Cool Cats
City Builder Report

Made By:

1. Tambi -u22511726
2. Khomotjo - u23544148
3. Tafara - u22565991
4. Karabelo - u23538318
5. Sean - u23592274
6. Thabiso - u22617362

Research Brief

Cities are a place inhabited by humans with a very large population which is very densely packed together. As such a city is a very complicated system that needs very defined and intentional design to ensure that all the components in this complicated system work efficiently and effectively. These complicated components include landmarks around the city and as such we have created parks , hospitals etc which we can add citizens to. A crucial aspect of the city is the opportunity to choose your government through voting , which is why we model voting in our city , for citizens to have a right to vote for their mayor. As we all know there are disparities in all the cities in the world when it comes to wealth distribution amongst citizens hence why we separated our citizens into Low, Middle and High class.

Multiple modes of transport are key to maintaining a working and vibrant city these include: train , bus , taxi , passenger plane and cargo. To maintain a large and industrial city it is very important that we have power , access to water and crucially a waste and sewage management system to ensure refuse in the city is collected and the city remains clean and habitable. That is why we thought it was incredibly important to include this in our simulation. Another aspect that is needed to ensure that a city runs efficiently is that the government must have enough money to render services to the residents of a city therefore a tax system is modeled in our system to ensure that the government has the budget to do its job.

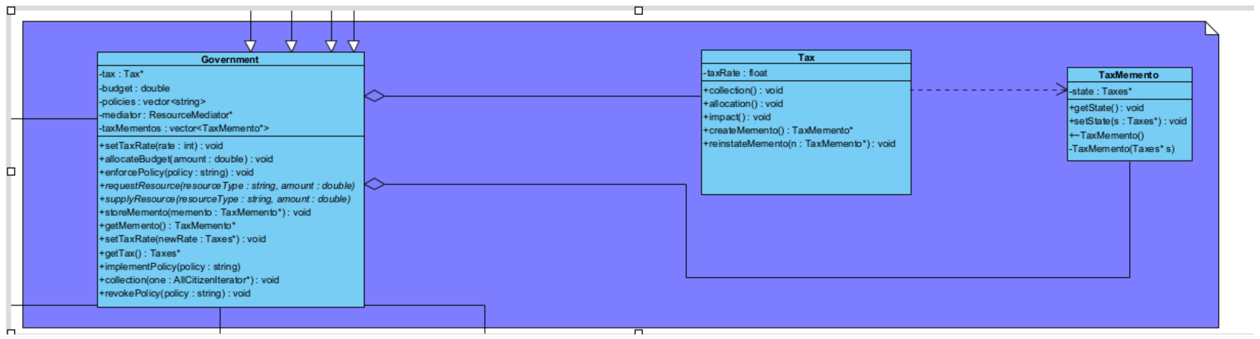
References:

Garvin, A. (2014). *The American city : what works, what doesn't*. New York: Mcgraw-Hill Education.

Design Patterns

Here are a list of design patterns we had decided to use, with examples of how we used them and a brief overview explaining our reasoning behind each design pattern:

Memento


Government
-tax : Tax*
-budget : double
-policies : vector<string>
-mediator : ResourceMediator*
-taxMementos : vector<TaxMemento*>
+setTaxRate(rate : int) : void
+allocateBudget(amount : double) : void
+enforcePolicy(policy : string) : void
+requestResource(resourceType : string, amount : double)
+supplyResource(resourceType : string, amount : double)
+storeMemento(memento : TaxMemento*) : void
+getMemento() : TaxMemento*
+setTaxRate(newRate : Taxes*) : void
+getTax() : Taxes*
+implementPolicy(policy : string)
+collection(one : AllCitizenIterator*) : void
+revokePolicy(policy : string) : void
Tax
-taxRate : float
+collection() : void
+allocation() : void
+impact() : void
+createMemento() : TaxMemento*
+reinstateMemento(n : TaxMemento*) : void
TaxMemento
-state : Taxes*
+getState() : void
+setState(s : Taxes*) : void
+~TaxMemento()
-TaxMemento(Taxes* s)

Overview:

We chose the Memento design pattern because we wanted the government to have the choice to revert back to a previous tax, if citizens are unhappy. There is a wide interface between the Tax and TaxMemento and a narrow interface between TaxMemento and government. This is to ensure that not everyone can just have access to a memento's state. We also decided for the government to hold a tax pointeras the government controls taxes.

Participants:

Originator:

- Tax

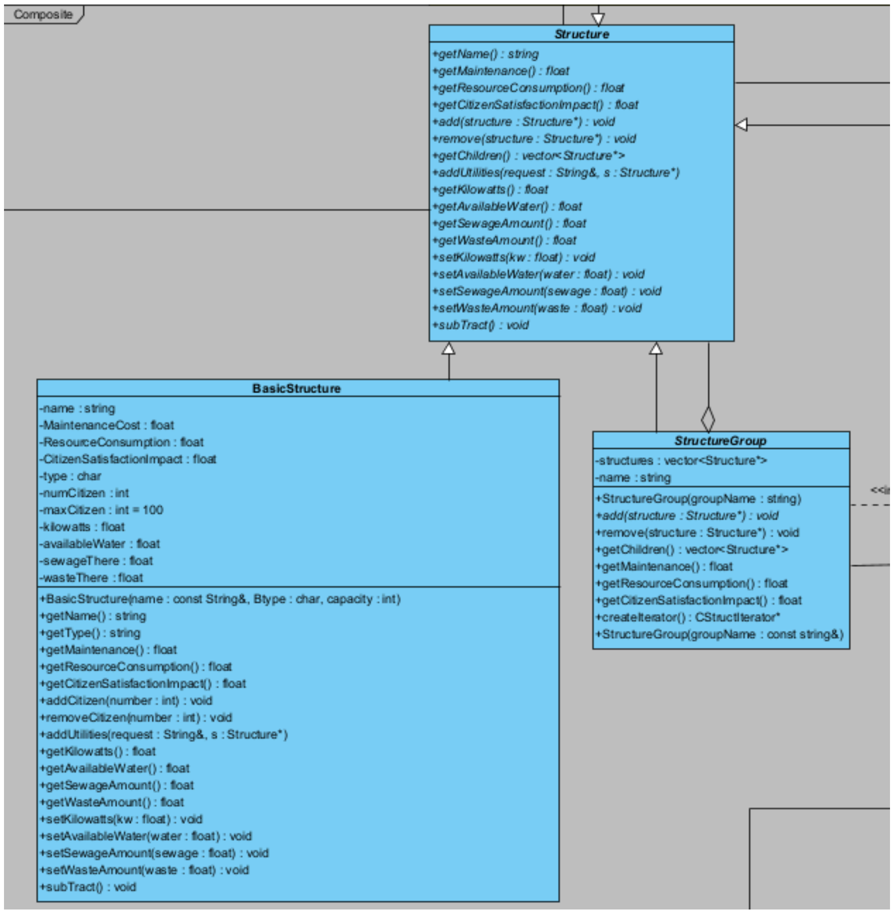
Memento:

- TaxMemento

Caretaker:

- Government

Composite


Composite
Structure
+getName() : string
+getMaintenance() : float
+getResourceConsumption() : float
+getCitizenSatisfactionImpact() : float
+add(structure : Structure*) : void
+remove(structure : Structure*) : void
+getChildren() : vector<Structure*>
+addUtilities(request : String&, s : Structure*)
+getKilowatts() : float
+getAvailableWater() : float
+getSewageAmount() : float
+getWasteAmount() : float
+setKilowatts(kw : float) : void
+setAvailableWater(water : float) : void
+setSewageAmount(sewage : float) : void
+setWasteAmount(waste : float) : void
+subTract() : void
BasicStructure
-name : string
-MaintenanceCost : float
-ResourceConsumption : float
-CitizenSatisfactionImpact : float
-type : char
-numCitizen : int
-maxCitizen : int = 100
-kilowatts : float
-availableWater : float
-sewageThere : float
-wasteThere : float
+BasicStructure(name : const String&, Btype : char, capacity : int)
+getName() : string
+getType() : string
+getMaintenance() : float
+getResourceConsumption() : float
+getCitizenSatisfactionImpact() : float
+addCitizen(number : int) : void
+removeCitizen(number : int) : void
+addUtilities(request : String&, s : Structure*)
+getKilowatts() : float
+getAvailableWater() : float
+getSewageAmount() : float
+getWasteAmount() : float
+setKilowatts(kw : float) : void
+setAvailableWater(water : float) : void
+setSewageAmount(sewage : float) : void
+setWasteAmount(waste : float) : void
+subTract() : void
StructureGroup
-structures : vector<Structure*>
-name : string
+StructureGroup(groupName : string)
+add(structure : Structure*) : void
+remove(structure : Structure*) : void
+getChildren() : vector<Structure*>
+getMaintenance() : float
+getResourceConsumption() : float
+getCitizenSatisfactionImpact() : float
+createIterator() : CStructIterator*
+StructureGroup(groupName : const string&)

<u>Overview:</u>

Composite allows us to treat individual buildings and compositions of objects(City areas) uniformly. For example there could be an industrial area that has a group of Industrial buildings.

Composite pattern makes it easier to handle both single buildings and collections of buildings in a consistent way.

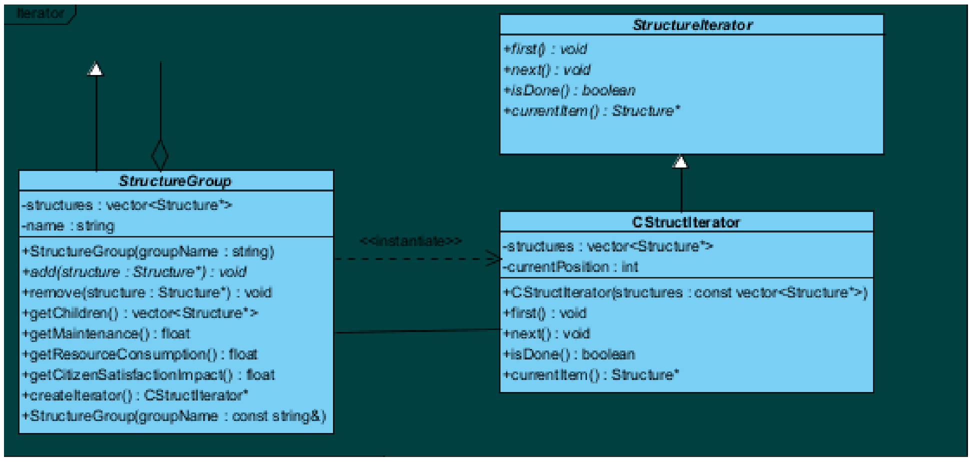
<u>Participants:</u>

Component: Structure

Leaf: Basic Structure

Composite: StructureGroup

Iterator


Iterator
StructureIterator
+first() : void
+next() : void
+isDone() : boolean
+currentItem() : Structure*
StructureGroup
-structures : vector<Structure*>
-name : string
+StructureGroup(groupName : string)
+add(structure : Structure*) : void
+remove(structure : Structure*) : void
+getChildren() : vector<Structure*>
+getMaintenance() : float
+getResourceConsumption() : float
+getCitizenSatisfactionImpact() : float
+createIterator() : CStructIterator*
+StructureGroup(groupName : const string&)
<<instantiate>>
CStructIterator
-structures : vector<Structure*>
-currentPosition : int
+CStructIterator(structures : const vector<Structure*>)
+first() : void
+next() : void
+isDone() : boolean
+currentItem() : Structure*

Overview:

Iterator provides a way to access the elements of Structure Group sequentially without exposing the underlying representation. This iterator allows us to loop through each Basic structure in a structure Group. This is useful when adding citizens to specific buildings or decorating specific structures.

Participants:

Aggregate

- Structure

ConcreteAggregate:

- StructureGroup

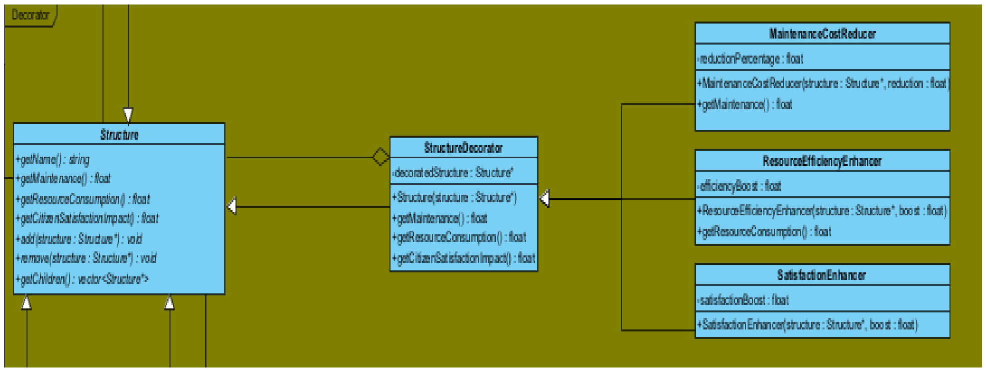
Iterator:

- StructureIterator

ConcreteAggregate:

- CstructIterator

Decorator


Decorator
Structure
+getName() : string
+getMaintenance() : float
+getResourceConsumption() : float
+getCitizenSatisfactionImpact() : float
+add(structure : Structure*) : void
+remove(structure : Structure*) : void
+getChildren() : vector<Structure*>
StructureDecorator
-decoratedStructure : Structure*
+Structure(structure : Structure*)
+getMaintenance() : float
+getResourceConsumption() : float
+getCitizenSatisfactionImpact() : float
MaintenanceCostReducer
-reductionPercentage : float
+MaintenanceCostReducer(structure : Structure*, reduction : float)
+getMaintenance() : float
ResourceEfficiencyEnhancer
-efficiencyBoost : float
+ResourceEfficiencyEnhancer(structure : Structure*, boost : float)
+getResourceConsumption() : float
SatisfactionEnhancer
-satisfactionBoost : float
+SatisfactionEnhancer(structure : Structure*, boost : float)

Overview:

Buildings can be enhanced with additional features like improving citizen satisfaction of residents, reducing maintenance cost or increasing resource efficiency.

Participants:

Component:

- Structure

ConcreteComponent:

- BasicStructure

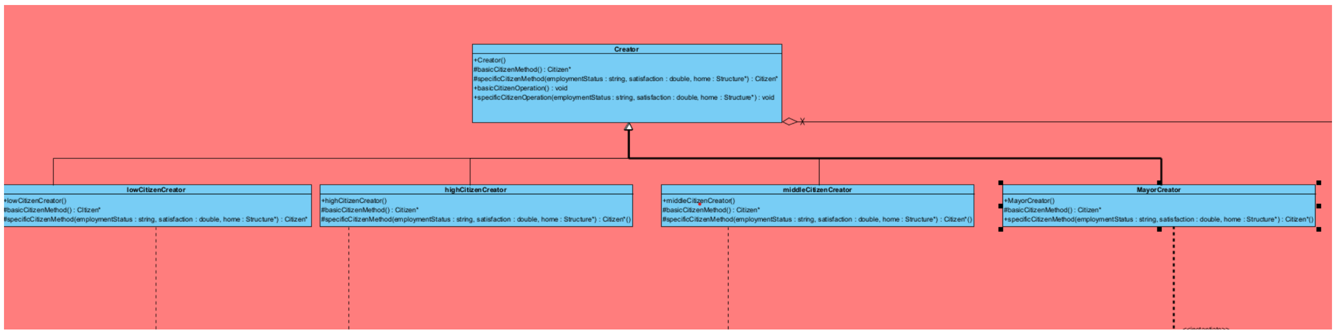
Decorator:

- StructureDecorator

ConcreteDecorator:

- ResourceEfficiencyEnhancer,
- MaintenanceCostReducer,
- SatisfactionEnhancer

Factory Method


Creator
+Creator()
#basicCitizenMethod() : Citizen*
#specificCitizenMethod(employmentStatus : string, satisfaction : double, home : Structure*) : Citizen*
+basicCitizenOperation() : void
+specificCitizenOperation(employmentStatus : string, satisfaction : double, home : Structure*) : void
lowCitizenCreator
+lowCitizenCreator()
#basicCitizenMethod() : Citizen*
#specificCitizenMethod(employmentStatus : string, satisfaction : double, home : Structure*) : Citizen*
highCitizenCreator
+highCitizenCreator()
#basicCitizenMethod() : Citizen*
#specificCitizenMethod(employmentStatus : string, satisfaction : double, home : Structure*) : Citizen*()
middleCitizenCreator
+middleCitizenCreator()
#basicCitizenMethod() : Citizen*
#specificCitizenMethod(employmentStatus : string, satisfaction : double, home : Structure*) : Citizen*()
MayorCreator
+MayorCreator()
#basicCitizenMethod() : Citizen*
+specificCitizenMethod(employmentStatus : string, satisfaction : double, home : Structure*) : Citizen*()

Overview:

Citizens can be created using a standard interface, which lets the subclasses decide which citizens to instantiate. Lets the creator defer instantiation to creator subclasses.

Participants:

Creator:

- Creator

ConcreteCreator:

- lowCitizenCreator,
- highCitizenCreator,
- middleCitizenCreator,
- MayorCreator

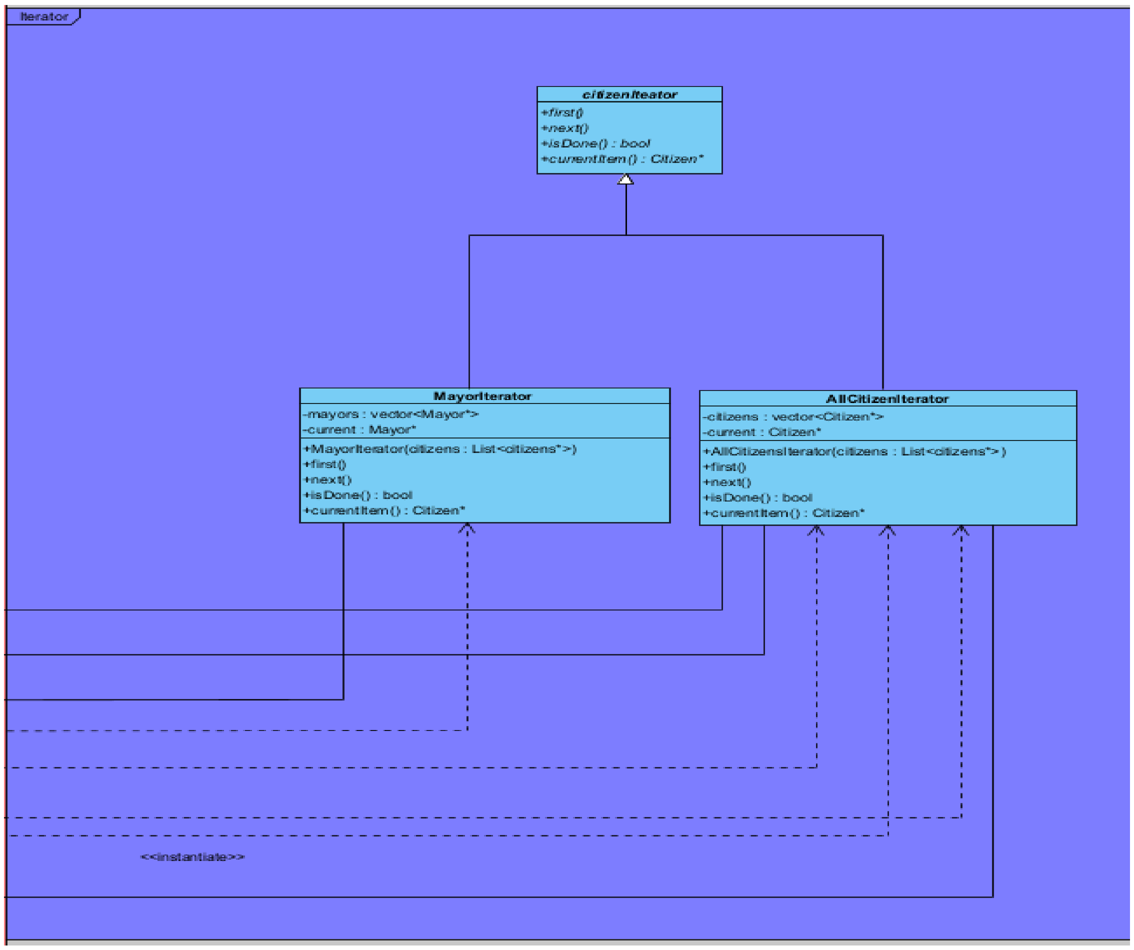
Product:

- Citizen

ConcreteProduct:

- Mayor,
- highCitizen,
- lowCitizen,
- middleCitizen

Iterator



Overview:

Provides a way to access all the citizens or just the mayors of the citizen aggregate class sequentially without exposing the citizens representation

Participants:

Iterator: citizenIterator

Concrete Iterator: MayorIterator, AllCitizenIterator

Aggregate: Citizen

Concrete Aggregate: Mayor, highCitizen, middleCiizen, lowCitizen

Chain Of Responsibility

Overview:

This implementation was done for the utilities. It allows the user to send more than one request at one time such that more than one utility is added to the building at once. The user can choose how many of these requests will be sent through with the user being able to send all the utilities at once for efficiency.

Participants:

ConcreteHandler1:

- PowerUtility

ConcreteHandler2:

- WaterUtility

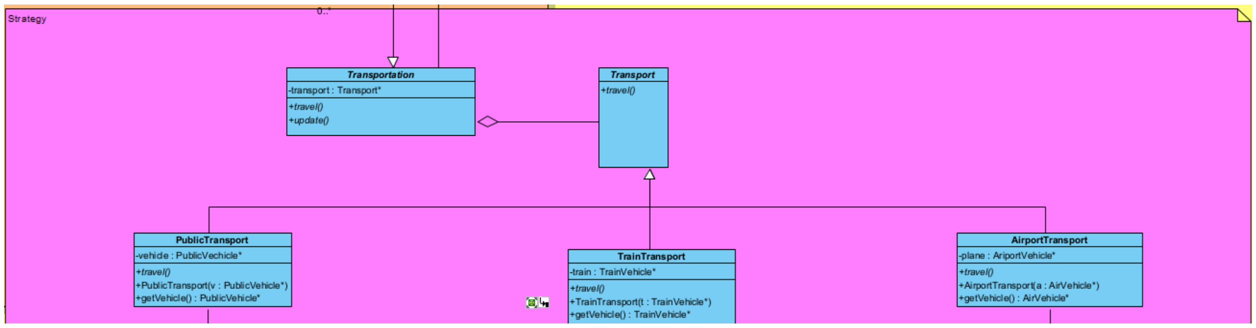
ConcreteHandler3:

- SewageUtility

ConcreteHandler4:

- WasteUtility

Strategy


Strategy
0..*
Transportation
-transport : Transport*
+travel()
+update()
Transport
+travel()
PublicTransport
-vehicle : PublicVechicle*
+travel()
+PublicTransport(v : PublicVehicle*)
+getVehicle() : PublicVehicle*
TrainTransport
-train : TrainVehicle*
+travel()
+TrainTransport(t : TrainVehicle*)
+getVehicle() : TrainVehicle*
AirportTransport
-plane : AriportVehicle*
+travel()
+AirportTransport(a : AirVehicle*)
+getVehicle() : AirVehicle*

<u>Overview:</u>

We decided to use the Strategy design pattern for handling our different types of transportation modes where the strategy function, travel(), was encapsulated and made interchangeable, letting its algorithm vary independently from the client that uses it. Allowing for variation in transportation around the city and surrounding cities

<u>Participants:</u>

Strategy:

- Transport

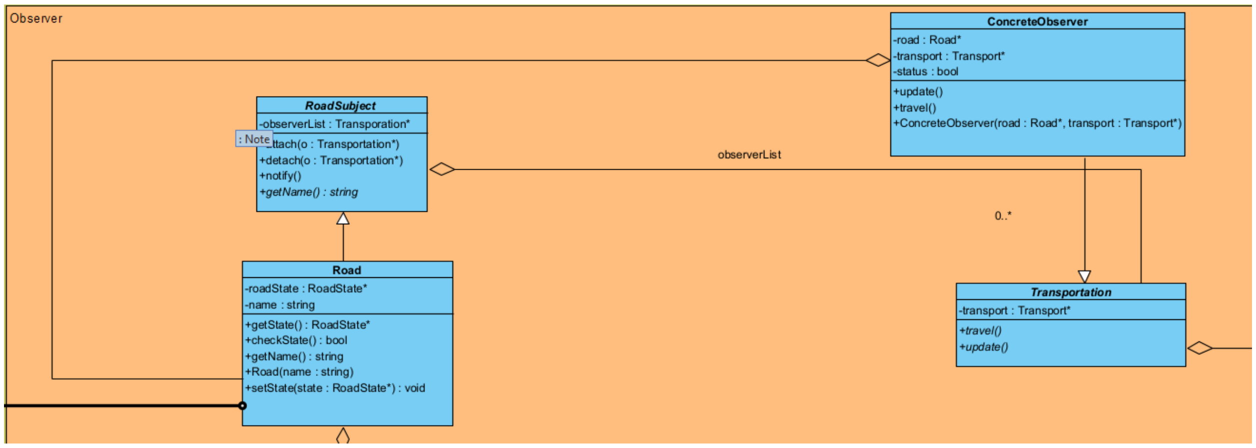
ConcreteStrategy:

- PublicTransport,
- TrainTransport,
- AirportTransport

Context:

- Transportation

Observer



Overview:

We decided to use the Observer design pattern for observing our Road objects as we use a collection of roads to make travel routes where we assign a Transport object to and that object would need to observe the state of any road in said travel route to see if it is in working condition or under construction which would make the travel route unusable. Observer is the best design pattern for this as it has a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Participants:

Subject:

- RoadSubject

ConcreteSubject:

- Road

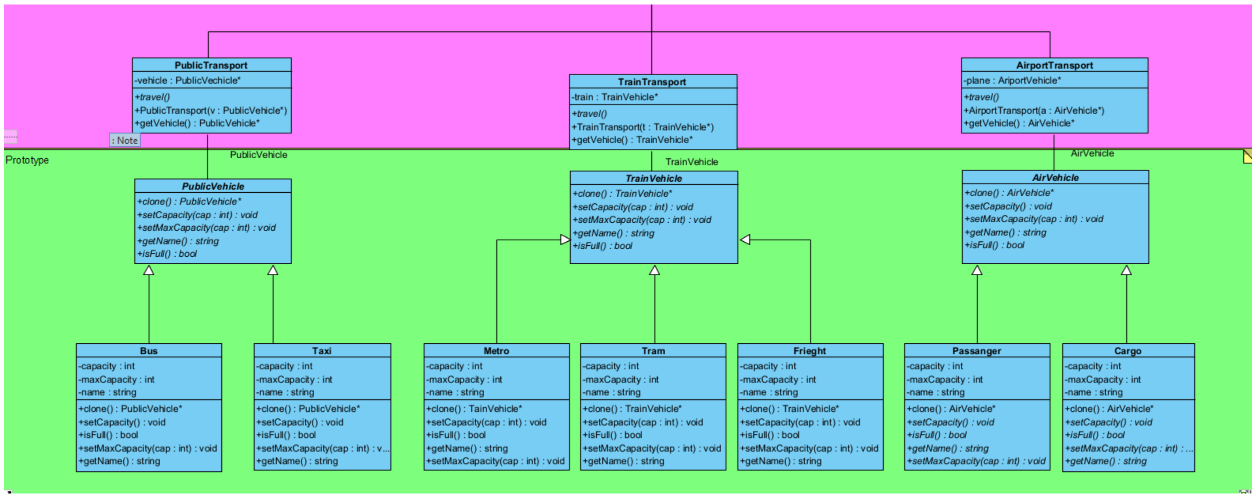
Observer:

- Transportation

ConcreteObsever:

- ConcreteObserver

Prototype



Overview:

We used the Prototype design pattern to specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. In this case the types of transportation variations.We used Prototype because creating new objects is computationally expensive and time consuming and would rather avoid such by creating copies of existing objects

Participants:

Prototype:

- PublicVehicle, TrainVehicle, AirVehicle

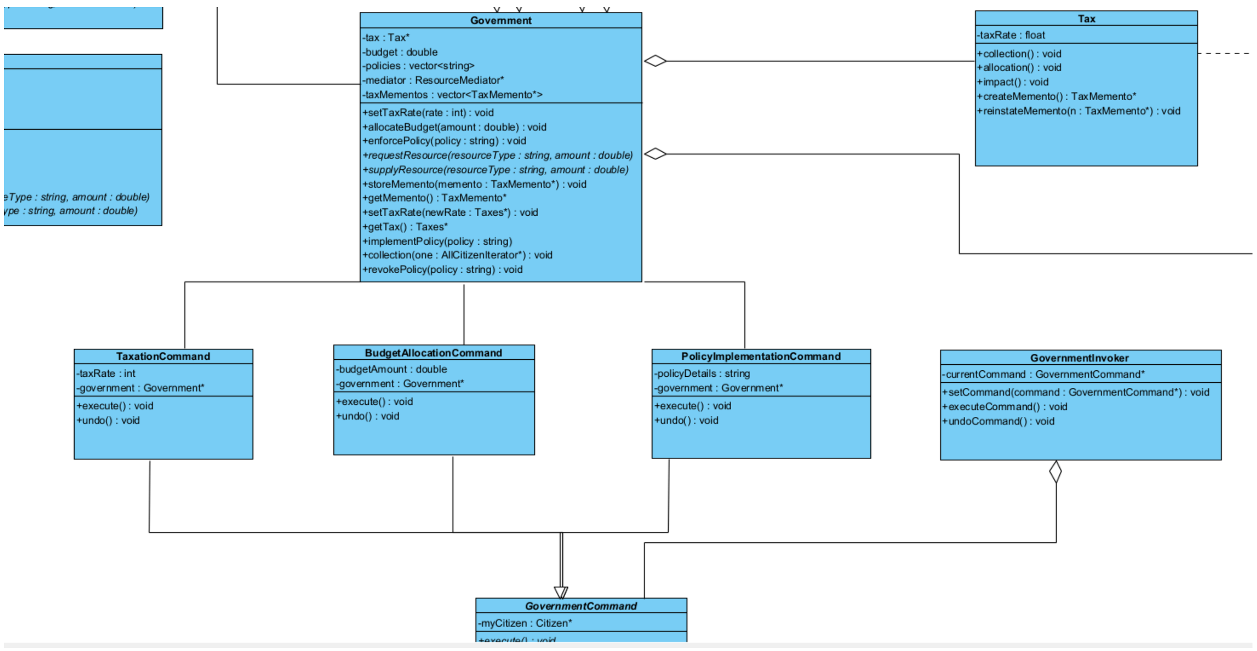
ConcretePrototype:

- Bus,Taxi,metro,Tram,Freight, Passenger, Cargo

Client:

- PublicTransportation, TrainTransportation, AirportTransportation

Command



eType : string, amount : double)
ype : string, amount : double)
Government
-tax : Tax*
-budget : double
-policies : vector<string>
-mediator : ResourceMediator*
-taxMementos : vector<TaxMemento*>
+setTaxRate(rate : int) : void
+allocateBudget(amount : double) : void
+enforcePolicy(policy : string) : void
+requestResource(resourceType : string, amount : double)
+supplyResource(resourceType : string, amount : double)
+storeMemento(memento : TaxMemento*) : void
+getMemento() : TaxMemento*
+setTaxRate(newRate : Taxes*) : void
+getTax() : Taxes*
+implementPolicy(policy : string)
+collection(one : AllCitizenIterator*) : void
+revokePolicy(policy : string) : void
Tax
-taxRate : float
+collection() : void
+allocation() : void
+impact() : void
+createMemento() : TaxMemento*
+reinstateMemento(n : TaxMemento*) : void
TaxationCommand
-taxRate : int
-government : Government*
+execute() : void
+undo() : void
BudgetAllocationCommand
-budgetAmount : double
-government : Government*
+execute() : void
+undo() : void
PolicyImplementationCommand
-policyDetails : string
-government : Government*
+execute() : void
+undo() : void
GovernmentInvoker
-currentCommand : GovernmentCommand*
+setCommand(command : GovernmentCommand*) : void
+executeCommand() : void
+undoCommand() : void
GovernmentCommand
-myCitizen : Citizen*

<u>Overview:</u>

We applied the Command pattern for the government components to encapsulate government actions (e.g., setting taxes, implementing policies, and allocating budgets) as individual command objects. This pattern provides flexibility in executing, undoing, and tracking these actions without modifying the invoker's core structure.

<u>Participants:</u>

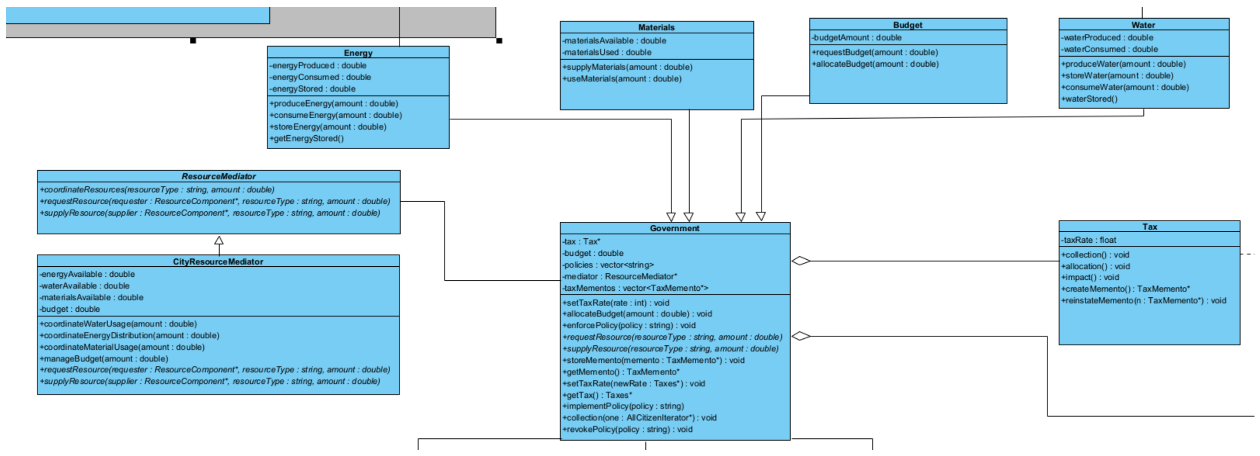
Command: GovernmentCommand

ConcreteCommand: TaxationCommand, BudgetAllocationCommand, PolicyImplementationCommand.

Invoker: GovernmentInvoker

Receiver: Government

Mediator


Energy
-energyProduced : double
-energyConsumed : double
-energyStored : double
+produceEnergy(amount : double)
+consumeEnergy(amount : double)
+storeEnergy(amount : double)
+getEnergyStored()
Materials
-materialsAvailable : double
-materialsUsed : double
+supplyMaterials(amount : double)
+useMaterials(amount : double)
Budget
-budgetAmount : double
+requestBudget(amount : double)
+allocateBudget(amount : double)
Water
-waterProduced : double
-waterConsumed : double
+produceWater(amount : double)
+storeWater(amount : double)
+consumeWater(amount : double)
+waterStored()
ResourceMediator
+coordinateResources(resourceType : string, amount : double)
+requestResource(requester : ResourceComponent*, resourceType : string, amount : double)
+supplyResource(supplier : ResourceComponent*, resourceType : string, amount : double)
CityResourceMediator
-energyAvailable : double
-waterAvailable : double
-materialsAvailable : double
-budget : double
+coordinateWaterUsage(amount : double)
+coordinateEnergyDistribution(amount : double)
+coordinateMaterialUsage(amount : double)
+manageBudget(amount : double)
+requestResource(requester : ResourceComponent*, resourceType : string, amount : double)
+supplyResource(supplier : ResourceComponent*, resourceType : string, amount : double)
Government
-tax : Tax*
-budget : double
-policies : vector<string>
-mediator : ResourceMediator*
-taxMementos : vector<TaxMemento*>
+setTaxRate(rate : int) : void
+allocateBudget(amount : double) : void
+enforcePolicy(policy : string) : void
+requestResource(resourceType : string, amount : double)
+supplyResource(resourceType : string, amount : double)
+storeMemento(memento : TaxMemento*) : void
+getMemento() : TaxMemento*
+setTaxRate(newRate : Taxes*) : void
+getTax() : Taxes*
+implementPolicy(policy : string)
+collection(one : AllCitizenIterator*) : void
+revokePolicy(policy : string) : void
Tax
-taxRate : float
+collection() : void
+allocation() : void
+impact() : void
+createMemento() : TaxMemento*
+reinstateMemento(n : TaxMemento*) : void

Overview:

We implemented the Mediator pattern to centralize and simplify resource management in the simulation. Each resource—such as materials, energy, water, and budget—has distinct behaviors, requirements, and dependencies. Using a mediator allows these resources to interact indirectly, with the mediator coordinating requests, allocations, and usage among them. This approach minimizes coupling, enabling easier adjustments to resource management without requiring changes to each individual component, making it ideal for a system with growing and interdependent resources.

Participants:

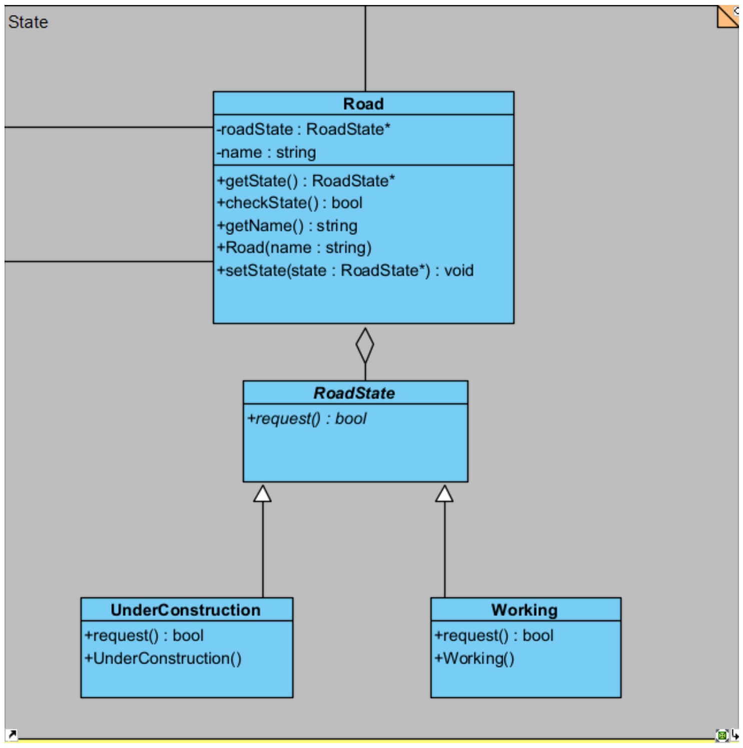
Mediator - ResourceMediator

ConcreteMediator- CityResourceMediator

Colleague - Government

ConcreteColleague - Energy, Water, Materials, Budget

State



Overview:

We used a State design pattern to set the state of the road as it allows an object to change its behaviour when its internal state changes, in this way the object will appear to change its class. We used State because of the Observer design pattern we used, as the observer observes the state of the road and here we can dictate the state of a road object to either be under construction, where the road is therefore unusable causing or working where the road is usable, which dictates how the travel() function in ConcreteObserver acts.

Participants:

Context: Road

State: RoadState

ConcreteState: UnderConstruction, Working

GitHub Branching Strategy

Each individual had their own branch with their own component that they worked on. Then each group member created and opened a pull request into the dev branch to merge their changes, this housed all components in their different folders. Then finally, we had a main branch which was our final changes, and a merge into a single folder for a simpler way to create our makefile Makefile and include all files in main.cpp

Commits per group member can be seen here:
https://github.com/COS214-Project-2024/Cool-Cats/pulse

Number of closed pull requests can be seen here:

https://github.com/COS214-Project-2024/Cool-Cats/pulls?q=is%3Apr+is%3Aclosed

Link to google docs Report:

SCAN ME

https://docs.google.com/document/d/19ISaTzrksIMs-bzit9RmgnynN1K0dgw7HejsaNrsK
us/edit?usp=sharing