City Builder System - Report Brief

Core Urban Development Components

• Transportation needs and traffic flow
• Public utilities (power, water) management
• Housing requirements for citizens
• Healthcare and education facilities
• Government policies and tax
• Budget allocation and financial constraints
• Economic growth indicators
• Population welfare metrics
• Crime and punishment

Design Patterns and Implementation

Infrastructure Management: The Factory Method pattern is used in the `BuildingFactory` class to simplify the creation of various types of buildings, such as houses, factories, offices and more. The State design pattern is used to manage the utilities of each building.

Economic Systems: The Observer pattern is utilised in the `CitizenObserver` and `CityObserver` classes. This allows the city manager to monitor and respond to changes in citizen welfare metrics, such as healthcare, education, and crime rates, ensuring the necessary responses are made. The Memento design pattern is also used to store and compare city metrics to previous years, ensuring economic growth over time.

Social Services: The State pattern and Chain of Responsibility patterns are used in the `CitizenMood` and `ComplaintHandler` classes to monitor citizen moods and complaints. This ensures that social welfare is met, by monitoring the needs of citizens.

Crime Punishment: The Strategy pattern is implemented in the `CrimePunishmentStrategy` class, enabling the dynamic adjustment of crime punishment, so that the punishment fits the crime. Ranging from community service to a death sentence.

Metric Monitoring: The Visitor design pattern is used to generate reports on the city metrics and structures, calculating various statistics based on the `City` class.

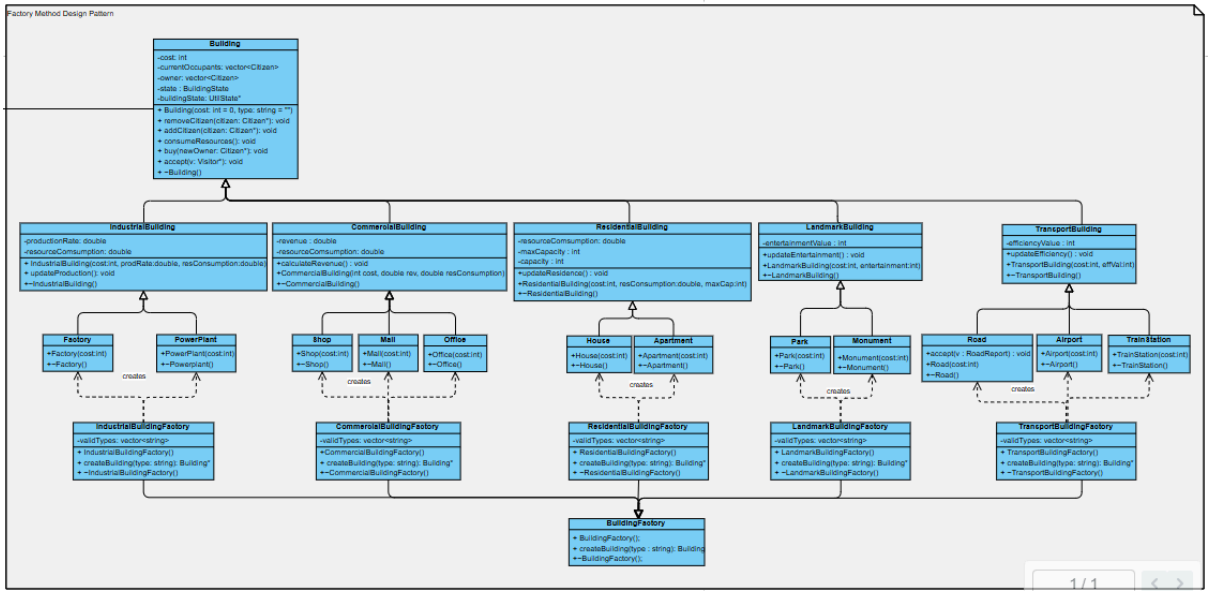
Menu Management: The Command design pattern is used to manage the functionality behind the user interface and allow the client to make a range of requests regarding city management, such as changing policies, adding buildings, exiting the program, etc. The Command design pattern manages the functionality of the other design patterns.

Taxation: The Command design pattern is used in the `AdjustTaxCommand` class to modify the tax state of the city. The Singleton pattern is used in the `TaxSystem` class, ensuring a centralised point of control and coordination for the tax needs of the city.

The Template Method pattern is utilised in the `Visitor` class and its subclasses, defining a common structure and workflow for generating various types of city reports, while allowing for customization in the specific reporting logic.

By leveraging these 10 design patterns, our city builder program maintains a high degree of modularity, flexibility, and scalability. This architectural approach enables us to effectively manage the intricate relationships and dynamics of the urban environment, while also providing a foundation for future expansion and adaptation to evolving city management requirements.

Design Pattern Analysis for City Simulator



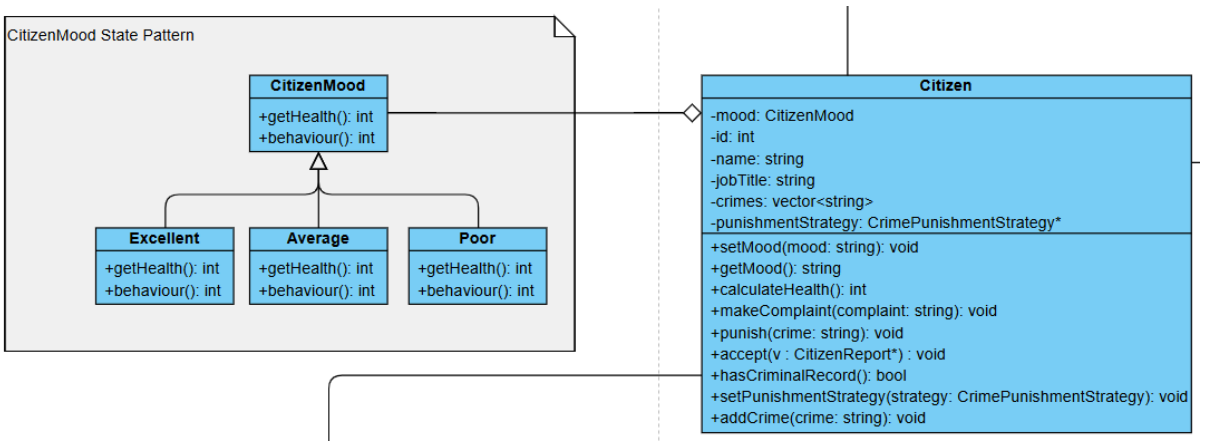
Factory Method Pattern

Implementation Context

- Used for creating different types of buildings (Residential, Commercial, Industrial, etc.)
- Hierarchical structure of building types and their factories

Why It's a Good Fit

1. **Extensibility**
    - Easy to add new building types without modifying existing code
    - Each building category has its own factory, making maintenance simpler
    - Supports future expansion (new building types, categories)
2. **Encapsulation**
    - Building creation logic is isolated in factory classes
    - Concrete building details are hidden from the main system
    - Makes testing and modification easier
3. **Standardisation**
    - Ensures all buildings follow the same interface
    - Maintains consistency in building creation
    - Simplifies building management



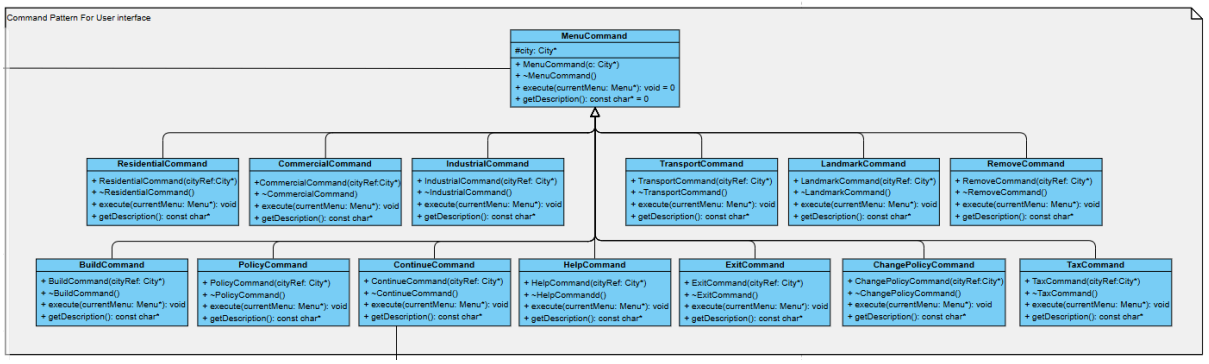
State Pattern (Citizen Moods)

Implementation Context

- Manages citizen moods (Excellent, Average, Poor)
- Affects citizen behaviour and city metrics

Why It's a Good Fit

1. **Dynamic Behavior**
    - Citizens' moods change based on city conditions
    - Different behaviours for different moods
    - Natural state transitions
2. **Simplified Logic**
    - Mood-specific behaviour is encapsulated
    - Clear transition rules
    - Easy to modify mood effects
3. **Scalability**
    - Additional moods can be added easily
    - Mood system can be extended without affecting other systems
    - Clear interface for mood-dependent behaviours



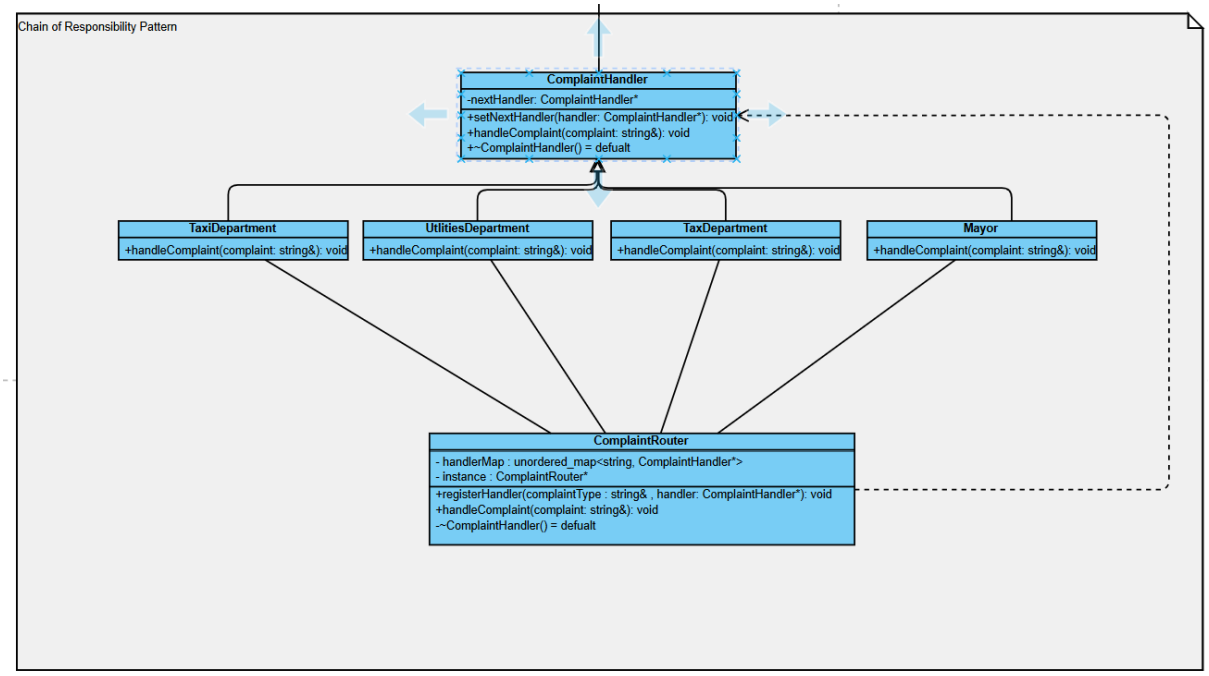
Command Pattern (UI)

Implementation Context

- Handles user interactions and menu system
- Manages different game commands

Why It's a Good Fit

1. **User Interface Flexibility**
    - Easy to add/remove commands
    - Commands can be composed and reused
    - Supports undo/redo (if implemented)
2. **Separation of Concerns**
    - UI logic is separated from game logic
    - Each command is self-contained
    - Easy to modify individual commands
3. **Extensibility**
    - New commands can be added without changing existing code
    - Commands can be grouped and organised
    - Supports complex command sequences



Chain of Responsibility Pattern (with Router)

Implementation Context

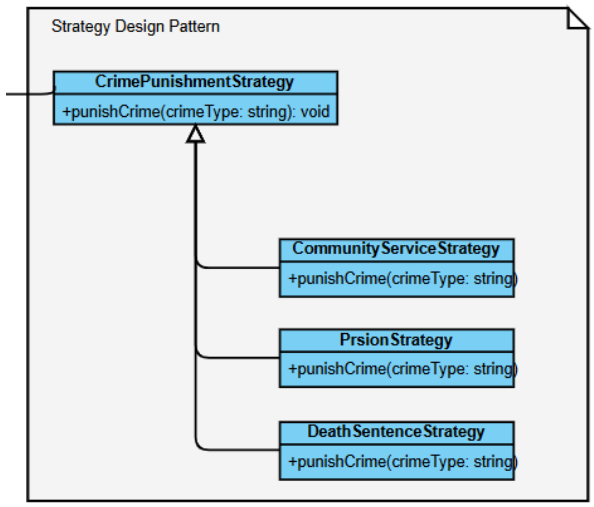
The pattern is implemented with two key components:

1. **ComplaintRouter (Singleton)**
    - Central complaint management using unordered_map
    - Registers and routes complaints to appropriate handlers
    - Static instance for global access
2. **Complaint Handlers Hierarchy**
    - Base ComplaintHandler with nextHandler pointer
    - Specialised departments:
        - TaxiDepartment

        - UtilitiesDepartment
        - TaxDepartment
        - Mayor (final handler)

Why It's a Good Fit

1. **Flexible Routing**
    - ComplaintRouter allows dynamic registration of handlers
    - Complaints can be directly routed to appropriate department
    - Easy to add new departments without changing existing code
2. **Centralised Management**
    - ComplaintRouter provides single point of complaint management
    - Handlers can be easily registered/unregistered
    - Clear flow of responsibility



Strategy Pattern (Crime Punishment)

Implementation Context

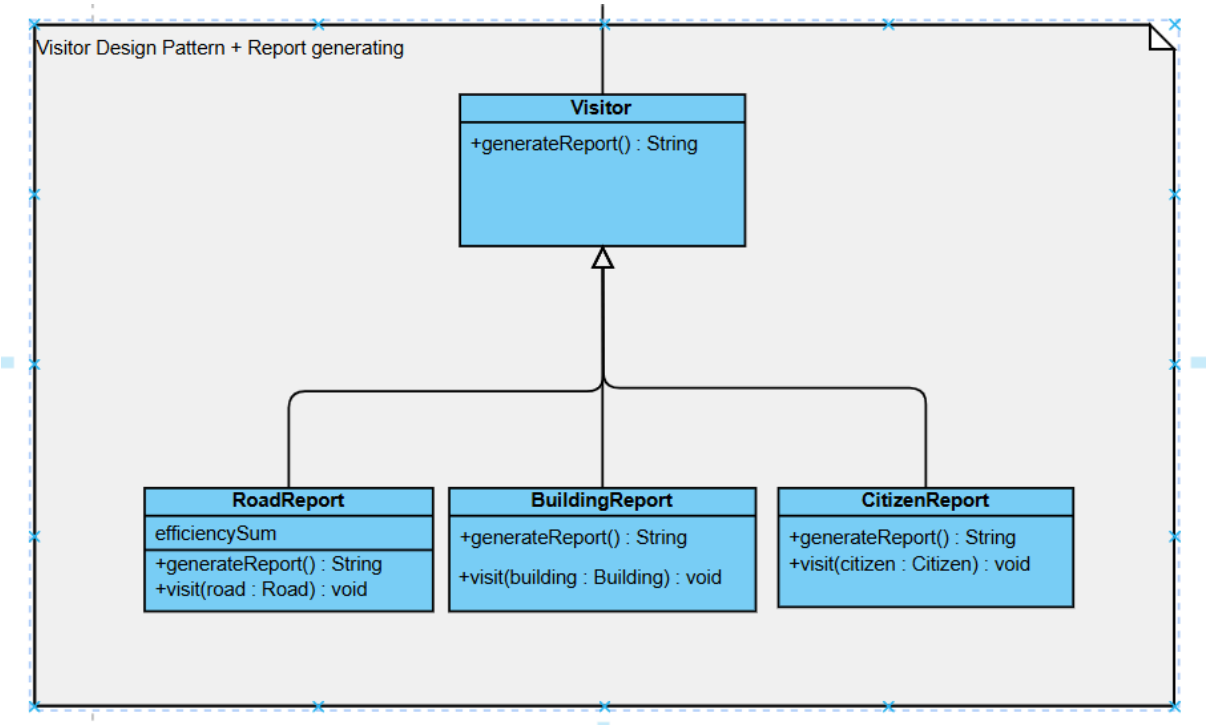
The pattern is implemented with:

1. **Interface**
    - CrimePunishmentStrategy base class
    - Pure virtual punishCrime method
2. **Concrete Strategies**
    - CommunityServiceStrategy
    - PrisonStrategy
    - DeathSentenceStrategy
3. **Integration with Citizen**
    - Citizen holds CrimePunishmentStrategy pointer
    - Can dynamically change punishment strategies
    - Manages crimes vector and criminal record

Why It's a Good Fit

1. **Dynamic Behavior**
    - Punishment strategies can change based on:
        - Crime severity

        - Criminal history
        - City policies
2. **Clean Implementation**
    - Each strategy encapsulates specific punishment logic
    - Easy to add new punishment types
    - Citizen class remains unchanged when adding strategies



Visitor Pattern Analysis

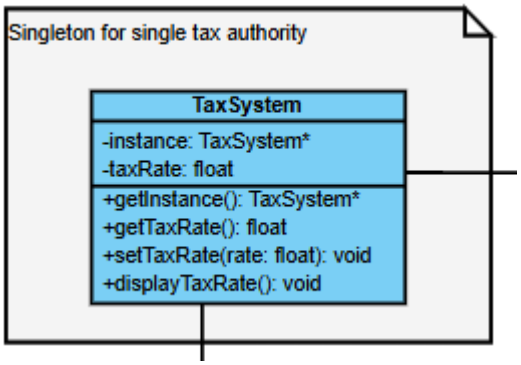
Implementation Context

The Visitor pattern is used for report generation across different city elements:

- Roads
- Buildings
- Citizens

Why It's a Good Fit

1. **Separation of Reporting Logic**
    - Report generation is separated from object classes
    - Each report type has its own visitor
    - Objects don't need to know about report generation
2. **Easy to Add New Reports**
    - New report types can be added without modifying existing classes
    - Each report can collect different statistics
    - Can implement specialised reporting behaviour
3. **Type-Safe Processing**
    - Each visitor method is typed to specific element
    - Compile-time type checking
    - Clear interface for adding new visitable elements



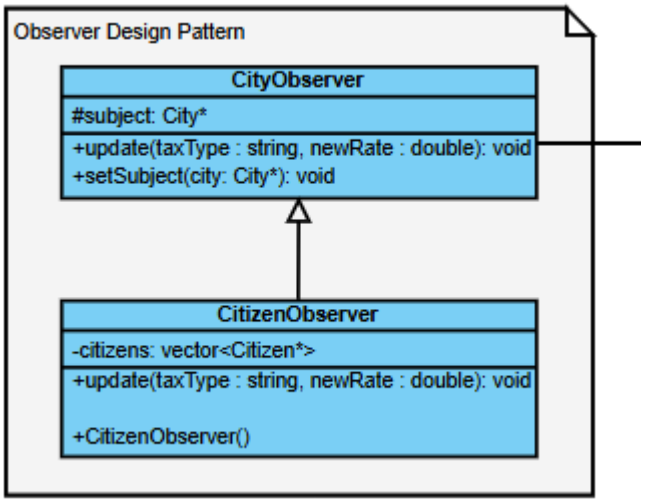
Singleton Pattern Analysis: Tax System

Implementation Context

The Singleton pattern is used for the TaxSystem, ensuring a single tax authority for the entire city simulation.

Why It's a Good Fit

1. **Single Source of Truth**
    - One tax authority for the entire city
    - Consistent tax rates across all operations
    - Centralised tax policy management
2. **Global Access**
    - Easy access from anywhere in the system
    - No need to pass tax system references
    - Consistent state across the application
3. **Resource Efficiency**
    - Only one instance created
    - Saves memory
    - Prevents conflicting tax policies



Observer Pattern Analysis

Implementation Context

The pattern is used to monitor and react to tax rate changes in the city, with:

- CityObserver as abstract observer
- CitizenObserver as concrete implementation
- City as the subject being observed

Why It's a Good Fit

1. **Real-time Updates**
    - Citizens immediately react to tax changes
    - Automatic propagation of policy changes
    - Maintains system consistency

Benefits

1. **Automated Updates**
    - No polling required
    - Real-time response to changes
    - Consistent system state
2. **Modularity**
    - Observers can be added/removed dynamically
    - Easy to modify update behaviour
    - Clean separation of concerns

References

Urban social functional requirements for a 20-minute neighbourhood in Melbourne: https://www.researchgate.net/figure/Urban-social-functional-requirements-for-a-20-minute-neighbourhood-in-Melbourne_fig2_361407207

How to build a city, step-by-step: A DIY guide: https://www.theguardian.com/cities/2015/jun/30/how-build-city-step-by-step-diy-guide

Taxes: https://www.investopedia.com/terms/t/taxes.asp