# Research Brief on Urban Development and City Management Principles

## 1. Overview of Urban Development and Management Principles

Urban development involves managing a city's growth to accommodate population increases while ensuring sustainable resource usage and citizen well-being. Key principles include resource allocation, infrastructure maintenance, transportation efficiency, and citizen satisfaction. For instance, effective transportation systems reduce commute times, and accessible utilities (like water and power) promote high standards of living. A well-managed city also balances residential, commercial, and industrial areas to support economic and social needs.

## 2. Role of Core City Components

- **Buildings:** Different types of buildings (residential, commercial, industrial, and landmarks) serve varied purposes. Residential buildings increase population capacity, commercial zones promote economic activity, and landmarks boost citizen satisfaction. Industrial buildings, while essential for production, may affect environmental quality, which can be factored into citizen satisfaction metrics in the simulation.
- **Utilities:** Essential for basic functioning, utilities include power plants, water supply systems, and waste management facilities. The provision and reliability of these utilities directly affect citizen satisfaction and the city's ability to grow sustainably. For instance, a well-developed water supply system will support population growth, whereas power shortages could reduce productivity in commercial and industrial sectors.
- **Transportation:** Efficient transportation reduces traffic congestion and ensures goods and citizens move seamlessly across the city. Including infrastructure like roads, public transit, and airports can improve accessibility and reduce commute times, affecting citizen satisfaction positively.
- **Citizens:** Citizens drive the demand for housing, jobs, and services. Simulating population growth and satisfaction levels involves modelling factors like employment opportunities, healthcare, and education. In the simulation, these factors could be represented through Observer patterns that update citizen satisfaction based on changes in policies or infrastructure.
- **Government**: A government system manages policies, tax collection, and resource distribution. For example, adjusting tax rates impacts the

city’s budget and potentially the citizens' satisfaction. Implementing a Command pattern could help simulate the effects of government decisions on various city components.

### 3. How Research Influenced Design

This research on urban development highlights the need for modular systems where each component (buildings, utilities, citizens) interacts dynamically. Our design will use the Observer pattern for citizen satisfaction, which updates based on factors like utility availability and tax rates. The Factory pattern will be applied to building types, enabling flexible expansion as the city grows. Additionally, a Command pattern allows the government system to adjust policies and see the direct impact on other components.

### 4. Assumptions and Design Decisions

- **Assumptions:** We assume citizens’ satisfaction will primarily be influenced by resource availability, commute times, and tax levels. Also, population growth will be steady and predictable to avoid overly complex calculations.
- **Design Decisions:** In depth design decisions are discussed later in the document.

# Introduction to Design Patterns Report:

This report presents the application of specific design patterns within the City Builder Simulation, detailing how each pattern supports core functionalities across the system’s transportation and resource management components. Each pattern was chosen to enhance modularity, maintainability, and efficiency, addressing specific requirements through scalable and robust design solutions.

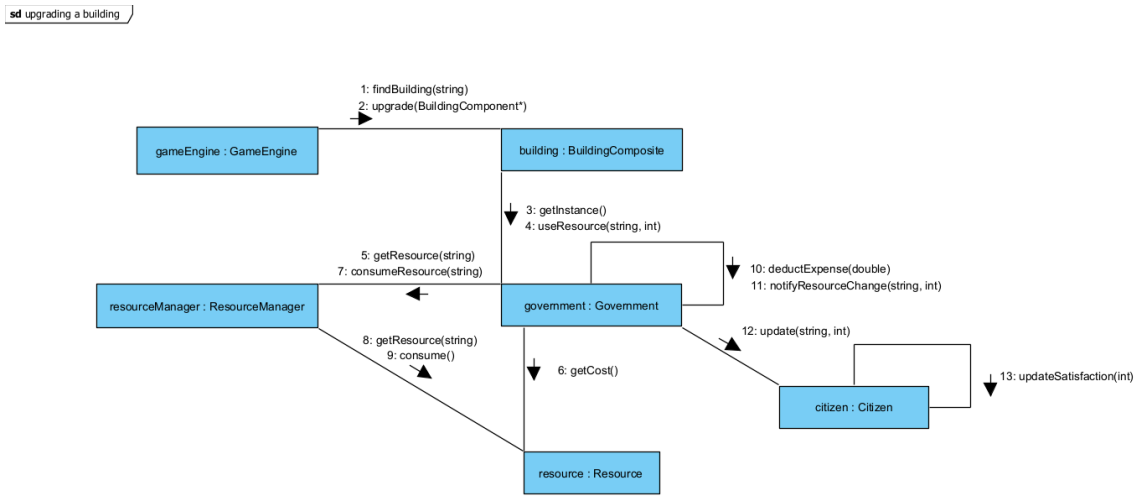
## Buildings and Utilities Component:

### Core Functionalities:

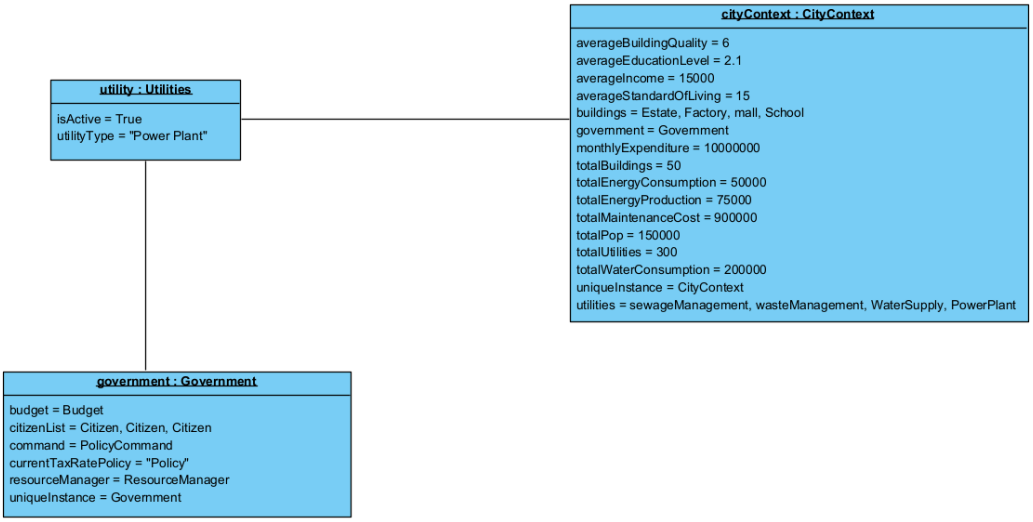
The Buildings and Utilities component of our City Builder Simulation is responsible for simulating urban structures and infrastructure systems. It provides the following essential features to enhance the city-building experience:

1. **Construction and Demolition**: Players can add new buildings and remove existing ones.

2. **Building Upgrades**: Upgrades improve building attributes like occupancy rates, production levels, or reduce operating costs.
3. **Occupancy Management**: Manages occupancy and vacancy across different building types (e.g., residents in residential buildings, businesses in commercial buildings).
4. **Resource Consumption**: Monitors and reports energy and water usage per building to the ResourceManager for accurate city-wide resource allocation.
5. **Maintenance**: Tracks maintenance costs, which affect building efficiency; unmaintained buildings incur higher costs and operate less efficiently.
6. **Revenue and Costs**: Ensures that buildings generate revenue (for commercial and industrial types) while adhering to the city's budget for construction and maintenance.




*Communication diagram illustrating how buildings are upgraded*




*Object diagram illustrating the Utilities component*

## Chosen Design Patterns:

To address the complex requirements of this component, we employed the **Factory Method** and **Composite** patterns.

## 1. Factory Method Pattern:

**Pattern Overview:**

The Factory Method pattern provides an interface for creating objects but allows subclasses to specify the exact class to instantiate. It's ideal for cases where objects fall into distinct categories, and the specific class of the object may vary based on context.

**Application in Simulation:**

- We implemented a BuildingFactory class as the primary creator. Specific building types (e.g., Residential, Commercial, Industrial) are generated through concrete factory classes, such as ResidentialBuildingFactory and CommercialBuildingFactory. Each concrete factory is responsible for creating instances of its respective building type.
- Similarly, a UtilityFactory was introduced with concrete factories for utilities, like WaterSupplyFactory, WasteManagementFactory, SewageSystemFactory, and PowerPlantFactory, each ensuring efficient instantiation of various city utilities.

**Benefits and Rationale in System:**

- **Robustness and Flexibility:** The Factory Method pattern allows for easy extension. For instance, if a new type of building or

utility is needed, adding a new factory subclass is straightforward, and existing code remains unaffected.

**Reduced Coupling:** By decoupling building creation from the components requiring buildings, this pattern lowers interdependencies within the simulation.

**Enhanced Testing and Maintenance:** Using static factory methods (instead of constructors) allows for more meaningful method names and better unit testing options since factory methods can flexibly return object subtypes.

**Why We Didn't Use Other Patterns:**
While the **Abstract Factory Pattern** could have been used, it's generally suited for cases involving families of related objects that must be created together. Here, buildings and utilities are independently created types, making the Factory Method pattern a more direct and efficient choice.

## 2. Composite Pattern:

**Pattern Overview:**
The Composite pattern organises objects into hierarchical structures, allowing individual objects and compositions of objects to be treated uniformly. This pattern is useful in managing complex structures, like interconnected building types, in the simulation.

**Application in Simulation:**

- A base BuildingComponent class was established as the foundation for all building types.
- We defined a BuildingComposite class to group buildings, allowing structures such as districts or neighbourhoods, where a group of buildings can act as a single unit. Individual buildings, created using the Factory Method, serve as leaf nodes in this hierarchy.

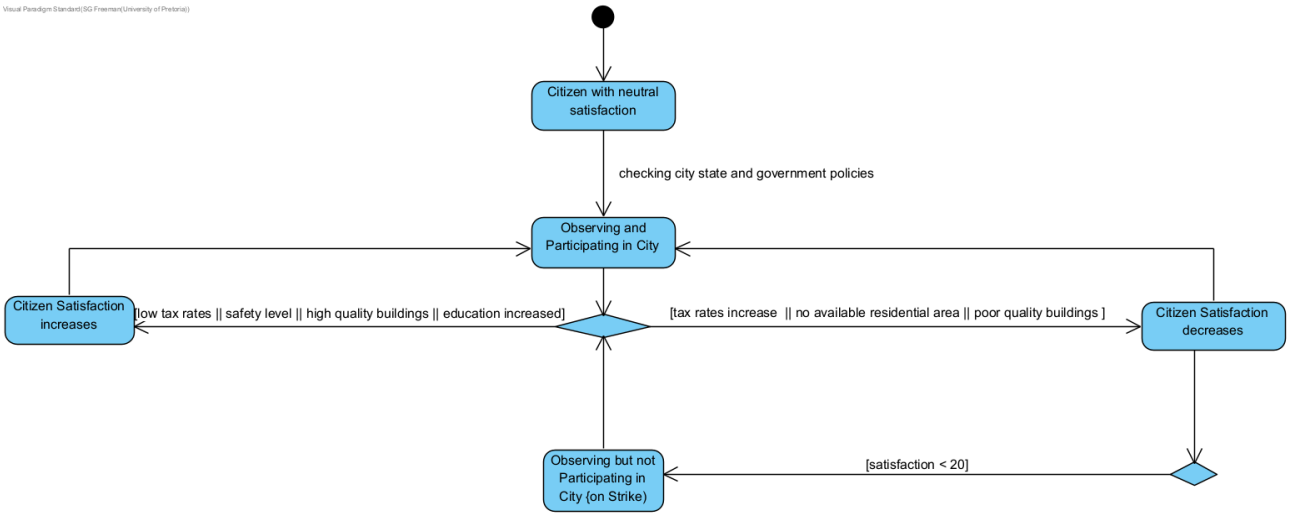
**Benefits and Rationale in System:**

- **Scalability:** The Composite pattern enhances scalability, making it straightforward to add new building types as the city grows.
- **Simplified Operations:** The hierarchical structure allows recursive operations, like calculating total resource usage or maintenance costs, by treating districts or neighbourhoods as single entities.
- **Enhanced Flexibility:** By treating individual buildings and groups of buildings uniformly, the Composite pattern simplifies code, promoting flexibility and easier maintenance.

# Citizens component:

## Core Functionalities:

The Citizen component is responsible for managing each citizen's satisfaction, employment, standard of living, and response to city policies. This component allows individual citizens and family units to observe changes within the CityContext and respond to government policies, taxes, and resource distributions. It ensures that citizen satisfaction dynamically reflects changes in their living standards, income, and job availability, while offering an organised way to manage both individuals and families as composite structures.




*Citizen Satisfaction State Diagram*

## Chosen Design Patterns:

# 1. Observer Pattern

**Pattern Overview**:

The Observer pattern establishes a one-to-many dependency, allowing changes in one object (subject) to trigger updates in all dependent objects (observers).

**Application in Simulation**:

Citizens, both individuals and families, observe the CityContext and Government. When city-wide events, policy changes, or resource adjustments occur, these observers are notified to update their satisfaction levels, tax contributions, and standard of living.

**Benefits and Rationale in System**:

Using the Observer pattern ensures real-time synchronisation between the CityContext and citizens. It simplifies city-wide updates by centralising

changes in the CityContext, which then notifies all citizen observers at once.

**Why We Didn't Use Other Patterns**:
An alternative could have been the Mediator pattern for managing citizen-Government interactions. However, the Observer pattern is more effective here due to its straightforward, centralised notification system without adding unnecessary complexity.

## 2. Composite Pattern:

**Pattern Overview**:
The Composite pattern allows treating individual objects and groups of objects (composites) uniformly, enabling hierarchies where objects can contain other objects.

**Application in Simulation**:
Families are composite structures, grouping individual citizens and allowing them to act as single units. For example, families have aggregated values for satisfaction and standard of living, which can influence district demographics.

**Benefits and Rationale in System**:
The Composite pattern simplifies handling families within the city, as it lets families inherit attributes like satisfaction and standard of living. This makes citizen management flexible and hierarchical, accommodating both single citizens and family units under the same structure.

**Why We Didn't Use Other Patterns**:
While an alternative like the Decorator pattern could handle family attributes, it lacks the inherent hierarchical organisation of the Composite pattern, which is essential for managing nested structures like families.

## 3. Memento Pattern

**Pattern Overview**:
The Memento pattern captures an object's internal state, allowing it to be restored later without exposing its details.

**Application in Simulation**:
The CityContext utilises Memento to save snapshots of its state at given points, through SavePoint as the memento and SavePointManager as the

caretaker. This provides functionality for undo/redo, enabling the city's state to revert to previous configurations.

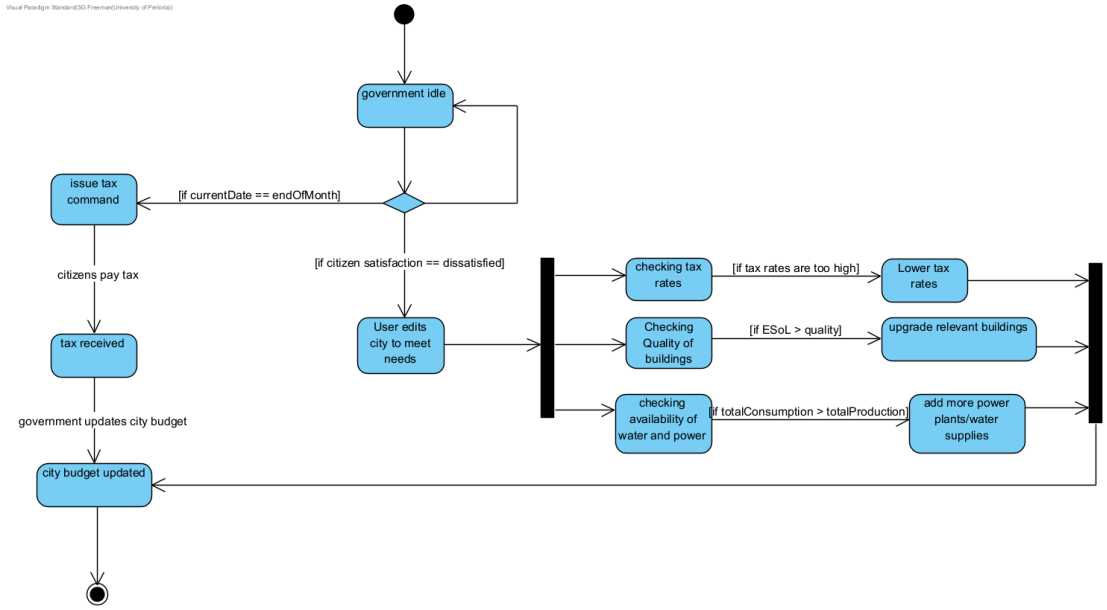
**Benefits and Rationale in System**:
Memento enables historical tracking and restoration of city states, which is essential for simulations needing "revert" functionalities, such as in the case of policy rollbacks or citywide event resets.

**Why We Didn't Use Other Patterns**:
An alternative pattern, such as Command, could handle actions on the city's state, but it wouldn't manage state restoration as effectively as Memento. The primary aim here is to store and revert state, a strength of the Memento pattern.

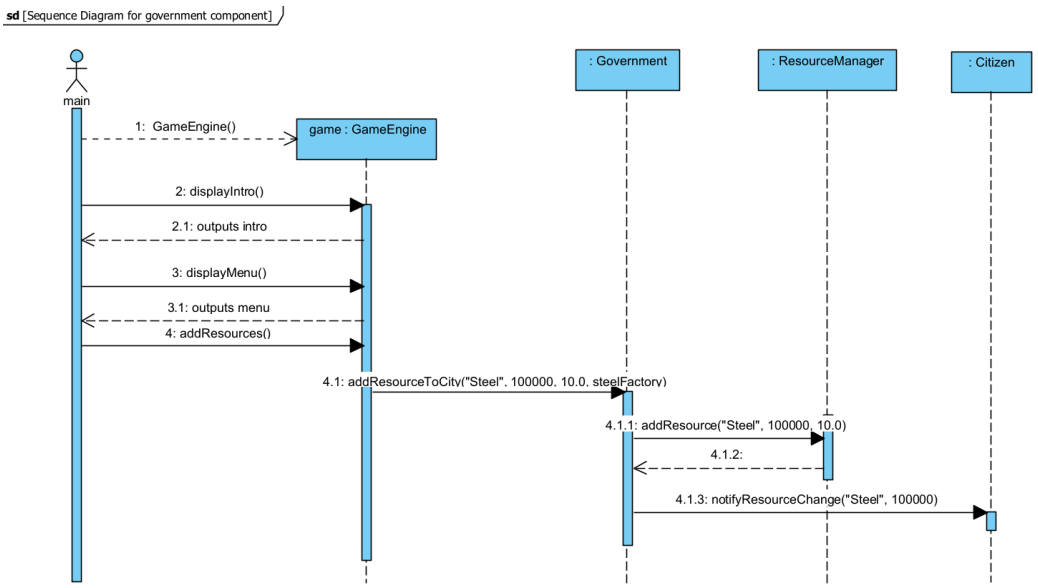
# Government Component:

## Core Functionalities:

The Government component serves as the central authority overseeing city operations. It manages critical aspects of city governance, including financial control, policy enforcement, resource allocation, and citizen satisfaction. As a central mediator, it communicates with various city components like citizens, resources, and budgets, ensuring that city operations align with policies and economic goals.




*Government State Diagram*




*Sequence diagram illustrating how the Government component works.*

## Chosen Design Patterns:

### 1. Singleton Pattern:

- **Pattern Overview**:

  Ensures that only one instance of a class exists and provides a global point of access to it.

**Application in Simulation**:

The Government component acts as a Singleton, as it needs a unique and consistent authority to handle city policies, resource allocations, and budget controls.

**Benefits and Rationale in System**:

- The Singleton pattern centralises control, ensuring that policy changes, resource allocations, and budget adjustments are consistent across the entire city.
- Reduces the risk of multiple conflicting government instances that might result in inconsistent policies or tax calculations.

**Why Not Other Patterns?**

Other patterns like Factory or Prototype were not suitable, as the Government requires a single instance with complete control rather than multiple, unique instances.

### Command Pattern

- **Pattern Overview**:
  Encapsulates a request as an object, allowing parameterization of clients with queues, logs, or undoable operations.

**Application in Simulation**:

- The Government class serves as the Invoker, issuing commands that encapsulate requests for taxation or policy enforcement.
- PolicyCommand and TaxationCommand act as Concrete Commands, while the CityContext and AbstractCitizen classes serve as Receivers that perform the requested actions.

**Benefits and Rationale in System**:

- Facilitates flexibility in managing diverse government commands, such as varying tax policies or policy adjustments, without directly modifying Government code.

- Enables structured, decoupled control over citizen interactions and resource policies.

- **Why Not Other Patterns?**
  Other behavioural patterns like Mediator or Observer were considered but don't allow direct encapsulation of requests as standalone commands, which would complicate command execution and history management.

## Visitor Pattern

- **Pattern Overview**:
  Allows new operations to be defined on elements without changing their classes, promoting extensibility.

- **Application in Simulation**:
  - TaxationCommand utilises the Visitor pattern by interacting with a TaxCollector visitor to apply taxation operations across all citizens.
  - Citizens (Concrete Elements) accept visits from TaxCollector, enabling flexible tax application without modifying the Citizen class.

- **Benefits and Rationale in System**:
  - Adds flexibility in applying different tax calculations or exemptions to citizens based on visitor logic.

- Decouples taxation calculations from citizen classes, supporting future extensibility in tax operations.

- **Why Not Other Patterns?**
  Patterns like Strategy or Chain of Responsibility were considered but wouldn't allow direct traversal of citizen lists with context-specific taxation behaviour.

# Taxation Component

## Core Functionalities:

The Taxation System manages the city's revenue generation by calculating and collecting taxes from citizens. It ensures that the income generated by citizens contributes to the government's budget, supporting financial stability and resource management for the city. It interacts closely with the Government and Citizen classes to apply, collect, and potentially adjust taxes over time.

## Chosen Design Patterns:

### Command Pattern

**Pattern Overview**:
Encapsulates requests as objects, allowing for parameterized actions and delayed execution, particularly useful in queuing and logging.

**Application in Simulation**:

- The TaxationCommand represents a Concrete Command, which calculates and applies taxes on citizens. The Government (Invoker) issues the TaxationCommand, and the TaxCollector executes it, ensuring that taxes are collected from all citizens and added to the city's budget.

**Benefits and Rationale in System**:

- Decouples the tax calculation and application logic from the Government, providing flexibility in adjusting and executing tax commands as city needs change.
- Allows for the reusability of tax operations, enabling tax adjustments and collection schedules without altering the Government's code.

**Why Not Other Patterns?**

Other options like Observer or Strategy were considered, but they lack the encapsulation and flexibility provided by Command for queueing, scheduling, and execution.

## Visitor Pattern

- **Pattern Overview**:

Allows operations to be performed on elements in a structure without changing their classes, supporting flexible operations across elements.

**Application in Simulation**:

- The TaxationCommand uses the Visitor pattern by invoking the TaxCollector visitor to apply taxes across citizens. Each Citizen (Concrete Element) accepts the visitor, facilitating tax deductions.

**Benefits and Rationale in System**:

- The Visitor pattern separates the tax operation from the Citizen class, providing flexibility in applying various tax policies without changing the Citizen structure.
- Enables different taxation policies and calculation methods to be applied uniformly across citizens.

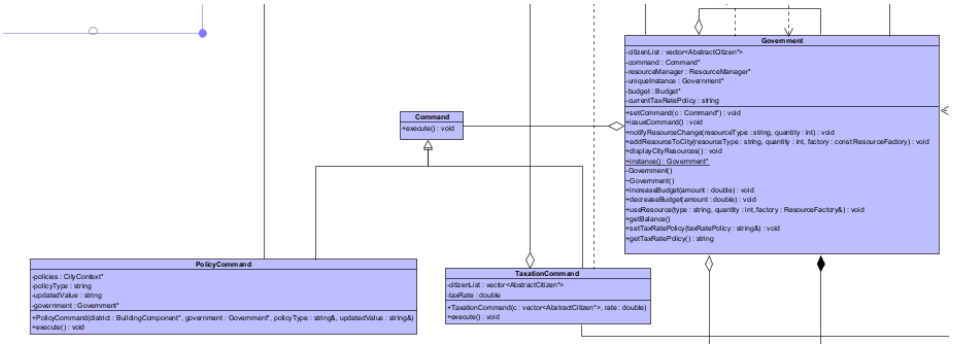
**Why Not Other Patterns?**

Alternatives like Strategy would necessitate changes within the Citizen class itself, which would reduce flexibility in adding new taxation policies.

# Policy Command Component

## Core Functionalities :

The Policy Command System enforces city-wide rules and policies, impacting buildings, resources, and citizens. It is primarily responsible for implementing and enforcing policies issued by the Government, such as resource limitations or regulations. Policies issued can range from factory restrictions to environmental regulations, influencing various city districts and components.




*Snippet of the Policy Command in the class diagram*

## Chosen Design Patterns:

### Command Pattern:

- **Pattern Overview**:
  Encapsulates requests as objects, allowing them to be stored, passed, or delayed.

- **Application in Simulation**:
  PolicyCommand acts as a Concrete Command, which allows the Government to issue various policies across city districts. This command is executed by affecting the relevant buildings and resources as defined in BuildingComposite.

- **Benefits and Rationale in System**:
  Using Command provides flexibility to store and execute policy commands as needed, allowing the Government to control policy implementation without directly handling resource allocation. Ensures scalability, as multiple types of policies can be issued and managed independently.

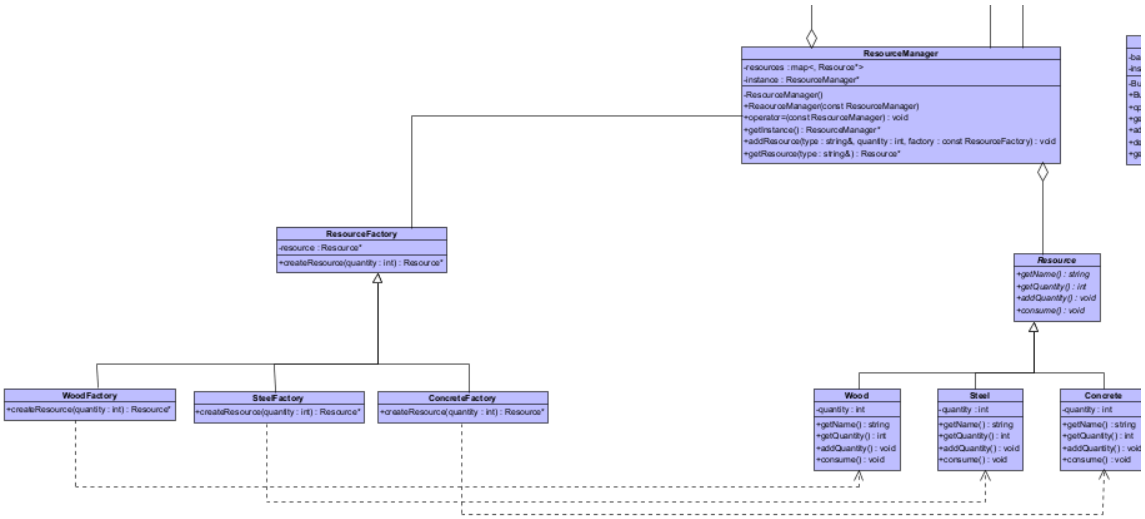
- **Why Not Other Patterns?**
  Alternative patterns like Mediator or Strategy wouldn't provide the same level of encapsulation and structured execution, which is essential for enforcing policies at the city-wide level.

## Resources Component:

### Core Functionalities :

The Resources Component enables the creation of specific resources (e.g., Wood, Steel) through factories, allowing easy extension for new types. It centralises resource and budget management, tracking availability and

dynamically allocating funds across the city. A single instance manages resource usage, while the budget system adjusts based on city needs. Additionally, the component alerts relevant sectors when resources or budget levels are low to prevent disruptions.




*Snippet of the ResourceFactory in the Class diagram*

## Chosen Design Patterns:

# 1. Factory Pattern:

- **Pattern Overview**:
  The Factory Pattern provides a way to instantiate objects without specifying their concrete classes, allowing for flexible and scalable resource creation.

- **Application in Simulation**:
  - **Resource Creation**: Concrete factories like WoodFactory, SteelFactory, and ConcreteFactory implement the ResourceFactory interface to create specific resources. Each factory instantiates a resource type based on simulation needs, such as quantity and type.

- **Benefits and Rationale in System**:
  - **Scalability**: New resource types can be added by creating additional factory classes without changing existing ones.
  - **Maintenance**: Reduces complexity by encapsulating creation logic in factory classes, making the system easier to maintain.

- **Why Other Patterns Were Not Used**:

**Abstract Factory**: Since each resource type (e.g., Wood, Steel) needs only one factory, the Factory Pattern is sufficient without the added complexity of an Abstract Factory. Also, there are no related families of resources that would justify this approach.

## 2. Singleton Pattern

- **Pattern Overview**:

The Singleton Pattern ensures a class has only one instance and provides a global point of access to it. This is particularly useful for managing resources and budget centrally.

- **Application in Simulation**:
  - **Resource Management**: ResourceManager is a singleton that tracks all resources, ensuring consistent access and updates across the city.
  - **Budget Management**: BudgetManager is also a singleton, managing the city's financial resources, handling income and expenses, and facilitating budget allocations.

- **Benefits and Rationale in System**:
  - **Consistency**: Ensures all city components access the same instance of ResourceManager and BudgetManager, preventing discrepancies in resource or budget availability.
  - **Centralised Control**: Simplifies resource and budget tracking by consolidating these operations within single instances.

- **Why Other Patterns Were Not Used**:
  - **Mediator Pattern**:

While this pattern could facilitate communication between components, it would add complexity without substantial benefits, as the primary need here is for single access points, not interaction management.

- **Observer Pattern**:

Though it could be used for notifying components of resource shortages, the overhead of implementing observers on a large scale could impact performance. A simpler notification mechanism was deemed more suitable for this project.

# Transportation Component:

## Core Functionalities:

The Transportation subsystem in the City Builder Simulation manages citizen travel, commute times, and traffic, adapting transportation strategies dynamically to optimise efficiency. Core functionalities include travel management, strategy selection, and real-time traffic monitoring. Travel management calculates commute times by considering traffic and connectivity, while dynamic strategy selection uses the Strategy Pattern to choose the best mode—road-based, rail-based, or public transit—based on current traffic and road conditions. The TrafficManager monitors congestion and helps select optimal routes, influencing citizen satisfaction by aiming to minimize travel time. Key classes include Transportation, which oversees strategy selection and commute calculation, TrafficManager for real-time monitoring, and various strategy classes like RoadBasedStrategy, RailBasedStrategy, and PublicTransitStrategy. This flexible design ensures maintainability and scalability, allowing smooth adaptation to changes in traffic and infrastructure.

## Chosen Design Pattern:

# 1. Strategy Pattern:

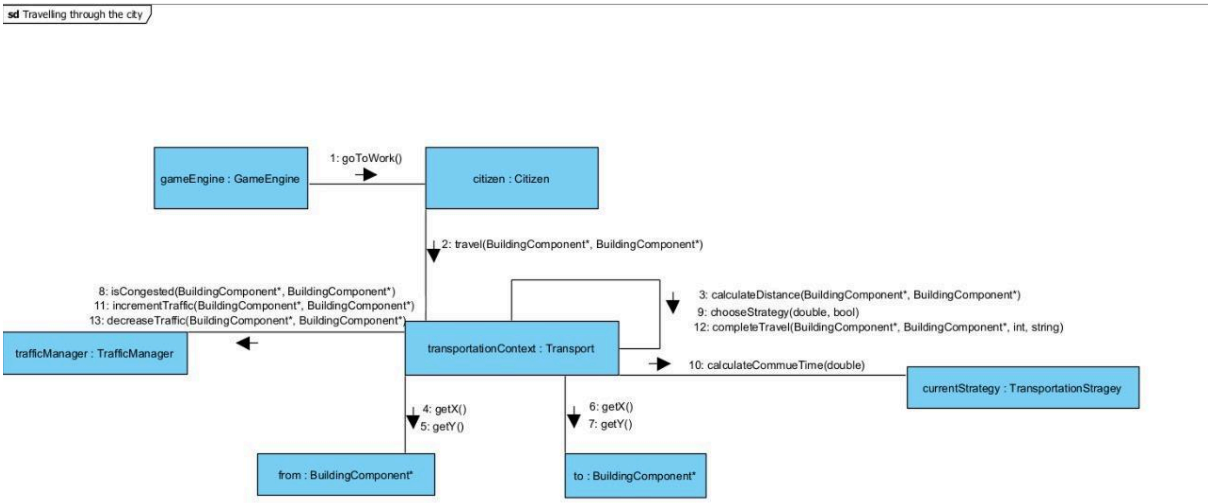
- **Pattern Overview**:
  The Strategy Pattern enables the system to dynamically switch between transportation modes based on traffic data and road conditions, optimising for efficiency.
- **Application in Simulation**:
  Strategies like road-based, rail-based, and public transit-based modes implement a common interface, allowing the Transportation class to select a strategy based on commute distance and congestion.

- **Benefits and Rationale**:
  This pattern allows the system to be easily extendable as new strategies can be added without altering existing code. It also improves adaptability, switching strategies as conditions change.

- **Why Not Other Patterns**:
  - Other patterns, like the State Pattern, could handle changes in behaviour, but they do not provide the flexibility to choose a strategy independent of the current "state" of traffic, which makes Strategy a better fit.

- The Observer Pattern was considered for real-time traffic updates but was ultimately not used. Instead, traffic monitoring is managed directly

by the TrafficManager, which keeps a map of buildings and maintains an association with the Transportation class. This design choice keeps the subsystem efficient without the added complexity of multiple observers, as the TrafficManager alone handles congestion monitoring and reporting effectively.




*Sequence diagram illustrating the transportation system*

# Game Engine Component:

## Core Functionalities:

The Engine Control Component acts as the central manager for the city's complex systems, streamlining interactions across all major subsystems. This component facilitates real-time and turn-based events within the simulation, including construction, resource allocation, taxation, and the needs of citizens. By centralising control, it ensures seamless coordination across city systems, delivering a cohesive and realistic simulation experience.

## Chosen Design Patterns:

### ○ Facade Pattern

- **Pattern Overview**:

  The Facade Pattern provides a unified interface that simplifies complex system interactions by encapsulating access to multiple subsystems. This pattern is particularly useful for abstracting

and managing various functionalities within the city, promoting easier control and maintenance.

**Application in Simulation**:

- **Engine Facade**: This high-level interface interacts with key subsystems such as Taxation, Resources, and Transportation, coordinating actions without exposing internal details. It allows for smooth control over various city components, making it easier to manage event-driven and simulation-based operations.

**Benefits and Rationale in System**:

- **Simplicity**: By consolidating the controls of diverse subsystems, the Facade Pattern reduces overall system complexity. This makes interactions across components straightforward and provides an efficient entry point for managing the entire engine.
- **Scalability**: This pattern supports easy integration of new features or subsystems, enabling future expansion without modifying the entire control structure. Through the Facade, we can introduce new functionality without exposing the underlying subsystems to changes.

**Why Not Other Patterns?**

- **Proxy Pattern**: The Proxy Pattern could help control access to individual parts of the system, but it doesn't provide the broad, centralized control needed to manage the engine smoothly.
- **Command Pattern**: The Command Pattern is good for organizing actions, but it would make the system more complex than needed. The Facade Pattern is a better choice because it keeps control simple and unified across all city functions.

## Conclusion:

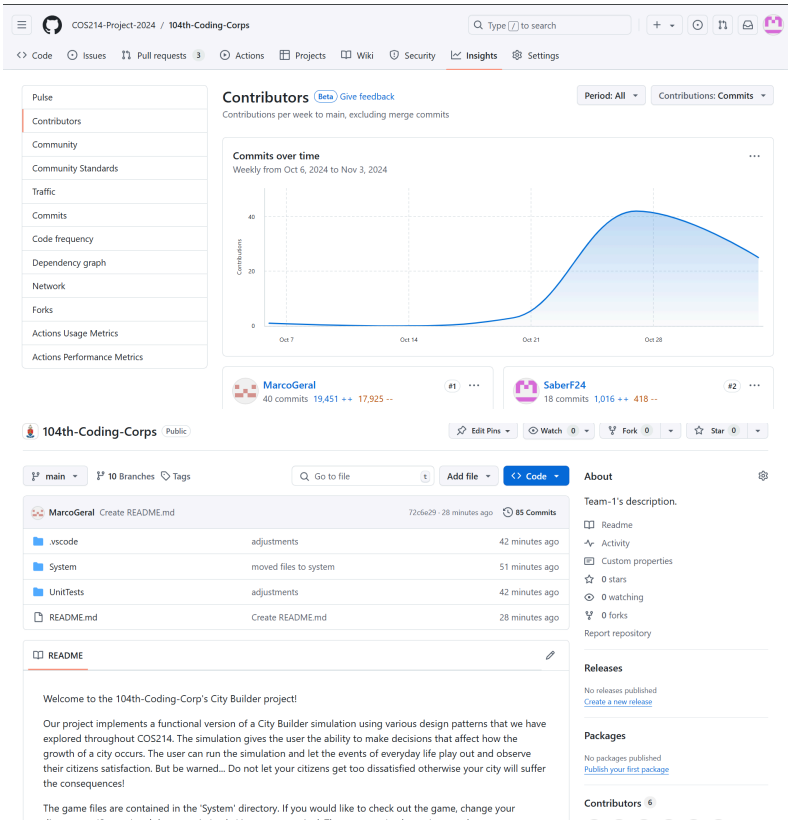
The design patterns implemented in the City Builder Simulation ensure adaptable, efficient handling of complex processes, providing a solid foundation for future expansion and improved system performance.

Link to Google Doc:

[Google Doc]

## GitHub:

Link to repository: https://github.com/COS214-Project-2024/104th-Coding-Corps.git




# References


- Lai, Shih-Kung. (2017). City Management: Theories, Methods, and Applications (Book Proposal). 10.13140/RG.2.2.32857.01127
- Bibri, S.E., Krogstie, J. and Kärrholm, M. (2020). Compact City Planning and Development: Emerging Practices and Strategies for Achieving the Goals of Sustainable Development. Developments in the Built Environment, [online] 4(1), p.100021. doi:https://doi.org/10.1016/j.dibe.2020.100021