REPORT

Research Brief:

Urban development and effective city management are essential for creating sustainable, functional, and vibrant communities. A well-managed city integrates various components that work together to ensure the welfare of its citizens and the longevity of its infrastructure. The following key elements play crucial roles in urban development:

Buildings
Buildings serve as the primary habitat for residents and the foundation for businesses. They must be designed with consideration for functionality and sustainability. Proper zoning and planning help manage building density and land use, ensuring that residential, commercial, and industrial needs are balanced.

Utilities
Utilities, including electricity, water, waste management, and sewage systems, are vital for maintaining the quality of life. Effective management of these utilities ensures that citizens have access to essential services, which directly impacts public health and satisfaction.

Transportation
A well-developed transportation system is critical for connecting citizens with jobs, services, and leisure activities. This includes private and public transit, and pedestrian pathways. Utilising strategies and state patterns in transport management allows for flexibility and adaptability in transport modes, making the system responsive to user needs and conditions.

Citizens
Citizens are the heart of any city. Their needs, preferences, and engagement in local governance significantly influence urban planning and management. Active participation through feedback mechanisms ensures that developments align with community values and expectations, fostering a sense of ownership and responsibility.

Government
Local government plays a pivotal role in urban development by establishing policies, regulations, and planning frameworks that guide growth. Effective governance involves collaboration with citizens to address challenges such as housing, infrastructure, and environmental sustainability.

Resources
Efficient management of resources, including energy, water, and materials, is critical for sustainable urban development. Implementing resource management strategies.

Taxes
Taxes are a primary revenue source for local governments, funding essential services and infrastructure projects. Transparent tax policies and equitable distribution of tax burdens contribute to a fair and sustainable urban environment, enabling cities to invest in city growth.

City Growth
City growth must be managed strategically to balance economic development with environmental considerations. Planning for growth involves anticipating changes in population, ensuring that infrastructure and services can support expansion without compromising citizen satisfaction.

References

- Bhatta, S. (2010). *Urban Development and Urban Planning: Towards a Global Paradigm*. Springer.
- Hall, P. (2014). *Cities of Tomorrow: An Intellectual History of Urban Planning and Design in the Twentieth Century*. Wiley-Blackwell.
- Glaeser, E. L. (2011). *The Triumph of the City*. Penguin Press.

Design Pattern Application Report

Transportation System:

The transportation system uses the Strategy and State design patterns to create flexible and adaptable transport management.

1. Strategy Pattern:
    - Purpose: Allows Citizen to choose different transport modes and switch between them easily.
    - Key components:
        - Context: Citizen class
        - Strategy Interface: TransportStrategy defines the common functions for transport modes
        - Concrete Strategies: ModeOfTransport with derived classes PublicTransport and PrivateTransport with specific types like Car, Bus, and Bike.
    - How it works:
        - Citizen can select and change transport modes at runtime, depending on user needs or conditions.
    - Benefit:
        - Makes the system flexible and easy to extend with new transport types
2. State Pattern:
    - Purpose: Allows ModeOfTransport to change behaviour based on its state (e.g., Busy, Quiet)
    - Key Components:
        - Context: ModeOfTransport class
        - State Interface: TransportationState defines behaviour changes
        - Concrete States: BusyState, QuietState, ModerateState under PublicTransportState or PrivateTransportState
    - How it works:
        - Each transport mode adjusts its behaviour (availability) according to its current state.
    - Benefit:
        - Ensures transport modes respond correctly to different conditions
3. Summary
    - Strategy Pattern: Simplifies adding and switching transport mode for Citizen
    - State Pattern: Enables transport modes to adapt behaviour based on current conditions

    Both patterns make the transportation system more flexible, maintainable, and easy to expand.

Command Pattern

The Command pattern is used in the government component to enable centralized management of various government actions such as adjusting tax rates, setting policies, allocating budgets, and updating public services. This design pattern decouples the request for an action from the implementation, allowing actions to be queued, executed, or undone independently.

Purpose:

Allows centralized and flexible control of government actions by encapsulating each operation as a command.

Key Components:

- **Invoker**: Government class – holds and executes commands.
- **Command Interface**: Command – defines a standard interface for executing commands.
- **Concrete Commands**:
    - TaxCommand – for adjusting tax rates.
    - AllocateBudgetCommand – for setting the city budget.
    - PolicyCommand – for implementing new policies.
    - PublicServicesCommand – for updating available public services.
- **Receiver**: City and its departments (e.g., TaxationDepartment, BudgetDepartment, PoliciesDepartment, PublicServicesDepartment).

How It Works:

1. **Command Creation**: Specific government actions are encapsulated in command objects (e.g., TaxCommand, PolicyCommand).
2. **Execution**: The Government class invokes each command, which triggers specific actions in the respective departments.
3. **Flexibility**: Commands can be stored, queued, or reused, allowing the government to adjust city operations without directly altering department logic.

Benefits:

- **Decoupling**: Separates government actions from department implementations, allowing flexibility in how and when actions are executed.
- **Maintainability**: Each command encapsulates a single responsibility, making the system easier to modify and extend.
- **Flexibility**: Commands can be modified, combined, or added without altering the government component's structure.

Summary:

- **Command Pattern**: Centralizes control over government actions, making operations like tax adjustments, budget allocations, and policy changes easy to execute and manage.

- **Benefit**: The pattern enhances flexibility and maintainability, allowing for scalable and extendable government functionality.

By using the Command pattern, the government component becomes a centralized control hub, allowing for organized and manageable city operations, while each department's responsibilities remain encapsulated. This makes the overall system robust, scalable, and adaptable to future needs.

Adapter Design Pattern

The Adapter Design Pattern is applied to facilitate integration between a class (**CompositeBuilding**) that has an incompatible interface with the system's expected operations defined by the **BuildingTarget** interface. By using this pattern, existing functionality from CompositeBuilding is reused effectively without altering its internal structure. This results in a more cohesive system where different components can interact seamlessly.

Addressing Specific Functionalities

1. **Interface Compatibility**: The system expects a consistent interface (**BuildingTarget**) for managing different types of units (residential, commercial, industrial). The Adapter pattern addresses this by wrapping the **CompositeBuilding** class and providing the required methods. This makes it possible to use **CompositeBuilding** as a **BuildingTarget**, enabling interaction through a unified set of methods.
2. **Simplification of Client Code**: The adapter hides the complexity of **CompositeBuilding**'s implementation from the client. Instead of clients dealing with **CompositeBuilding**'s detailed logic, they interact with a simplified and uniform interface. This makes the system easier to understand and reduces the potential for errors.
3. **Extensibility and Flexibility**: By implementing the Adapter pattern, the system gains flexibility to accommodate new functionality without significant refactoring. For example, if new methods or unit types need to be added, the **Adapter** class can be extended to incorporate these changes, ensuring minimal impact on existing code.
4. **Encapsulation of Complex Operations**: The adapter encapsulates complex operations like population and job management in **CompositeBuilding** and exposes them through simplified methods. This abstraction not only simplifies the integration but also ensures that the internal workings of **CompositeBuilding** are protected and modularized.

UML Diagram Explanations

The UML diagrams illustrate the following:

1. **Class Diagram**: Shows the relationships between BuildingTarget, Adapter, and CompositeBuilding. The Adapter class implements the BuildingTarget interface and uses an instance of CompositeBuilding to delegate method calls. This highlights the core structure of the Adapter pattern.
2. **Sequence Diagram**: Illustrates the flow of method calls from the client to the Adapter and then to CompositeBuilding. This diagram emphasises how the Adapter translates and forwards requests, demonstrating the seamless interaction between previously incompatible interfaces.

3. **Object Diagram**: This diagram depicts an example scenario where the Adapter is instantiated with a CompositeBuilding object, showcasing the state of these objects and their interaction at runtime.

Key Components:

- **Target**: BuildingTarget – standard interface for city buildings.
- **Adapter**: Adapter – translates BuildingTarget methods for CompositeBuilding.
- **Adaptee**: CompositeBuilding – existing class adapted to meet the new interface

Observer design pattern

The **Observer Design Pattern** is implemented in the system to enable communication between the government and its citizens. This pattern establishes a publish-subscribe mechanism, where **Citizen** objects (observers) listen to updates from the **Government** (subject) and react accordingly. This design efficiently handles the need for citizens to be aware of changes in policies or tax rates, providing a dynamic and responsive interaction model.

Addressing Specific Functionalities

1. **Dynamic Updates**: The **Government** class serves as the subject that keeps track of all registered citizens. By implementing the observer pattern, the government can notify citizens immediately when changes occur, such as updates in policies or adjustments in tax rates. This dynamic communication ensures that citizens are always informed about the current state of governance.
2. **Event Handling and Notifications**: The **PublicServicesDepartment** class, which is a concrete subject derived from **Government**, extends this notification system. It has the capability to manage city functions and public services, broadcasting relevant updates to citizens. This structure allows for efficient event handling, as changes in public services or policies are communicated to all affected parties in real-time.
3. **Citizen Response to Changes**: The **Citizen** class, which represents the observers, reacts to government notifications through the **update()** method. This allows each citizen to adjust their state based on the new information, such as updating satisfaction levels or responding to policy changes. For example, if the tax rate increases, citizens adjust their satisfaction to reflect the impact of the change. The observer pattern makes this process seamless, ensuring each observer reacts promptly and consistently.
4. **Loose Coupling and Scalability**: By decoupling the **Citizen** observers from the **Government** subject, the system achieves loose coupling. This design makes it easier to add new citizen types or modify existing government behaviour without significant refactoring. It also allows for easy extension of the notification system, enhancing the scalability of the application.

UML Diagram Explanations

1. **Class Diagram**: The class diagram shows the relationships between **Government** (subject), **PublicServicesDepartment** (concrete subject), and **Citizen** (observer). It highlights how the **Government** maintains a list of **Citizen** observers and provides methods to add, remove, and notify them. The diagram also illustrates how the **PublicServicesDepartment** inherits from the **Government** and manages city-specific functionalities.
2. **Sequence Diagram**: A sequence diagram illustrates the flow of a notification event. It shows the process from a change in government policy triggering a call to **notifyObservers()**, which then invokes the **update()** method on each registered **Citizen**. This visualises the communication and response mechanism in action.
3. **State Diagram**: The state diagram represents how a Citizen's state changes in response to notifications from the government. For instance, when a tax rate change

is announced, the citizen's satisfaction level transitions accordingly. This diagram helps illustrate the dynamic nature of the observer pattern in managing citizen states.

Key Components:

- **Subject**: Government – maintains and notifies observers.
- **Concrete Subject**: PublicServicesDepartment – manages services and updates citizens.
- **Observer**: Citizen – responds to government changes and adjusts accordingly.

Utilities System:

The Utilities System employs the Decorator design pattern to enhance the functionality of ConcreteComponents (Residential, Commercial, Industrial, Landmarks) by providing essential services that improve citizen satisfaction and building performance.

Decorator Pattern:

Purpose: The Utilities System uses decorators to dynamically add functionality to ConcreteComponents without altering their structure. Each utility decorator enhances specific aspects of a building's operation and the overall city environment.

Key Components:

- **Component Interface:** Utilities serves as the base interface for all utility decorators.
- **Concrete Components:** Residential, Commercial, Industrial, and Landmark classes represent the buildings in the city.
- **Decorators:** Various utility decorators (e.g., PowerPlant, WaterSupply, WasteManagement, SewageSystem) that provide specific services to the ConcreteComponents.

How it Works: Each utility decorator interacts with ConcreteComponents, modifying their state and behaviors based on the services provided. This interaction leads to changes in crucial variables, such as population growth, employment ratings, and environmental impacts. For instance, a PowerPlant decorator may provide electricity, impacting the happiness of residents and functionality of businesses, while a WaterSupply decorator ensures access to water, affecting health and operational capabilities.

Benefit: The system allows for dynamic enhancements to building functionality without modifying the underlying component classes, promoting flexibility, maintainability, and easier expansion of services.

Summary

- **Utilities System** enhances building functionality by managing critical resources through the use of decorators.
- Decorators impact key variables across different building types, improving overall city management and citizen satisfaction.

This revised structure maintains focus on the general role of decorators in the Utilities System while providing essential details on how they operate within the simulation.

Iterator Pattern:

Purpose: The Iterator Pattern provides a consistent way for citizens to traverse through various types of buildings in the city. It hides the internal structure of the building collection, allowing citizens to navigate through elements without needing to understand how they are stored or organized.

Key Components:

- **Iterator Interface:** Defines the standard operations for iterating over a collection. In this case, BuildingIterator is used to provide the traversal logic for the Building collection.
- **Concrete Iterator:** BuildingIterator is a concrete implementation that iterates through a collection of Building objects. It includes methods like hasNext() to check for remaining elements and next() to retrieve the next building.
- **Collection:** The collection of Building objects can include various types of buildings (e.g., Residential, Commercial). BuildingIterator is initialized with the beginning and end of this collection.

How it Works: The BuildingIterator is created by specifying the start and end of the building collection. Citizens can use this iterator to navigate through each building in a standardized way. They call hasNext() to check if there are more buildings to visit and next() to move to the next building in the collection. This structure enables a smooth traversal experience across different types of buildings without exposing the underlying details of the collection, which could vary in structure (array, list, etc.).

Benefit: The Iterator Pattern simplifies building navigation for citizens by standardizing traversal. It abstracts away the complexity of the collection structure, making it easier to add new building types or modify the collection without impacting client code. This improves the flexibility and maintainability of the system, as well as the user experience for citizens exploring the city.

Summary: The Iterator Pattern provides a uniform traversal method, enabling citizens to explore different types of buildings without needing to understand the collection’s internal structure. By separating the traversal logic from the collection itself, the system maintains flexibility and adaptability for future enhancements.

Resources System:

The Resources System establishes associations between Utilities and shared Resource objects using the Flyweight design pattern, ensuring efficient resource management in the City Builder Simulation.

Utilities --> Resource (Flyweight) Association

Purpose: Concrete utility decorators, such as PowerPlants and WaterSupply, need access to shared resources like Energy or Water for their operation.

How It Works:

- **Interaction:**
  - When a building requests power, the PowerPlants decorator interacts with the Energy flyweight to assess available energy.
  - When water is needed, the WaterSupply decorator checks the Water flyweight for capacity and supply verification.
- **Example for PowerPlants:**
  - The PowerPlants decorator maintains variables like `powerGenerated` and `powerConsumed`.
  - It ensures that power consumption does not exceed the total available power.
  - The interaction with the Energy flyweight involves:
    - **Check Availability:** Calls methods to verify if sufficient power can be allocated.
    - **Consume Energy:** Adjusts the energy available for the building by decrementing the value in the Energy flyweight.
- **Efficiency:** Utilizing the Energy flyweight means that instead of creating separate instances for each building, the PowerPlants decorator references a single shared Energy object, updating availability centrally.

Budget Resource:

- The construction of buildings and utilities consumes budget resources, ensuring that financial management is integrated into resource allocation. The Budget resource monitors the costs associated with creating and maintaining these structures, preventing overspending.

UML Representation:

- The association is represented as **Utilities (abstract class) --> Resource (Flyweight)**, indicating access to shared resources.
- Concrete utility classes (e.g., PowerPlants, WaterSupply) associate with their respective resource types, leveraging the flyweight pattern for management efficiency.

Utilities --> ResourceFactory (FlyweightFactory) Association

Purpose: The ResourceFactory provides instances of shared resources to utility decorators, streamlining resource management.

How It Works:

- **Interaction:**
  - When a utility requires a resource, it queries the ResourceFactory instead of creating the resource directly.
  - The factory checks if the resource already exists:
    - If it exists (e.g., Energy flyweight), the factory returns the existing instance.
    - If it doesn't exist (e.g., a new material type), the factory creates a new flyweight instance and stores it.
- **Example for PowerPlants:**
  - The PowerPlants utility decorator requests the current Energy flyweight from the ResourceFactory.
  - The factory either returns the existing flyweight or creates a new one if necessary.
- **Efficiency:** This approach minimizes memory usage by ensuring only one instance of each resource type exists, shared among all relevant components. The factory centralizes resource creation control to prevent duplication.

UML Representation:

- The association is shown as **Utilities (abstract class) --> ResourceFactory**, indicating that every utility interacts with the factory to obtain the appropriate shared resources (flyweights).
- Concrete utility classes obtain resources from the ResourceFactory without needing to manage their creation.

Summary

- **Utilities --> Resource (Flyweight) Association:**
  - **Role:** Enables utility decorators to interact with shared resource objects (e.g., Energy, Water).
  - **Interaction:** Decorators check availability and consume resources through shared flyweights.
  - **Efficiency:** Guarantees a single instance of a resource is used across multiple buildings.
  - **Budget Resource:** Building and utility construction costs are deducted from the Budget resource, integrating financial management into resource allocation.
- **Utilities --> ResourceFactory (FlyweightFactory) Association:**
  - **Role:** Utilities request resources from the factory rather than creating them directly.
  - **Interaction:** The factory manages the lifecycle of flyweights, returning existing or new resource objects as needed.

- **Efficiency:** Centralizes resource management, ensuring only necessary flyweights are created and reused.

These associations contribute to efficient resource management, enabling dynamic city growth while minimizing redundant resource allocation, in line with the Flyweight pattern's objectives to reduce memory usage.

Government System

The Government system uses the Template Method and Chain of Responsibility for management of the different government departments to oversee the city.

1. Template Method:

    - Purposes: The government runs the city through different departments that manage the taxes, policies, budget and public services and notifies citizens of the changes.

    - Key Components:

        § Abstract Class: Government class implements a template method operation that generates a government report.

        § Concrete Class: TaxationDepartment, BudgetDepartment, PoliciesDepartment and PublicServicesDepartement classes that all implement their primitive operation for changes made by the government.

    - How it works: The Government can set and update the tax rate, city budget, policies and public services and will notify citizens of changes made and generate a report that has details of these values and changes made

    - Benefits: Defines a consistent structure for generating the report across all department classes

2. Chain of Responsibility:

    - Purposes: Allows citizens to request services from the Public Service Department which finds the service to handle the request and provide the service.

    - Key Component:

        § Handler: PublicServiceDepartment classes

        § Concrete Handler: HealthCare, LawEnforcement, Education classes

    - How it works: Upon request the public service department processes the request along the chain of different service departments until it reaches the department that successfully handles the request and provides the service to the citizen.

    - Benefits: Flexibility of assigning and adding concrete handlers, for example if services are added or removed.

3. Summary:

    - Template Method: Government can easily manage city taxes, budget, policies and services and generate a report of them and notify citizens of changes.

    - Chain of Responsibility: Public Services can easily process requests to provide services to citizens.

Building System:

The Composite pattern was applied to manage the hierarchical structure of buildings in the city simulation. This design pattern enables us to treat individual objects and compositions of objects uniformly. In this context, the CompositeBuilding acts as a container of various Building components, such as Residential, Commercial, Industrial, and Landmark types. Each building type can, in turn, contain sub-elements, like specific residential units or commercial areas, making it straightforward to organise and manipulate the hierarchy of structures.

Application in the System

In the City Simulation System, the Composite pattern addresses the need for a flexible building structure, allowing individual buildings and building groups to be managed seamlessly. This pattern enables functionalities like adding, removing, or retrieving buildings from the city's structure.

- Component Class: Building
  This abstract base class defines common operations for both individual and composite buildings, including methods for adding, removing, and displaying building information.
- Composite Class: CompositeBuilding
  This class contains a collection of Building objects, representing a group of buildings as a single unit. CompositeBuilding enables recursive behaviour, allowing each building to hold other buildings in a tree-like structure.
- Leaf Classes: Residential, Commercial, Industrial, Landmark
  These are individual building types that do not contain further components. However, they can be added to the CompositeBuilding, thereby forming the nodes of the tree.

Benefits

- Simplifies Building Management: Using the Composite pattern, the system treats both individual buildings and groups of buildings uniformly, simplifying the implementation of complex city layouts.
- Scalability: New building types can easily be added as subclasses of Building without modifying existing code, supporting system scalability.

City Growth System

The Builder pattern was chosen to manage the step-by-step construction of complex city structures, specifically the initial setup and ongoing development of city elements. In this simulation, various city attributes like population, jobs, infrastructure, and economic sectors require specific and sequential construction steps. The Builder pattern allows creating a City object in distinct stages, controlled by the Player class.

Application in the System

The Builder pattern addresses the need for flexible, controlled construction of city attributes and structures. It was especially useful when creating a complex city setup with multiple configurable parameters, where each attribute needs distinct construction logic.

- Builder Interface: Builder
  This abstract class defines the method signatures for building different city aspects, such as buildPopulationGrowth, buildHousingExpansion, buildEconomicDevelopment, and buildInfrastructureExpansion.
- Concrete Builder: CityBuilder
  This class implements the Builder interface, defining the logic for creating and adding each component of the city. For example, buildPopulationGrowth increases the city's population, while buildEconomicDevelopment adds job opportunities within the city. Each of these steps builds an aspect of the CompositeBuilding.
- Director Class: Player
  This class acts as the Director, coordinating the construction steps in a specified order to ensure a consistent and fully-formed City. The player uses the builder to construct city elements based on user choices.
- Product: CompositeBuilding
  Once built, CompositeBuilding serves as the final product, representing the complete city structure, including its population, economic sectors, infrastructure, and other elements.

Benefits

- Encapsulates Construction Logic: Each stage of the city's development is encapsulated within distinct methods, making the construction process more modular and readable.
- Customizable City Configuration: By adjusting the builder sequence in the Player class, different city configurations can be created without modifying the underlying code, enhancing flexibility and scalability.

Branching strategy

To ensure smooth collaboration and effective version control, our branching strategy includes a structured approach with the following branches: a main branch, a testing branch, and individual branches for each team member. This structure allows us to manage code integration effectively and minimise conflicts.

1. Main Branch
    - The main branch is the central, stable branch containing the final version of the project code. Only thoroughly tested and reviewed code is merged here. The main branch represents the official project state and is intended for release or submission.
    - Direct commits to the main branch are discouraged. Instead, changes should be merged from the testing branch after integration testing has been completed.
2. Testing Branch
    - The testing branch serves as an intermediary branch for code integration. Team members merge their individual branches into the testing branch, where we conduct integration tests to identify any compatibility issues between different components.
    - After successful testing and validation on this branch, the integrated code is ready to be merged into the main branch.
    - The testing branch allows us to catch and resolve conflicts, bugs, or unexpected behaviour early, ensuring that only stable code reaches the main branch.
3. Individual Branches
    - Each team member has their own branch to work on specific features or components of the project independently. This separation helps to isolate work, preventing conflicts and keeping each developer's work organised.
    - In individual branches, team members can develop and test their code without affecting the work of others. Once a feature is stable and ready for integration, it is merged into the testing branch.
    - Team members are encouraged to frequently pull updates from the testing branch to ensure their components remain compatible with the current state of the project.

Workflow :

- Team members develop on individual branches, regularly testing and committing changes.
- Once a feature is complete and tested individually, it is merged into the testing branch for integration testing.
- After successful integration testing, the tested code from the testing branch is merged into the main branch, ensuring a stable, cohesive final product.

This branching strategy enhances modular development, ensures integration accuracy, and maintains the stability of the main project code.