Team-24 Design Pattern Application Report

We made use of 11 design patterns:

Singleton: The singleton was used in the ResourceManager, CitizenManager, CityManager, ZoneManager to ensure a single instance of the core management systems was used

Command: We encapsulate city operations as objects to make issuing said operations simple and efficient. It was used in the BuildCommand, DemolishCommand, ChangePolicyCommand, UpdateResourcesCommand, SimulationCommands.

State: It was used to manage the life cycle of buildings used in the Abandoned, Operational, UnderConstruction and UnderMaintenance.

Observer: Notification and alerts of changes in the system that affected the civilization. Used in the Building, Zone, Government, Citizens and Statistics classes.

Visitor: Separated the algorithms and function calls from object structures. Used in the BuildingVisitor, InspectionVisitor, MaintenanceVisitor , Building, Commercial, Industrial, Residential , Landmark classes

Strategy: Defined different methods and policies for running the city. Used in the TaxStrategy, HighIncomeTaxStrategy, LowIncomeTaxSrategy and TaxPolicy classes

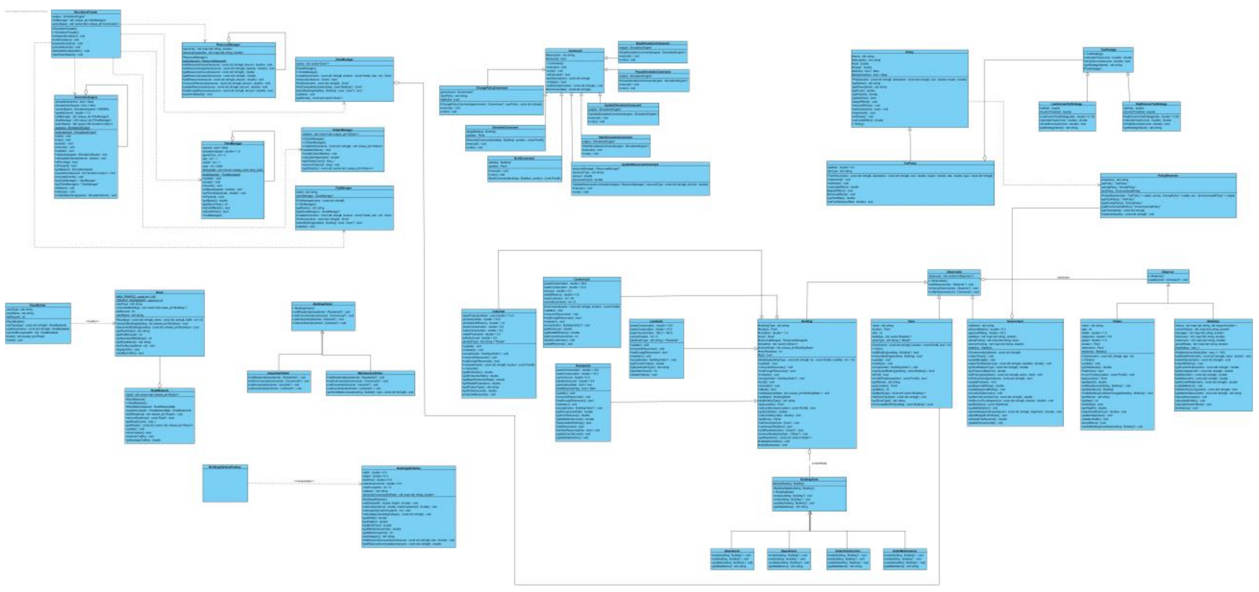
Factory Method: Used to create building attributes. Used in the BuildingAttributeFactory and BuildingAttribute classes.

Memento: Used to store specific policies and previously successful strategies. Used in the Policy, PolicyMemento and Government classes.

Façade: Provided an easy to use, unified interface. Used in the SimulationFacade, SimulationEngine classes

Composite: Handled zones and treated them uniformly. Used in the ZoneManager and zone classes.

Builder: Used to construct complex road networks efficiently. Used in the RoadBuilder, Road and RoadNetwork classes.



The diagram illustrates how each design pattern organizes classes in the city management system to create a cohesive and modular architecture:

1. Singleton Pattern: `CityManager`, `ResourceManager`, `CitizenManager`, and `ZoneManager` are shown as single-instance classes, each managing core functions like resource allocation and citizen data to avoid duplication.

2. Command Pattern: The `Command` interface links to concrete classes like `BuildCommand`, `DemolishCommand`, and `ChangePolicyCommand`, encapsulating city operations as executable actions in a queue for efficient processing.

3. State Pattern: The `Building` class connects to states such as `Abandoned`, `Operational`, `UnderConstruction`, and `UnderMaintenance`, with arrows showing transitions to manage the lifecycle of each building dynamically.

4. Observer Pattern: The `Government`, `Citizens`, `Building`, and `Statistics` classes are observers subscribed to system updates, illustrated by notifications that keep each component synchronized with real-time changes.

5. Visitor Pattern: `BuildingVisitor`, `InspectionVisitor`, and `MaintenanceVisitor` classes are connected to building types like `Commercial`, `Residential`, and `Industrial`, each visitor

representing separate algorithms applied across these structures without modifying their definitions.

6. Strategy Pattern: The `TaxPolicy` class references `TaxStrategy`, with specific strategies like `HighIncomeTaxStrategy` and `LowIncomeTaxStrategy,` allowing adaptable tax methods for city revenue management.

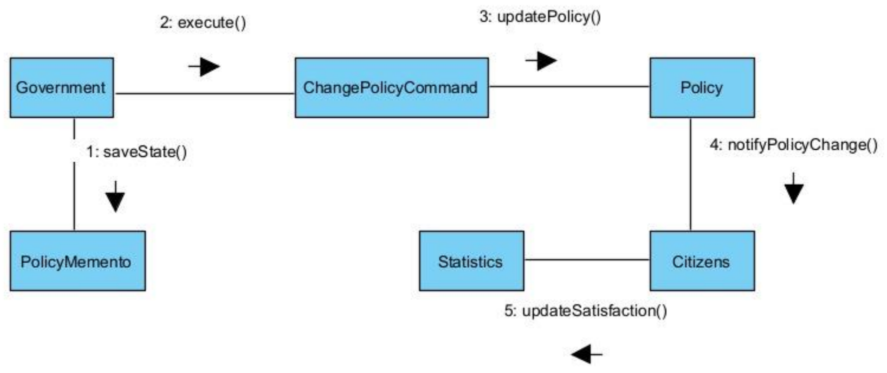
7. Factory Method Pattern: `BuildingAttributeFactory` produces `BuildingAttribute` instances, standardizing building attribute creation across different types, as seen by class arrows linking attributes to building types.

8. Memento Pattern: `PolicyMemento` stores snapshots of the `Policy` class, linked to `Government`, enabling it to revert to prior successful policies.

9. Façade Pattern: `SimulationFacade` interacts with `SimulationEngine`, simplifying complex simulation interactions by providing a streamlined interface for core functionality access.

10. Composite Pattern: `ZoneManager` manages composite `Zone` objects, represented by nested zones or individual buildings, allowing unified zone management.

11. Builder Pattern: `RoadBuilder` constructs `RoadNetwork` by assembling `Road` objects, simplifying the process of creating complex road systems in the city.



As shown above the government acts as a caretaker and a concreteSubject. It can both save policies while changing policies when needed. It demonstrates use of the Command pattern in the ChangePolicyCommand class. The Memento class and Command classes are shown to work in tandem as Citizens and Satisfaction are notified upon a successful change in policy from the Memento patterns store. This is just an example of the way patterns function and interlink within the system.

The implementation of these patterns was delegated to various members in the group. Group members were assigned related patterns. Each of these groups of patterns were implemented in separate branches in the Github repository. When each section was completed they were all merged into one main branch. Incremental commits were made to show careful documentation of the small improvements and tests that were ran during the coding process.