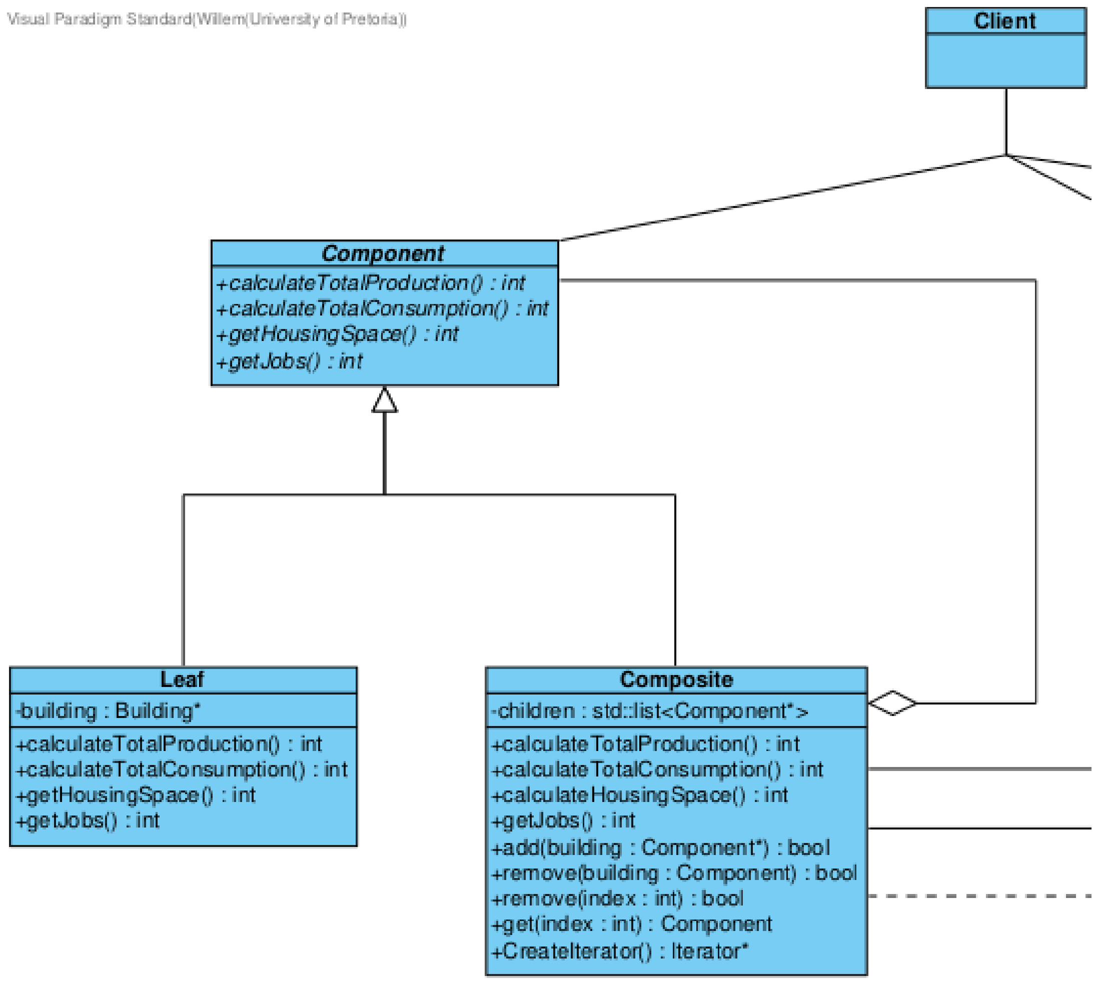
Participants for the mediator pattern in our program:

- CityMediator acts as the abstract mediator base class. It holds data such as an iterator to iterate through buildings. It also provides an interface for the derived classes.
- BuildingMediator. It is responsible for managing building objects and their respective operations. It uses the building factory classes as the colleague for specific building types such as ResidentialBuildingFactory. Buildings are also only accessible via the iterator to prevent direct modification.
- TurnMediator. Manages the progression of turns in the simulation by handling population changes, and buildings state.
- StatsMediator. Provides statistics for other patterns to use such as population size, average age, etc. It also utilizes the government to gain access to information from citizens. It also gets data from buildings by iterating over the buildings.

The mediator pattern is good as it provides centralized control and enables loose coupling. It provides complex inter-object communication and allows for the other patterns to communicate to each other. The mediator provides flexibility and makes it easy to expand as more functionality is required.


Mediator
CityMediator
-buildings : Iterator*
+getBuildings() : Iterator*
StatsMediator
-government : ConcreteGovernment*
+getPopulation() : int
+getSatisfaction() : int
+getAverageAge() : int
+getAvailableHousingSpace() : int
+getTotalHousingSpace() : int
+getAllStats() : std::string
+buildingDetails() : std::string
+populationDetails() : std::string
+resourceDetails() : std::string
BuildingMediator
-factory : BuildingFactory*
-government : ConcreteGovernment*
+build(buildingName : string, locationX : int, locat...
+demolish(locationX : int, locationY : int) : bool
TurnMediator
-government : ConcreteGovernment*
+nextRound() : void
+changeTaxRates(newTaxRates : int) : void
+handlePopulation() : int
State

Composite Design Pattern:


Visual Paradigm Standard(Willem(University of Pretoria))
Client
Component
+calculateTotalProduction() : int
+calculateTotalConsumption() : int
+getHousingSpace() : int
+getJobs() : int
Leaf
-building : Building*
+calculateTotalProduction() : int
+calculateTotalConsumption() : int
+getHousingSpace() : int
+getJobs() : int
Composite
-children : std::list<Component*>
+calculateTotalProduction() : int
+calculateTotalConsumption() : int
+calculateHousingSpace() : int
+getJobs() : int
+add(building : Component*) : bool
+remove(building : Component) : bool
+remove(index : int) : bool
+get(index : int) : Component
+CreateIterator() : Iterator*

Purpose

The Composite Design Pattern in our city management system enables us to model complex hierarchies of city elements—like buildings, parks, and districts—within a single unified structure. This pattern is especially effective for organizing elements that can be grouped or nested, such as city blocks containing individual buildings or parks within districts. By using the Composite pattern, we can treat individual components and groups of components in a uniform way, making it easier to manage and manipulate city structures as a whole.

Implementation

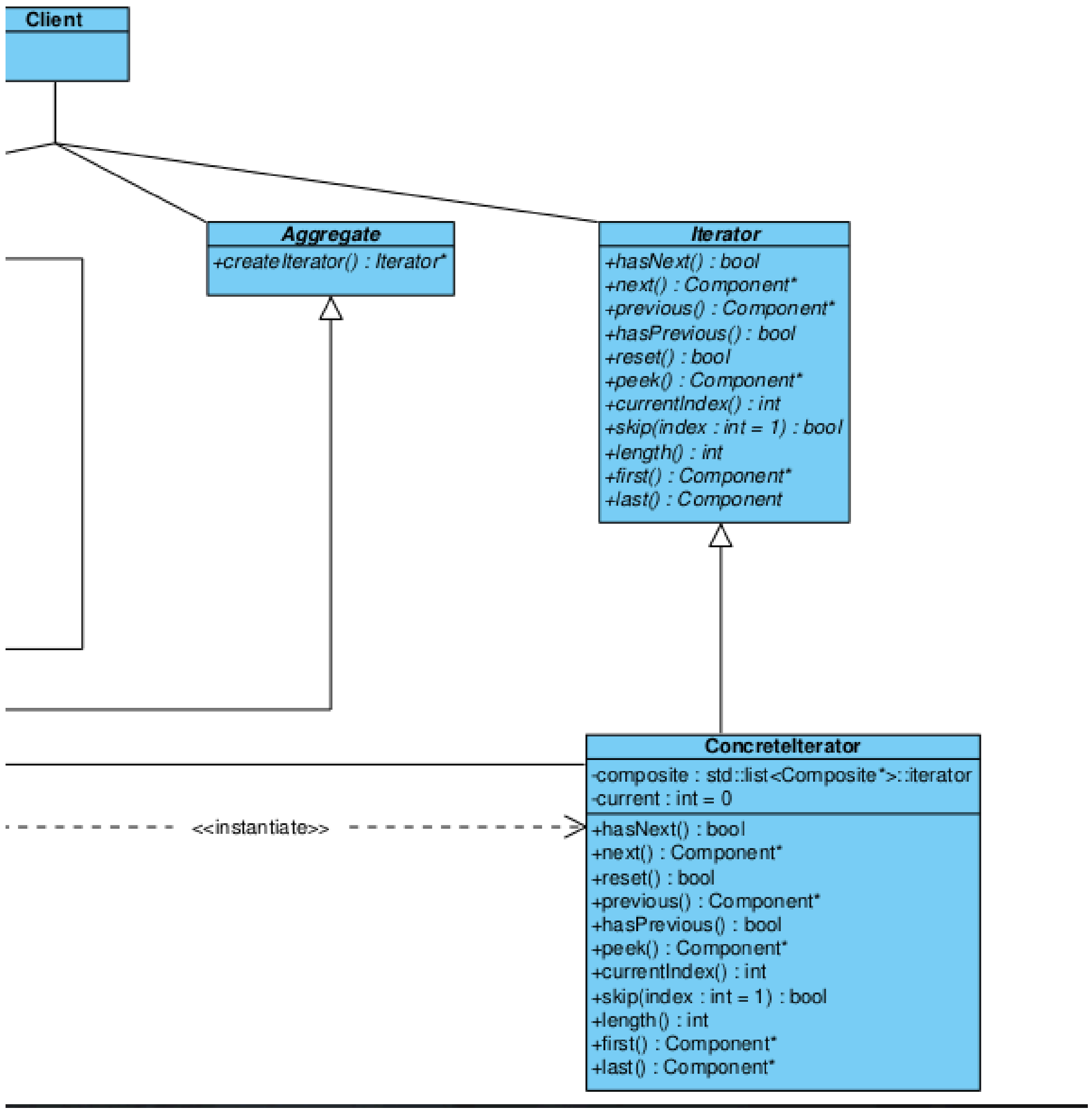
1. **Context Class (Component)**:
    - The abstract Component class defines a common interface for all city elements, enabling both individual buildings and collections of buildings to adhere to a single protocol. This uniform interface allows us to manage and query city elements without distinguishing between single elements and groups.
2. **Leaf Class (Building)**:
    - The Leaf class represents individual city elements like buildings. Each Leaf in the Composite can carry specific attributes and operations, such as space availability and resource requirements.
3. **Composite Class (CityComposite)**:
    - The Composite class encapsulates a collection of `Leaf` objects, enabling groups of buildings to be managed as a single unit. This class is responsible for operations like adding and removing buildings, and it can retrieve all buildings contained within itself and its sub-components.
4. **Composite Structure and Aggregation**:
    - The Composite class stores individual buildings (or other composites) within a 2D structure to represent city blocks. Each building or sub-composite within the structure can be accessed or modified, supporting the dynamic and hierarchical nature of the city layout.

Example

By using the Composite pattern, we can create a tree-like structure for the city, where entire districts or neighborhoods are treated as single entities, while individual buildings within these areas can still be accessed and manipulated independently.

.

Iterator Design Pattern:



Purpose

The Iterator Design Pattern allows for flexible traversal through the collection of city elements—whether they are single buildings or groups nested within a larger structure. This pattern is crucial for navigating through the city's complex hierarchy without exposing its underlying structure to the client code. By encapsulating the traversal logic, the Iterator makes it easier to search for specific elements, such as available residential buildings, without directly interacting with the composite's internals.

Implementation

1. **Context Class (Iterator)**:
    - The Iterator class provides the mechanism for navigating through the city's composite structure, including the retrieval of specific types of buildings, such as residential or commercial properties. This class defines methods for iterating over buildings based on criteria, ensuring that the iteration logic is encapsulated and independent of the composite's structure.
2. **Concrete Iterator Method (getHouse)**:
    - The getHouse method within the Iterator class iterates over the city's composite structure, searching for buildings with available space and specific characteristics. This method allows clients to retrieve specific building types without needing to understand the underlying organization of the city.
3. **Integration with Composite**:
    - The Iterator works in tandem with the Composite by calling methods like getAllBuildings(), which retrieve a collection of all building elements. This allows the Iterator to apply criteria or filters, such as finding the first available residential building, without accessing the composite's internal storage directly.

Example

Using the Iterator pattern, we can seamlessly search for residential buildings in the city that meet specific conditions, such as having available space. This allows for complex queries within a large city layout, improving the system's flexibility and extensibility.

The Iterator pattern acts as a single access point for navigating and interacting with the city's structure, which is built using the Composite pattern. By relying solely on the Iterator to traverse the city elements, we avoid directly interacting with the Composite's internal structure. This means that any changes made to the internal organization or data handling within Composite won't affect the way client code interacts with it. In other words, the Iterator provides a simplified and consistent way to access city elements, regardless of how they're stored or managed in Composite, making it a centralized and seamless entry point.

GitHub Branching Strategy

Throughout the project, we utilized four branches on our GitHub. Namely; “main”, “Class-Definitions”, “Testing”, and “additional-features”.

Our “main” branch served as the stable production branch, containing all our up to date and reviewed code. The “Class-Definitions” branch was used mostly throughout our initial implementation stage. We added our .h and .cpp files to this as we were working on the project, merging into main once the branch was at a high enough standard. The “Testing” branch was used for our testing stage. This includes our main.cpp, and our unit testing. Only after all tests were passed and we were happy, then we merged into “main”. The “additional-features” branch was used for anything not related to the previously mentioned branches. An example would be our Doxygen comments. Once everything was complete and reviewed, then it was merged into “main”.

https://docs.google.com/document/d/1mr7gzRIgOJUAeV4l-UJfysWwHmbm1FjR9VRC6nwPFro/edit?tab=t.0