UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

COS 214 - Project

by

Armand van der Colf - u22574982
Dreas Vermaak - u22497618
Jadyn Stoltz - u22609653
Liam van kasterop - u22539761
Diaan Botes - u22598538
Jean Steyn- u22537229
Eugen Vosloo - u20445696

In the Faculty of EBIT
University of Pretoria

2024/11

Table Of Contents:

4.1 – Research Brief: Urban Development and City Management

Urban Development Overview

Urban development represents the complex interplay of infrastructure, economics, and social systems in modern civilization. Cities function as dynamic hubs where resources, services, and growth must be carefully managed to ensure citizen well-being.

Core Systems & Implementation

Infrastructure & Economy

- Physical Components: Buildings, transportation, utilities

- Economic Elements: Commercial growth, taxation, resource management

- Social Dynamics: Population growth, housing, public services

Design Approach

Our simulation implements these urban principles through:

- Balanced infrastructure and resource management

- Dynamic economic and population systems

- Quality of life considerations

- Sustainable growth patterns

References:

World Bank, 2024. *Urban Development*. [online] Available at: https://www.worldbank.org/en/topic/urbandevelopment/overview [Accessed 4 November 2024].

Parnell, S., 2001. *Sustainable urban infrastructure planning in South Africa: A case study of Johannesburg's metropolitan development strategy*. Cities, [online] 18(6), pp.353-368. Available at: https://www.sciencedirect.com/science/article/abs/pii/S0264275101000269 [Accessed 4 November 2024].

Han, X. and Zhang, C., 2020. *Emerging strategies for urban governance in China: A study of smart cities and green development*. Procedia CIRP, [online] 89, pp.85-90. Available at: https://www.sciencedirect.com/science/article/abs/pii/S2210670720304935 [Accessed 4 November 2024].

4.2 – Design Pattern Application Report

Pattern 1: Observer Pattern

Implementation Context

- **Purpose**: To monitor and update various city metrics (e.g., population growth, economic growth, citizen satisfaction, resource consumption) in real-time.
- **Context**: The system needs to notify multiple observers about changes in city statistics, allowing each observer to handle updates independently.

Technical Implementation

- **Subject Interface**: `CityStats`
  - Manages a list of observers and provides methods to attach and detach them.
  - Method: `notify(int population, double satisfaction, double economicGrowth)` to update all observers.
- **Concrete Subject**: `ConcreteCityStats`
  - Extends `CityStats` and implements the logic to update city statistics.
  - Method: `setStats(double sat, double growth)` to set new statistics and notify observers.
- **Observer Interface**: `CityObserver`
  - Defines the interface for objects that should be notified of changes in city statistics.
  - Method: `update(int population, double satisfaction, double economicGrowth)` to receive updates.
- **Concrete Observers**:
  - `PopulationGrowth`
    - Calculates and displays population growth metrics.
  - `EconomicGrowth`
    - Calculates and displays economic growth metrics.
  - `CitizenSatisfaction`
    - Calculates and updates citizen satisfaction metrics.
  - `ResourceConsumption`
    - Calculates and displays resource consumption metrics.

Benefits

- **Loose Coupling**: Observers are decoupled from the subject, allowing for independent changes and extensions.
- **Scalability**: New observers can be added without modifying the subject.
- **Real-Time Updates**: Observers receive immediate updates when the subject’s state changes.

Pattern 2: Factory Pattern

Implementation Context

- **Purpose**: To create different types of buildings (e.g., residential, commercial, industrial, landmarks) in a city simulation.
- **Context**: The system needs to instantiate various building types dynamically based on input parameters.

Technical Implementation

- **Factory**: `BuildingFactory`
  - Defines the interface for creating buildings.
  - Method: `createBuilding(const std::string& type, const std::string& name, double area, int capacity, double cost, const std::string& districtName, const std::string& neighborhoodName)`
- **Concrete Factory**: `ConcreteBuildingFactory`
  - Implements the `BuildingFactory` interface.
  - Method: `createBuilding` creates specific building types based on the provided type parameter.
  - Ensures buildings are added to the correct district and neighborhood within the city.
- **Product**: `Building`
  - Defines the interface for all building types.
  - Methods: `getDescription`, `getCost`, `getBuildingType`, `getFloorArea`, `getMaxCapacity`, `setMaxCapacity`, `add`, `remove`
- **Concrete Products**:
  - `Residential`
    - Implements the `Building` interface for residential buildings.
  - `Commercial`
    - Implements the `Building` interface for commercial buildings.
  - `Industrial`
    - Implements the `Building` interface for industrial buildings.
  - `Landmarks`
    - Implements the `Building` interface for landmark buildings.

Benefits

- **Encapsulation**: The creation logic for different building types is encapsulated within the factory, promoting single responsibility.
- **Flexibility**: New building types can be added without modifying existing code, adhering to the open/closed principle.
- **Reusability**: The factory method can be reused to create buildings in different contexts, ensuring consistency.

Pattern 3: StrategyPattern

Implementation Context

- **Purpose**: To calculate different types of taxes (e.g., income tax, building tax) using interchangeable strategies.
- **Context**: The system needs to apply various tax calculation strategies dynamically based on the type of tax required.

Technical Implementation

- **Strategy Interface**: `TaxStrategy`
    - Defines the interface for tax calculation.
    - Method: `calculateTax(double amount)`
- **Concrete Strategies**:
    - IncomeTax
        - Implements `calculateTax` with specific logic for income tax.
        - Additional method: `calculateTotalTax` to calculate total tax for all citizens.
    - BuildingTax
        - Implements `calculateTax` with specific logic for building tax.
        - Additional method: `calculateTotalTax` to calculate total tax for all buildings.
- **Context**: `TaxSystem`
    - Maintains a reference to a `TaxStrategy` object.
    - Methods:
        - `setStrategy(TaxStrategy* newStrategy)`: Changes the strategy at runtime.
        - `calculateTax(double amount)`: Delegates the tax calculation to the current strategy.

Benefits

- **Flexibility**: Easily switch between different tax calculation strategies at runtime.
- **Maintainability**: Adding new tax calculation strategies does not require modifying existing code.
- **Single Responsibility**: Each strategy class handles its specific tax calculation logic, adhering to the single responsibility principle.

Pattern 4: State Pattern

Implementation Context

- **Purpose**: To manage the state of buildings in a city simulation, allowing buildings to change behaviour based on their state.
- **Context**: The system needs to represent different states of a building (e.g., under construction, operational, abandoned) and change the building's behaviour accordingly.

Technical Implementation

- **State Interface**: `BuildingState`
    - Defines the interface for building states.
    - Method: `getState() const` to return the current state as a string.
- **Concrete States**:
    - UnderConstruction
        - Implements `BuildingState` and represents a building that is under construction.
        - Method: `getState() const` returns "Under Construction".
    - Operational
        - Implements `BuildingState` and represents a building that is operational.
        - Method: `getState() const` returns "Operational".
    - Abandoned
        - Implements `BuildingState` and represents a building that is abandoned.
        - Method: `getState() const` returns "Abandoned".
- **Context Class**: `Building`
    - Maintains a reference to a `BuildingState` object.
    - Methods:
        - `setState(BuildingState* newState)`: Changes the state of the building.
        - `getState() const`: Returns the current state of the building.

Benefits

- **State Encapsulation**: Each state is encapsulated in a separate class, promoting single responsibility.
- **Flexibility**: The building can change its behaviour dynamically by switching states.
- **Maintainability**: Adding new states does not require modifying existing state classes or the context class.

Pattern 5: Prototype Pattern Implementation

Implementation Context

Purpose: To create new instances of city citizens (e.g., workers, retired individuals, students) by cloning existing prototypes.

Context: The system needs to efficiently create new citizen objects with similar properties to existing ones, allowing for easy duplication and customization.

Technical Implementation

Prototype Interface: `CitizenPrototype`

- Defines the interface for cloning and common properties of citizens.
- **Method:** `clone() const` to create a copy of the citizen.
- **Method:** `getRole() const` to get the role of the citizen.
- **Method:** `performTask()` to perform the citizen's task.

Concrete Prototypes:

- `Worker`: Implements `CitizenPrototype` for worker citizens.
    - **Method:** `clone() const` to create a copy of the worker.
    - **Method:** `getRole() const` to return "Worker".
    - **Method:** `performTask()` to perform the worker's task.
- `Retired`: Implements `CitizenPrototype` for retired citizens.
    - **Method:** `clone() const` to create a copy of the retired citizen.
    - **Method:** `getRole() const` to return "Retired".
    - **Method:** `performTask()` to perform the retired citizen's task.
    - **Method:** `setSalary(double newSalary)` to prevent modifying pension.
- `Student`: Implements `CitizenPrototype` for student citizens.
    - **Method:** `clone() const` to create a copy of the student.
    - **Method:** `getRole() const` to return "Student".
    - **Method:** `performTask()` to perform the student's task.
    - **Method:** `setSalary(double newSalary)` to set part-time job salary.

Benefits

- **Efficient Object Creation:** New objects are created by cloning existing prototypes, reducing the need for repetitive initialization.
- **Customization:** Cloned objects can be customised independently of their prototypes.
- **Flexibility:** New prototype types can be added without changing existing code.

Pattern 6: Composite Pattern #1 Implementation

Implementation Context

Purpose: To represent a city structure where individual components (e.g., buildings, neighbourhoods, districts) can be composed into larger structures, allowing for hierarchical organisation and management.

Context: The system needs to manage complex city structures by treating individual components and compositions of components uniformly.

Technical Implementation

Component Interface: `CityComponent`

- Defines the interface for all city components.
- **Method:** `add(CityComponent *component)` to add a component.
- **Method:** `remove(CityComponent *component)` to remove a component.
- **Method:** `getName() const` to get the name of the component.
- **Method:** `print(int indent = 0) const` to print the component details.

Composite Components:

- `Neighbourhood`: Implements `CityComponent` for neighbourhoods.
    - **Method:** `add(CityComponent *component)` to add a building to the neighbourhood.
    - **Method:** `remove(CityComponent *component)` to remove a building from the neighbourhood.
    - **Method:** `getName() const` to return the neighbourhood name.
    - **Method:** `print(int indent = 0) const` to print the neighbourhood details.
- `District`: Implements `CityComponent` for districts.
    - **Method:** `add(CityComponent *component)` to add a neighbourhood to the district.
    - **Method:** `remove(CityComponent *component)` to remove a neighbourhood from the district.
    - **Method:** `getName() const` to return the district name.
    - **Method:** `print(int indent = 0) const` to print the district details.
- `City`: Implements `CityComponent` for the entire city.
    - **Method:** `add(CityComponent *component)` to add a district to the city.
    - **Method:** `remove(CityComponent *component)` to remove a district from the city.
    - **Method:** `getName() const` to return the city name.
    - **Method:** `print(int indent = 0) const` to print the city details.

Benefits

- **Hierarchical Organisation:** Allows for the creation of complex structures by composing simple components.
- **Uniformity:** Treats individual components and compositions uniformly, simplifying management.
- **Flexibility:** Easily add or remove components without affecting the overall structure.

Pattern 7: Decorator Pattern Implementation

Implementation Context

Purpose: To dynamically add new functionalities to buildings (e.g., energy efficiency, capacity, security) without altering their structure.

Context: The system needs to enhance buildings with additional features while keeping the original building class unchanged.

Technical Implementation

Component Interface: `BaseBuilding`

- Defines the interface for all buildings.
- **Method:** `getDescription() const` to get the building description.
- **Method:** `getCost() const` to get the building cost.

Concrete Component:

- `Building`: Implements `BaseBuilding` for basic buildings.
  - **Method:** `getDescription() const` to return the building description.
  - **Method:** `getCost() const` to return the building cost.

Decorator: `BuildingUpgrade`

- Extends `Building` to add new functionalities.
  - **Method:** `getDescription() const` to return the building description.
  - **Method:** `getCost() const` to return the building cost.
  - **Method:** `getUpgradeCost() const` to get the cost of the upgrade.
  - **Method:** `incrementUpgradeLevel()` to increase the upgrade level.

Concrete Decorators:

- `EnergyEfficiencyUpgrade`: Adds energy efficiency features to a building.
  - **Method:** `applyUpgrade()` to apply the energy efficiency upgrade.
  - **Method:** `getUpgradeCost() const` to return the cost of the energy efficiency upgrade.
- `CapacityUpgrade`: Adds capacity features to a building.
  - **Method:** `applyUpgrade()` to apply the capacity upgrade.
  - **Method:** `getUpgradeCost() const` to return the cost of the capacity upgrade.
  - **Method:** `getDescription() const` to return the building description with capacity upgrade details.
- `SecurityUpgrade`: Adds security features to a building.
  - **Method:** `applyUpgrade()` to apply the security upgrade.
  - **Method:** `getUpgradeCost() const` to return the cost of the security upgrade.
  - **Method:** `getDescription() const` to return the building description with security upgrade details.

Benefits

- **Flexible Enhancements:** Allows for the addition of new functionalities to buildings without modifying their structure.
- **Reusability:** Decorators can be reused across different buildings.
- **Scalability:** New decorators can be added without changing existing code.

Pattern 8: Composite Pattern#2 Implementation

Implementation Context

Purpose: To manage and distribute capital among various government departments (e.g., health, education, security, transportation) in a hierarchical structure.

Context: The system needs to handle complex government structures by treating individual departments and compositions of departments uniformly.

Technical Implementation

Component Interface: `GovDepartment`

- Defines the interface for all government departments.
- **Method:** `setCapital(double Capital)` to set the department's capital.
- **Method:** `getCapital()` to get the department's capital.
- **Method:** `getBuilding()` to get the department's building.

Composite Component:

- `CompositeGovDepartment`: Implements `GovDepartment` for composite government departments.
  - **Method:** `addDepartment(GovDepartment *department)` to add a department.
  - **Method:** `removeDepartment(GovDepartment *department)` to remove a department.
  - **Method:** `distributeCapital()` to distribute capital among departments.
  - **Method:** `getDepartmentCount() const` to get the number of departments.
  - **Method:** `getDepartments() const` to get the list of departments.

Leaf Components:

- `HealthDepartment`: Implements `GovDepartment` for the health department.
  - **Method:** `getBuilding()` to return the building.
  - **Method:** `getCapital()` to return the capital.
  - **Method:** `setCapital(double capital)` to set the capital.
- `EducationDepartment`: Implements `GovDepartment` for the education department.
  - **Method:** `getBuilding()` to return the building.
  - **Method:** `getCapital()` to return the capital.
  - **Method:** `setCapital(double Capital)` to set the capital.
- `SecurityDepartment`: Implements `GovDepartment` for the security department.
  - **Method:** `getBuilding()` to return the building.

    - **Method:** `getCapital()` to return the capital.
    - **Method:** `setCapital(double Capital)` to set the capital.
- `TransportationDepartment`: Implements `GovDepartment` for the transportation department.
    - **Method:** `getBuilding()` to return the building.
    - **Method:** `getCapital()` to return the capital.
    - **Method:** `setCapital(double Capital)` to set the capital.

Benefits

- **Hierarchical Organisation:** Allows for the creation of complex structures by composing simple components.
- **Uniformity:** Treats individual components and compositions uniformly, simplifying management.
- **Flexibility:** Easily add or remove components without affecting the overall structure.

Pattern 9: Builder Pattern Implementation

Implementation Context

- **Purpose**: To construct complex infrastructure projects (e.g., utility networks, road networks, public transit systems) in a city simulation.
- **Context**: The system needs to create various types of infrastructure step-by-step, ensuring that the construction process is flexible and can be customised.

Technical Implementation

- **Builder Interface**: `InfrastructureBuilder`
    - Defines the interface for building infrastructure.
    - Methods: `reset()`, `BuildFoundation()`, `BuildStructure()`, `BuildFinishing()`, `GetResult()`, `isValidBuild()`, `estimateTotalCost()`
- **Concrete Builders**:
    - `UtilityNetworkBuilder`
        - Implements the `InfrastructureBuilder` interface for utility networks.
        - Methods: `BuildFoundation()`, `BuildStructure()`, `BuildFinishing()`, `GetResult()`, `isValidBuild()`, `estimateTotalCost()`
    - `RoadNetworkBuilder`
        - Implements the `InfrastructureBuilder` interface for road networks.
        - Methods: `BuildFoundation()`, `BuildStructure()`, `BuildFinishing()`, `GetResult()`, `isValidBuild()`, `estimateTotalCost()`
    - `PublicTransitBuilder`
        - Implements the `InfrastructureBuilder` interface for public transit systems.
        - Methods: `BuildFoundation()`, `BuildStructure()`, `BuildFinishing()`, `GetResult()`, `isValidBuild()`, `estimateTotalCost()`

- **Director Class**: `CityPlanner`
    - Directs the construction process using a builder.
    - Methods:
        - `constructInfrastructure(InfrastructureBuilder& builder)`: Constructs the infrastructure using the provided builder.
        - `estimateProjectCost(InfrastructureBuilder& builder)`: Estimates the total cost of the project.
        - `clearProjects()`: Clears all completed projects.
        - `getCompletedProjects() const`: Returns a list of completed projects.
- **Product Classes**:
    - `PublicTransit`
        - Represents public transit infrastructure.
        - Methods: `setDescription()`, `getDescription()`, `setCost()`, `getCost()`, `getType()`, `isValid()`, `clone()`, `setCompleted()`, `getCompleted()`, `displayInfo()`
    - `RoadNetwork`
        - Represents road network infrastructure.
        - Methods: `setDescription()`, `getDescription()`, `setCost()`, `getCost()`, `getType()`, `isValid()`, `clone()`, `setCompleted()`, `getCompleted()`, `displayInfo()`
    - `UtilityNetwork`
        - Represents utility network infrastructure.
        - Methods: `setDescription()`, `getDescription()`, `setCost()`, `getCost()`, `getType()`, `isValid()`, `clone()`, `setCompleted()`, `getCompleted()`, `displayInfo()`

Benefits

- **Separation of Concerns**: The construction process is separated from the representation, promoting single responsibility.
- **Flexibility**: Different builders can construct different types of infrastructure, allowing for customization.
- **Reusability**: The same construction process can be reused with different builders to create various types of infrastructure.

Pattern 10: Mediator Pattern Implementation

Implementation Context

Purpose: To manage and coordinate resource distribution among various utility systems (power, water, sewage, waste) in a city infrastructure.

Context: The system needs to handle complex interactions between utility systems while maintaining loose coupling.

Technical Implementation

Mediator Interface: `ResourceDistributor`

- Defines the interface for resource distribution and utility management.
- **Method:** `createBuilding(string type, string name, double area, int capacity, double cost)`
- **Method:** `notify(Building* building)`
- **Method:** `distributeResources(string resource, int amount)`
- **Method:** `registerUtility(string name, UtilitySystem* system)`
- **Method:** `unregisterUtility(string name)`
- **Method:** `hasUtility(string name)`
- **Method:** `getUtility(string name)`
- **Method:** `getRegisteredUtilities()`
- **Method:** `canDistributeResource(string resource, int amount)`
- **Method:** `getAvailableResources()`

Concrete Mediator:

CityResourceDistribution: Implements `ResourceDistributor`

- **Member:** `utilities` (map of utility systems)
- **Member:** `resourcePool` (map of available resources)
- **Member:** `factory` (building factory)
- **Method:** `addResourceToPool(string resource, int amount)`
- **Method:** `getResourcePoolAmount(string resource)`
- **Method:** `performMaintenanceCheck()`
- **Method:** `getTotalResources()`

Colleague Components:

PowerPlant: Implements `UtilitySystem`

- **Method:** `generatePower()`
- **Method:** `setOperational(bool status)`
- Standard utility methods (receive/process resources, maintenance)

WaterSupply: Implements `UtilitySystem`

- **Method:** `getWaterQuality()`
- **Method:** `treatWater()`
- Standard utility methods

SewageSystem: Implements `UtilitySystem`

- **Method:** `connectZone(string zoneName)`
- **Method:** `disconnectZone(string zoneName)`
- **Method:** `isZoneConnected(string zoneName)`
- **Method:** `processSewage()`

WasteManagement: Implements `UtilitySystem`

- **Method:** `processWaste()`
- **Method:** `getRemainingCapacity()`
- Standard utility methods

Benefits

- **Decoupled Communication:** Utilities don't communicate directly, reducing dependencies.
- **Centralised Control:** Resource distribution and management handled through a single mediator.
- **Extensibility:** Easy to add new utility systems without modifying existing code.
- **Resource Management:** Efficient handling of resource pools and distribution.
- **Maintenance:** Centralised maintenance checks and operational status monitoring.

The implementation demonstrates a robust mediator pattern where `CityResourceDistribution` acts as the central coordinator for all utility systems, managing their interactions and resource distribution while maintaining loose coupling between components. Each utility system has specialised functionality while conforming to a common `UtilitySystem` interface, allowing for uniform treatment in the mediator's operations.

Pattern 11: Adapter Pattern Implementation

Implementation Context

Purpose: To make a legacy transport system work with a modern interface.

Context: The system needs to integrate an old transport system with a new, modern transport interface without modifying the legacy system.

Technical Implementation

Adapter Interface: `ModernTransport`

- Defines the interface for modern transport systems.

- **Method:** `operationModernTransport()`
- **Method:** `getPassengerCapacity() const`
- **Method:** `getFuelEfficiency() const`
- **Method:** `getTransportType() const`
- **Method:** `getMaintenanceCost() const`
- **Method:** `isOperational() const`

Concrete Adapter: `TransportSystemAdapter`

- Adapts a `LegacyTransportSystem` to work with the `ModernTransport` interface.
- **Member:** `legacySystem` (pointer to the legacy transport system)
- **Method:** `operationModernTransport() override`
- **Method:** `getPassengerCapacity() const override`
- **Method:** `getFuelEfficiency() const override`
- **Method:** `getTransportType() const override`
- **Method:** `getMaintenanceCost() const override`
- **Method:** `isOperational() const override`
- **Method:** `updateLegacySystem()`
- **Method:** `hasLegacySystem() const`

Legacy Component: `LegacyTransportSystem`

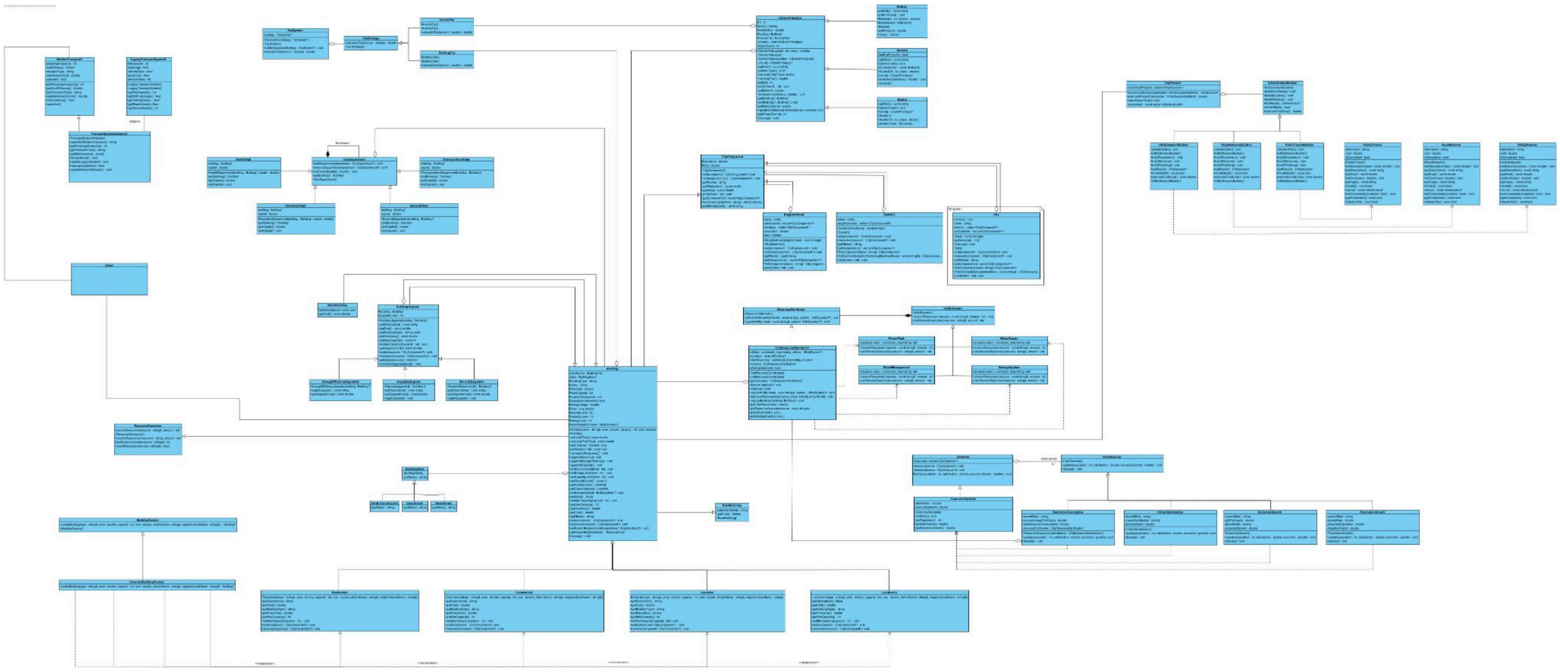
- Provides basic functionality for managing legacy transport vehicles.
- **Method:** `operateOldTransport()`
- **Method:** `getOldCapacity() const`
- **Method:** `getOldFuelUsage() const`
- **Method:** `getVehicleClass() const`
- **Method:** `getRepairCosts() const`
- **Method:** `getServiceStatus() const`

Benefits

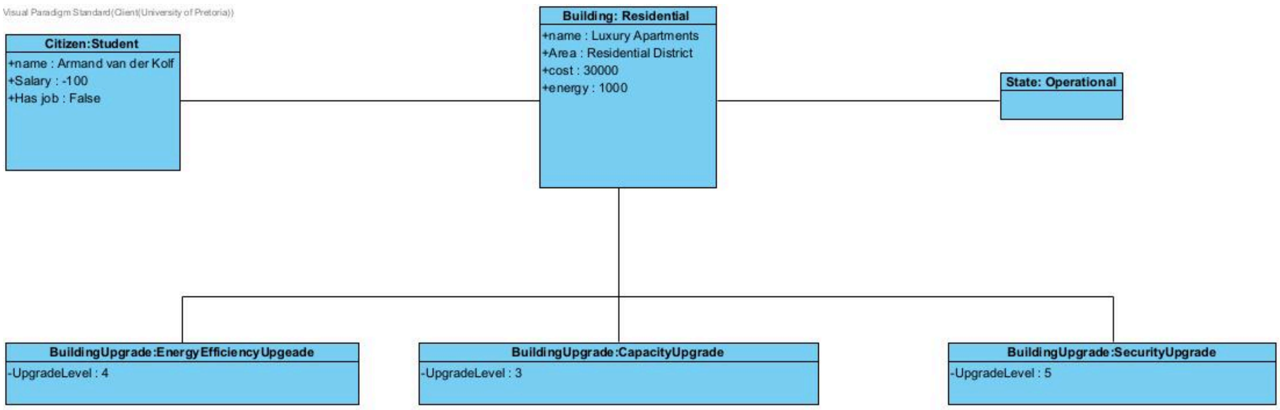
- **Compatibility:** Allows legacy systems to work with modern interfaces without modification.
- **Reusability:** Reuses existing legacy systems, reducing the need for complete overhauls.
- **Flexibility:** Easily integrates with new systems by adapting old interfaces to new ones.

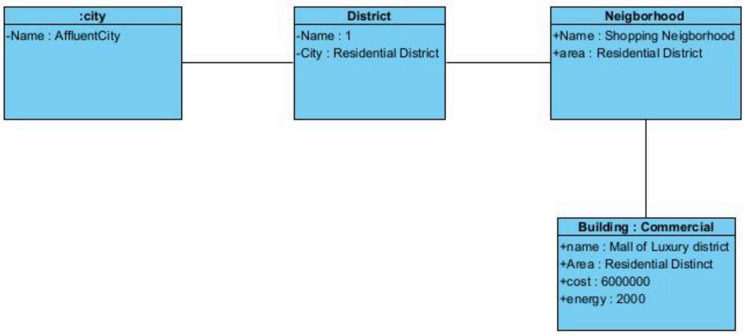
UML Diagrams:

Class Diagram:

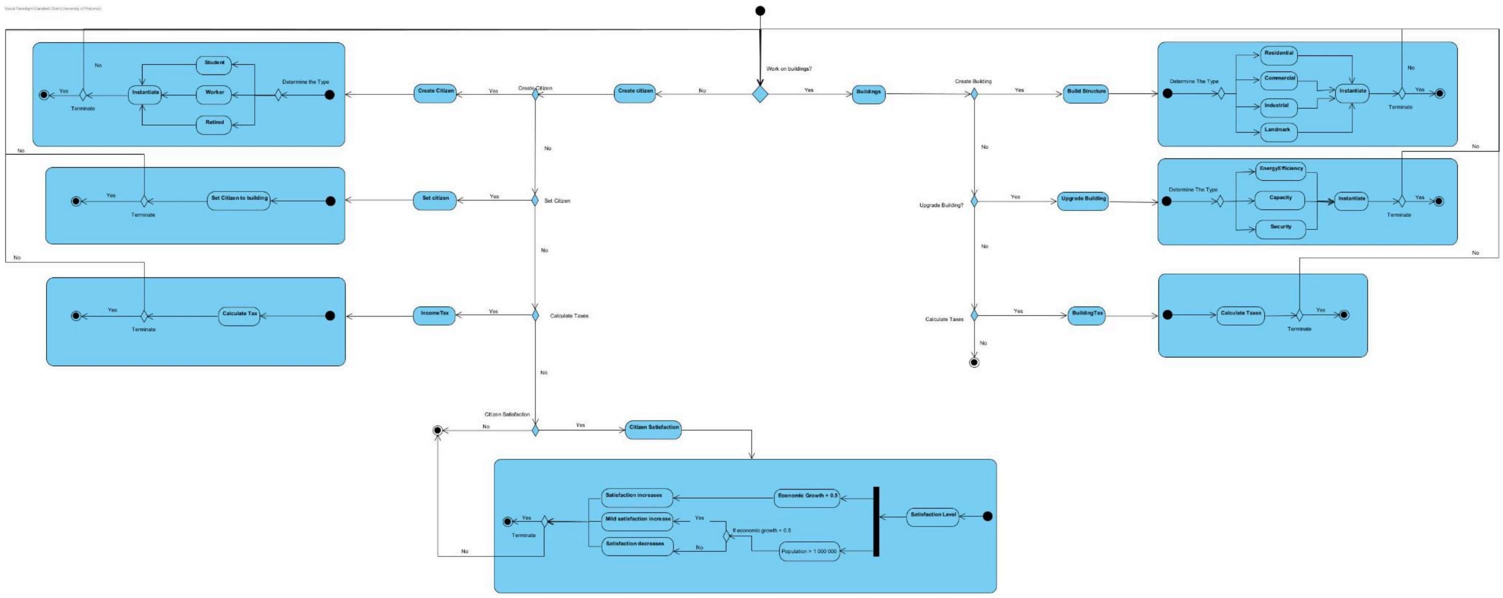


Object Diagram 1

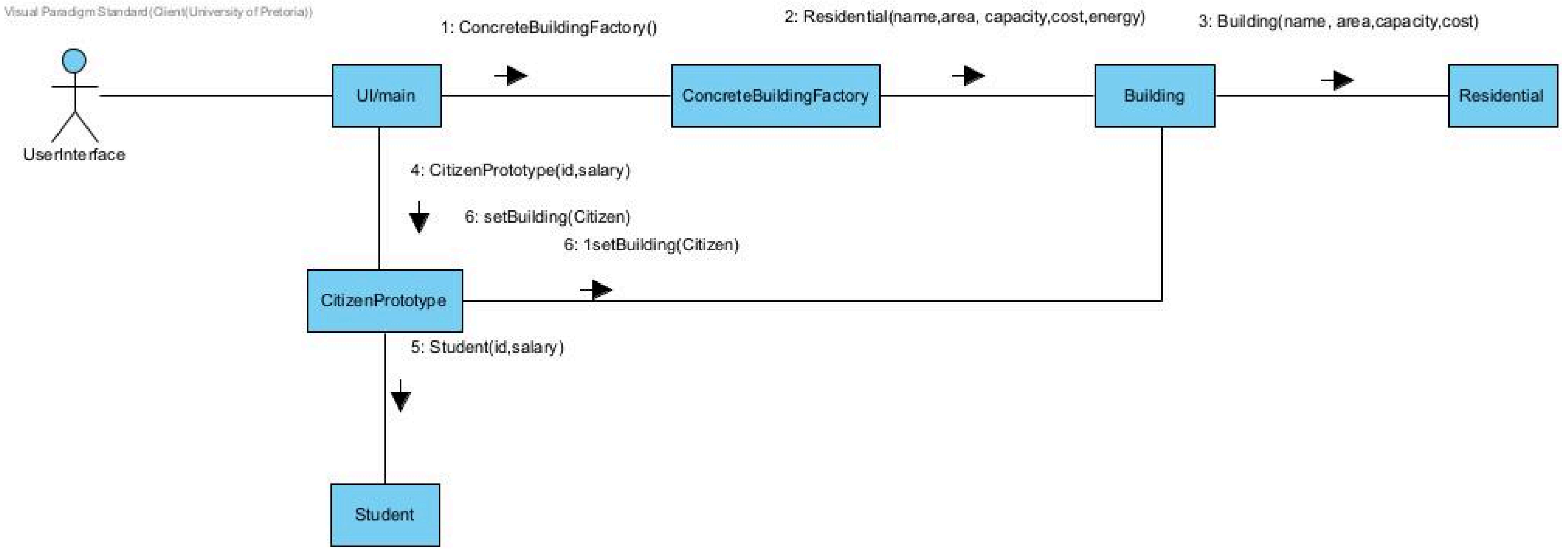




Activity Diagram

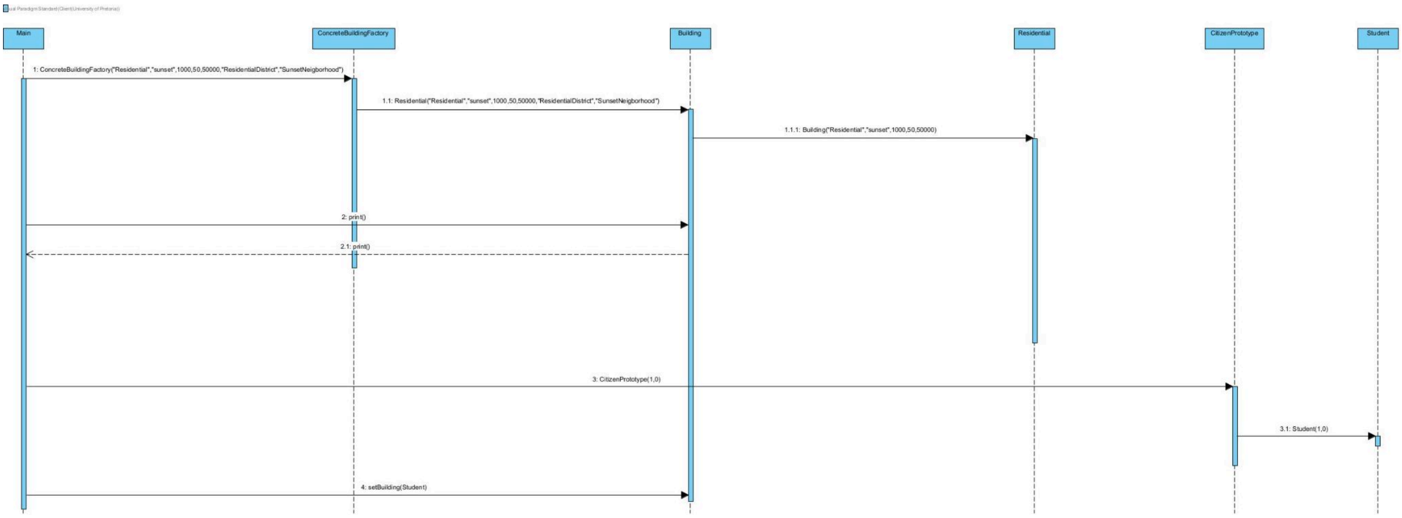


Communication Diagram



2024/11

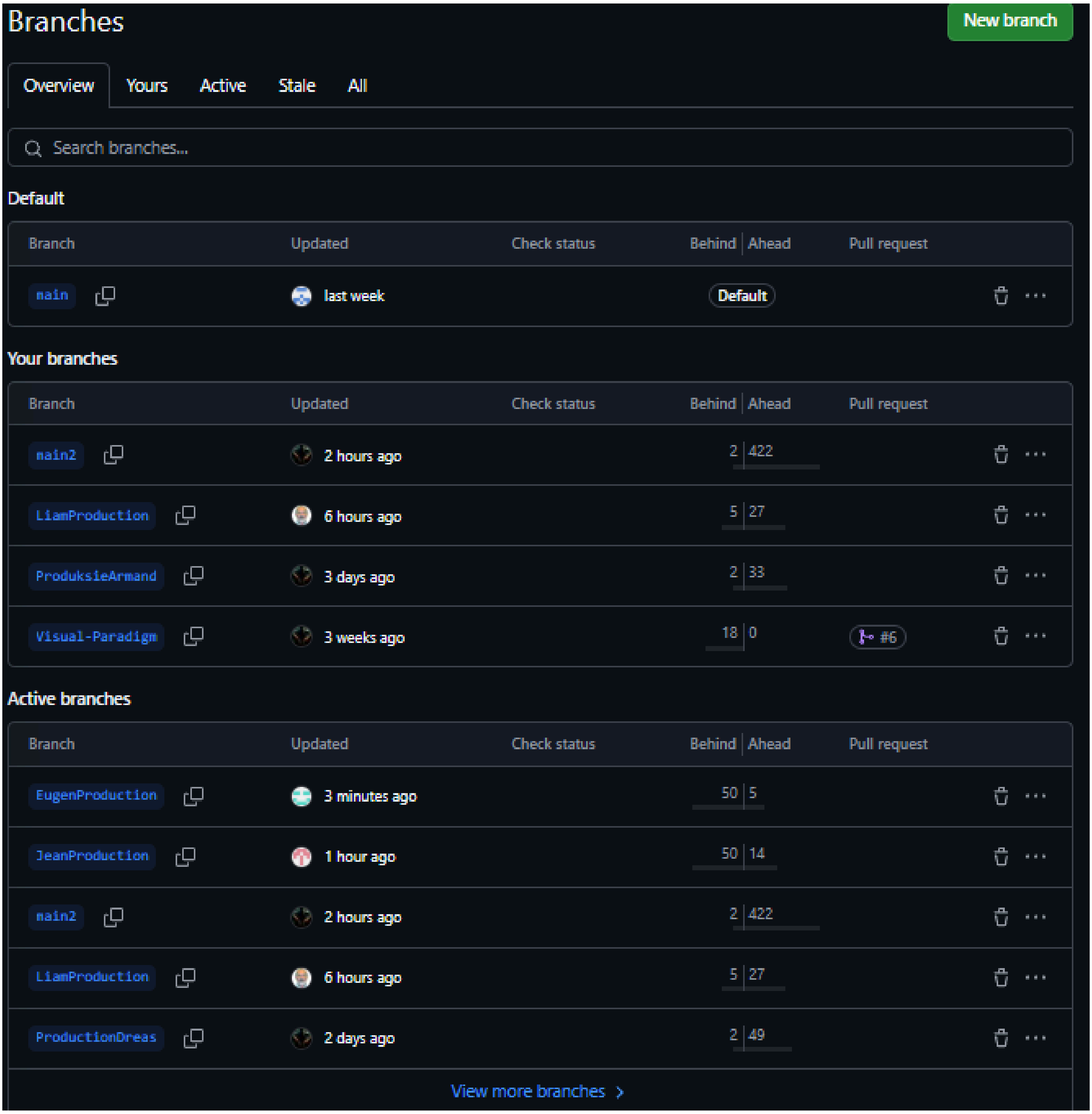
Sequence Diagram



Development Practices

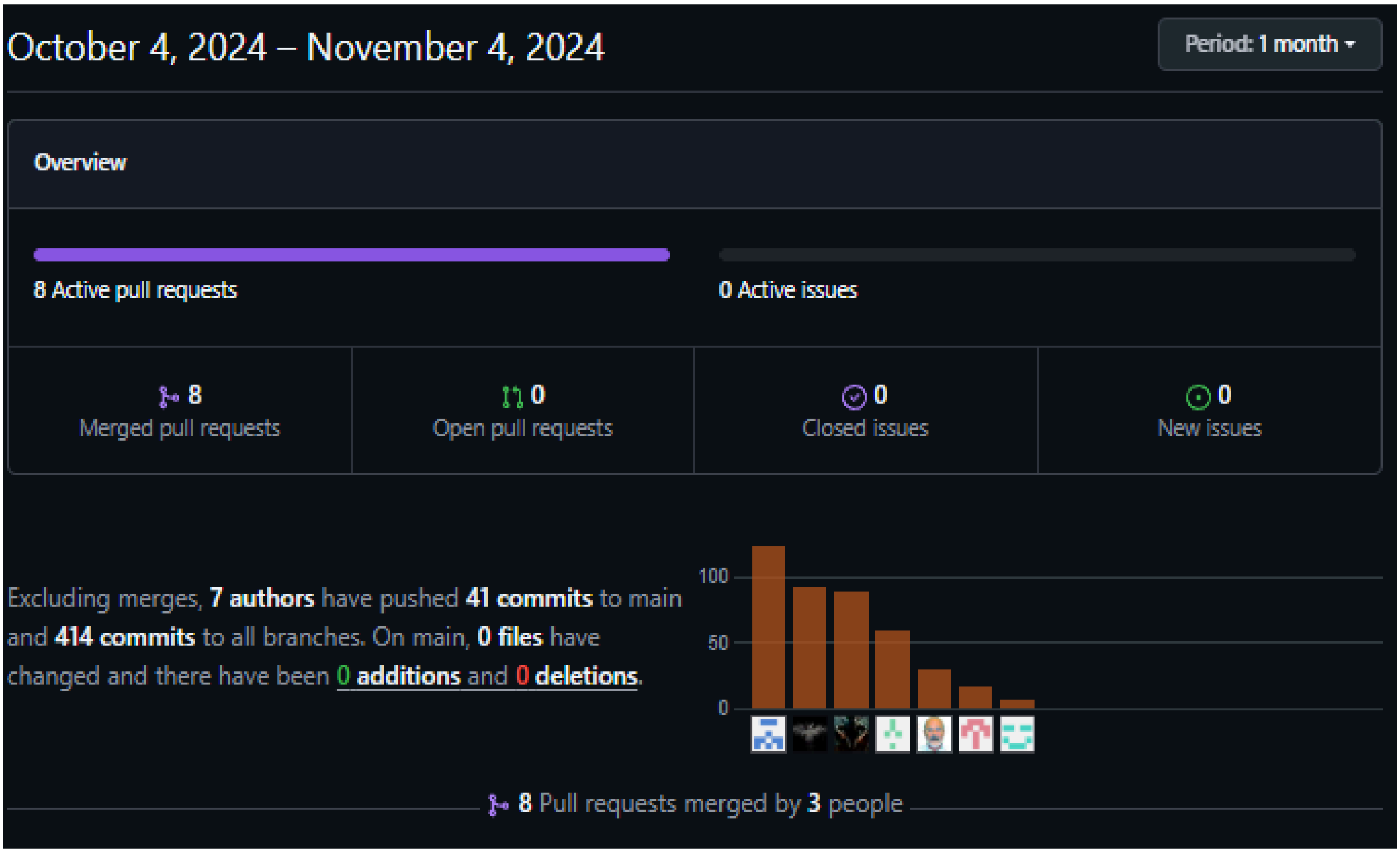
Git Workflow Strategy

Branching Strategy



Github Statistics:

– Total of 8 merge pull requests
– 414 total commits between the 7 of us
– A total of 41 commit to main from other branches



Code Documentation Standards

Class Documentation – Doxygen

City Simulation

Urban Management Simulation System

Main Page | Classes | Files

Class List

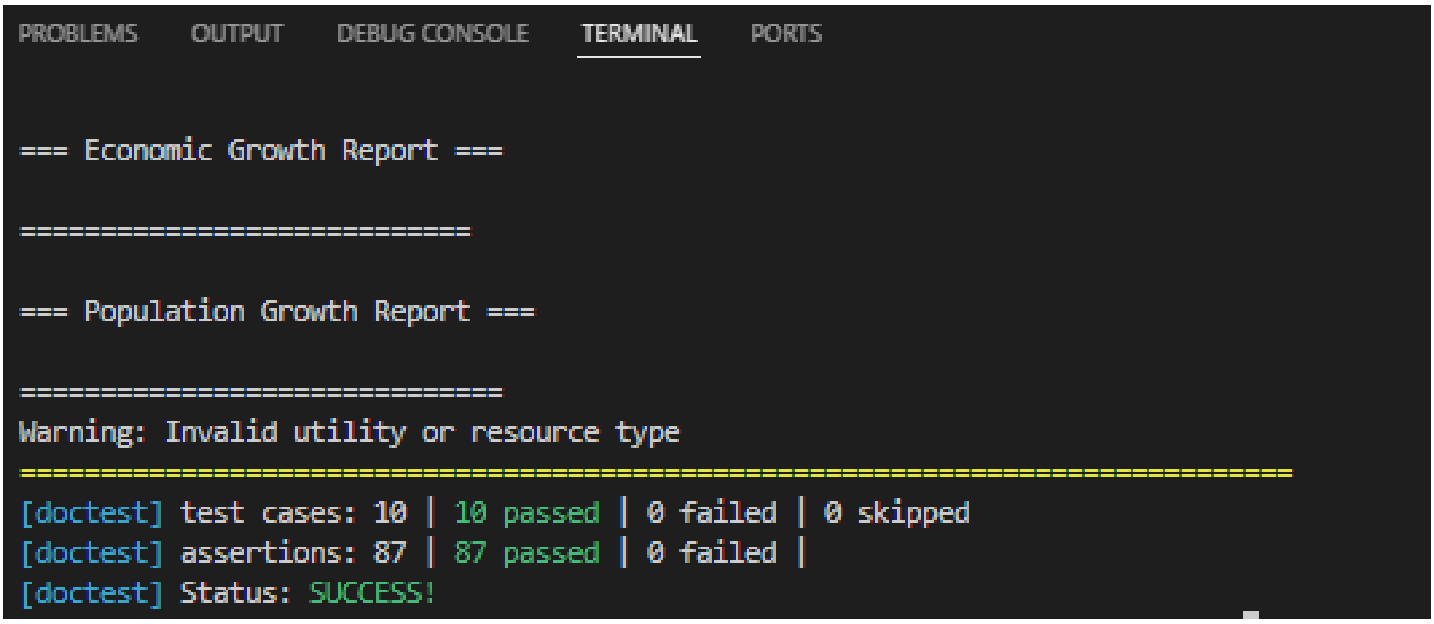
Here are the classes, structs, unions and interfaces with brief descriptions:

Abandoned	Concrete state class for abandoned buildings
BaseBuilding	Abstract base class defining core building interface
Building	Abstract base class representing a building in the city
BuildingFactory	Abstract factory class that defines interface for creating Building objects
BuildingState	Abstract base class for building states
BuildingTax	Concrete strategy for calculating building taxes
BuildingUpgrade	Decorator class that adds upgrade functionality to Building objects
CapacityUpgrade	A class that handles capacity upgrades for buildings @inherits BuildingUpgrade
CitizenPrototype	Abstract base class for different citizen types
CitizenSatisfaction	A class that monitors and calculates citizen satisfaction metrics @inherits CityObserver
City	Singleton class representing the entire city structure
CityComponent	Abstract base class for city components in a composite pattern
CityObserver	Abstract base class for city statistics observers
CityPlanner	Manages infrastructure construction and transport system integration
CityResourceDistribution	Singleton class managing resource distribution across the city
CityStats	Manages collection of city statistics observers
Commercial	Represents commercial buildings in the city
CompositeGovDepartment	A composite class that manages multiple government departments
ConcreteBuildingFactory	Concrete implementation of the BuildingFactory interface
ConcreteCityStats	Concrete implementation of city statistics tracking
District	Represents a city district containing neighborhoods
EconomicGrowth	Observer class for monitoring economic growth metrics
EducationDepartment	Represents the Education Department in the government structure
EnergyEfficiencyUpgrade	Handles energy efficiency upgrades for buildings
GovDepartment	Abstract base class representing a government department
HealthDepartment	Concrete implementation of a government health department
IncomeTax	Class implementing income tax calculation strategy
Industrial	Class representing an industrial building
Infrastructure	Abstract base class for infrastructure components
InfrastructureBuilder	Abstract builder class that defines the interface for constructing Infrastructure objects
Landmarks	Represents landmark buildings in the city
LegacyTransportSystem	Legacy transport system implementation
ModernTransport	Class representing a modern transportation system
Neighborhood	Represents a neighborhood in the city
Operational	Represents the operational state of a building
PopulationGrowth	Observer class that monitors and analyzes population growth in the city
PowerPlant	Manages power generation and distribution
PublicTransit	Represents public transit infrastructure in a city system
PublicTransitBuilder	Builder class for constructing PublicTransit objects
Residential	Represents a residential building in the city system
ResourceConsumer	Abstract interface for entities that consume resources
ResourceConsumption	Monitors and manages resource consumption in the city simulation
ResourceDistributor	Abstract interface for resource distribution
Retired	Represents a retired citizen in the city system
RoadNetwork	Represents road network infrastructure in a city system
RoadNetworkBuilder	A builder class responsible for constructing road network infrastructure @inherits InfrastructureBuilder
SecurityDepartment	Represents a security department that inherits from GovDepartment
SecurityUpgrade	A decorator class that adds security upgrades to buildings @inherits BuildingUpgrade
SewageSystem	Implementation of UtilitySystem for sewage processing
Student	Represents student citizens in the city
TaxationPolicy	Manages tax collection and budget calculation
TaxStrategy	Abstract base class for implementing various tax calculation strategies
TaxSystem	Implements the Strategy pattern for tax calculations
TransportationDepartment	Represents the Transportation Department in the government structure @inherits GovDepartment
TransportFactory	Creates and manages transport system instances
TransportSystemAdapter	Adapts a LegacyTransportSystem to work with the ModernTransport interface @inherits ModernTransport
UnderConstruction	Represents the under construction state of a building @inherits BuildingState
UtilityNetwork	Represents a utility network infrastructure system @inherits Infrastructure
UtilityNetworkBuilder	Builder class that constructs UtilityNetwork objects
UtilitySystem	Base interface class for handling and processing resources
WasteManagement	Manages waste collection and disposal
WaterSupply	Manages water distribution system

Testing Strategy

Unit Tests

Doctest was used to comprehensively test every design pattern implementation, spanning 10 total test cases which each included multiple subcases. The total test cases comprised 87 further assertions that all passed.



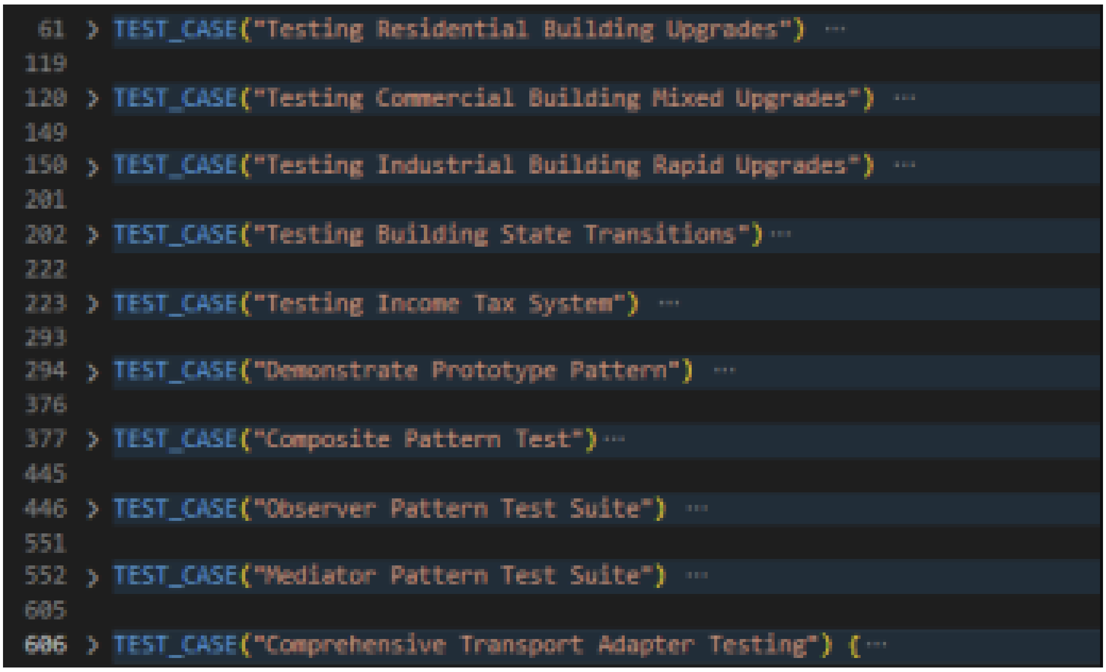
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
=== Economic Growth Report ===

============================

=== Population Growth Report ===

==============================
Warning: Invalid utility or resource type
===============================================================================
[doctest] test cases: 10 | 10 passed | 0 failed | 0 skipped
[doctest] assertions: 87 | 87 passed | 0 failed |
[doctest] Status: SUCCESS!
```



```
> TEST_CASE("Testing Residential Building Upgrades") ...

> TEST_CASE("Testing Commercial Building Mixed Upgrades") ...

> TEST_CASE("Testing Industrial Building Rapid Upgrades") ...

> TEST_CASE("Testing Building State Transitions") ...

> TEST_CASE("Testing Income Tax System") ...

> TEST_CASE("Demonstrate Prototype Pattern") ...

> TEST_CASE("Composite Pattern Test") ...

> TEST_CASE("Observer Pattern Test Suite") ...

> TEST_CASE("Mediator Pattern Test Suite") ...

> TEST_CASE("Comprehensive Transport Adapter Testing") { ...
```

GUI Implementation

A simple GUI simulation was implemented using only the SFML c++ library, a few image files and the project code. The GUI generates a dynamic map by reading from a designated map.txt file and allows the user to set their own custom map. Each ‘B’ character in the .txt file generates a map cell that allows for the creation of a specific building on that cell. A building is created by first setting the required parameters required by the factory implementation by clicking on the “Customise Building” button and then clicking on the “Build Building” button. The GUI automatically fills the available building blocks in the map until it is full and changes the display image accordingly. The parameters for new buildings can be changed at any time and the details of every building on the map can be viewed in the simulated terminal by simply clicking on the building. At any point, the strategy implementation can be employed to calculate the total building taxes of the map and display the result in the terminal. The map can also be partitioned into different districts that will automatically show up in building details.

References

World Bank, 2024. *Urban Development*. [online] Available at: https://www.worldbank.org/en/topic/urbandevelopment/overview [Accessed 4 November 2024].

Parnell, S., 2001. *Sustainable urban infrastructure planning in South Africa: A case study of Johannesburg's metropolitan development strategy*. Cities, [online] 18(6), pp.353-368. Available at: https://www.sciencedirect.com/science/article/abs/pii/S0264275101000269 [Accessed 4 November 2024].

Han, X. and Zhang, C., 2020. *Emerging strategies for urban governance in China: A study of smart cities and green development*. Procedia CIRP, [online] 89, pp.85-90. Available at: https://www.sciencedirect.com/science/article/abs/pii/S2210670720304935 [Accessed 4 November 2024].

Google doc Link:

https://docs.google.com/document/d/1BxTNnG1xQb_PBlWXuZTG1qkPb5ZKyK11NgKCVtiVu-E/edit?usp=sharing