***Smart Grid Discrete Event Simulation System***

## (Java-Based OOP Project Report)

# Revised Report

# **Name: Zaid Muhammad**

## **Matricola Number: 560175**

## **Course: Data Analysis**

## **Professor:** **Dr. Salvatore Distefano**

## **University Name: University of Messina**

## **Submission Date: 29 May 2025**



# Abstract

This project presents a **simulation of a smart grid system** involving various types of **entities** such as houses, factories, electric vehicles, and renewable energy sources.

Each entity manages its own energy production, consumption, and interaction with the grid based on changing weather conditions and battery levels. The simulation employs **discrete event scheduling** and **object-oriented programming (OOP)** concepts, including the observer pattern.

The report details how key OOP principles: **encapsulation**, **inheritance**, and **polymorphism** were applied in the system's design. It also explains how the architecture was built to be **modular**, **extensible**, and **maintainable**. Overall, the simulation provides insights into smart grid behavior and demonstrates how OOP can effectively model complex, dynamic systems.

# **1. Project Overview**

The goal of this project is to simulate a **smart grid** where different entities like houses, factories, and renewable sources manage their **own energy production, consumption, and trading.** The system also handles **faults and repairs** to keep things realistic.

Some key features include:

* **Energy production** depends on **weather conditions**, so things change dynamically.
* Entities **buy and sell energy** with the grid based on their battery levels.
* The program checks for **battery overflow and underflow** to avoid errors.
* It uses a **discrete event simulation** to handle all actions in order over time.
* The **Observer pattern** is used to update entities about weather changes.
* The design follows **OOP principles** to keep the code organized and easy to extend.

# **2. Discrete Event Simulation Explanation**

The simulation is based on a **Discrete Event Simulation (DES)** model. The core idea is to **manage and process events** in chronological order, simulating how the smart grid behaves over time.

* The **EventCalendar** class uses a **PriorityQueue<Event>** to keep track of all scheduled events. This queue automatically orders events by their **scheduled time** and **priority**, ensuring they are processed in the correct sequence.
* The **TimeManager** class keeps track of the current simulation time, advancing it as events are processed.
* The simulation includes several types of **events,** each representing different actions in the system:

1. **ProductionEvent:** entities produce energy.
2. **ConsumptionEvent:** entities consume energy.
3. **TradeEvent**: energy trading between entities and the grid.
4. **WeatherEvent:** updates weather conditions affecting energy production.
5. **FaultEvent:** simulates faults like transformer failures.
6. **RepairEvent**: simulates repairs to restore functionality.

* Each event class implements a **process()** method that contains the logic executed when the event occurs.
* During the simulation, events are fetched one by one from the **EventCalendar**, the simulation time is advanced accordingly, and the **event’s process()** method is called. This **cycle continues** until all events are processed.

# **3. Decentralized Design**

In this **project**, each entity handles its own stuff. That means things like managing the battery, deciding how much energy to **produce or consume**, and when to **trade energy** with the grid are all done inside the entity itself.

The **SmartGridManager** just provides shared things like the **current weather** or how much energy and money the grid has. It doesn’t tell the entities what to do or control their behavior.

Because of this, **each entity acts on its own** based on its situation and the info it gets, like **weather updates**. The simulation’s event system schedules these actions **independently**, so entities can do their thing without waiting on a **central controller.**

This way of doing things makes the system easier to **manage and expand.** If I want to add new types of entities later on.

# 4. Object-Oriented Principles

The **OOPs principles** are given below:

4.1 Encapsulation

Defination:

**Encapsulation** is the **object-oriented principle** of bundling data **(attributes**) and the **methods** that operate on that data into a **single unit**, typically **a class**. It restricts direct access to an **object's internal state** by declaring **attributes** as private, and exposes controlled access through **public or protected methods** (getters and setters). This mechanism ensures that the internal representation of an object can only be changed in a **controlled** and **safe** manner.

**How I Applied It:**

In my project, encapsulation is implemented in every core class, especially the abstract Entity class and its subclasses. All instance variables are declared as private, and access to these variables is provided through **public getters** and **protected setters**. Additionally, business logic methods (such as storeEnergy and useEnergy) encapsulate operations on the data, ensuring all changes are validated and safe.

A computer screen shot of text

AI-generated content may be incorrect.

Explanation

* **Getters provide controlled, read-only access** to private attributes, preventing direct external modification.
* **Setters are protected and include validation**, so only trusted code (the class or subclasses) can modify state, and only with valid values.
* **Business methods** (like storeEnergy and useEnergy) encapsulate logic that operates on the internal state, ensuring all mutations respect class invariants (e.g., battery capacity limits).
* **Exception handling** (with BatteryException) is part of encapsulation, as it ensures that invalid operations are caught and handled within the class.

Motivation

**Why getters/setters?**  
To provide controlled interfaces to internal data, enabling validation and preventing inconsistent states.

**Why protected setters?**  
To restrict modification rights to trusted code, avoiding arbitrary external changes.

**Why encapsulated logic?**  
To centralize validation and state management, ensuring object integrity and robustness.

**Why use exceptions?**  
To enforce correct usage and error handling internally, preventing invalid state changes from propagating.

Benefits Of Encapsulation

* **Improves Maintainability:**  
  Changes inside a class do not affect external code because internal details are hidden.
* **Enhances Robustness:**  
  Prevents invalid or inconsistent states by controlling how data is accessed and modified.
* **Increases Security:**  
  Protects data from unauthorized or accidental access and modification.
* **Supports Clear Interfaces:**  
  Simplifies understanding and debugging by exposing only necessary methods.
* **Promotes Modularity and Reusability:**  
  Encapsulated classes can be reused and extended without breaking other parts of the system.

4.2: Information Hiding:

Defination:

**Information hiding** is the principle of **restricting access** to the **internal implementation details** of a class, exposing only what is necessary through well-defined interfaces. It focuses on protecting **sensitive data** and **internal workings** from **unauthorized** or **unintended** access, thereby reducing system complexity and coupling.

**How I Applied Information Hiding (3-Step Engineering)**

**Step 1: Always Private**: All instance variables are declared **private** to prevent direct external access, isolating the internal state.

**Step 2: Class or Instance**: Class-level constants (shared by all entities) are declared private static final, e.g.:

|  |  |  |  |
| --- | --- | --- | --- |
| Attribute | Access | Class/Instance | Constant/Variable |
| id | private | instance | constant (set once) |
| battery | private | instance | variable (mutable) |
| batteryCapacity | private | instance | constant (set in constructor) |
| marketPrice | private | class (static) | constant (final) |
| MAX\_BATTERY\_CAPACITY | private | class (static) | constant (final) |

A black and white screen with blue text

AI-generated content may be incorrect.

**Instance variables** (unique to each entity) are declared **private,** example:

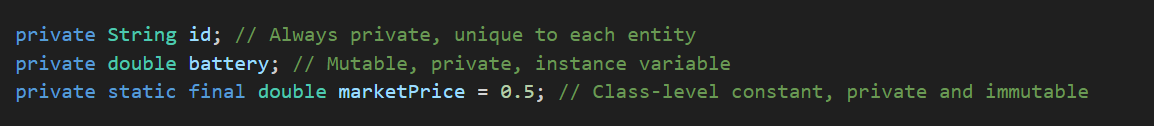
A black background with blue and green text

AI-generated content may be incorrect.

**Step 3: Constant or Variable**

* **Constants** are marked final to enforce immutability.
* **Mutable variables** (like battery) are private and only modifiable through controlled methods.

**Code Example:**



Explanation

* **All instance variables are private**, preventing any external class from directly accessing or modifying the internal state.
* **Constants are private static final**, ensuring immutability and class-level sharing without external visibility.
* **The 3-step engineering process** is applied attribute, based on its purpose and usage in the class.

Motivation (Reasoning)

* **id:** Private to uniquely identify each entity without external tampering.
* **battery & batteryCapacity:** Private to prevent invalid direct manipulation (e.g., negative battery levels), maintaining simulation integrity.
* **energySource & transformer:** Private to hide implementation details of energy management.
* **Financial and daily tracking variables (totalCost, dailyProduced, etc.):** Private to ensure that only the entity’s own methods update these values, preserving consistency.
* **marketPrice & MAX\_BATTERY\_CAPACITY:** Private static final to enforce consistent, unchangeable configuration accessible only internally.

Benefits Of Information Hiding

* **Improves Security:**  
  By restricting access to **internal data** and **implementation details,** information hiding protects **sensitive data** from unauthorized or accidental modification.
* **Reduces Complexity:**  
  Hiding **internal details** simplifies the interface exposed to other parts of the program, making it easier to understand and use.
* **Enhances Maintainability:**  
  Internal implementation can be changed without **affecting** external code, since only the interface is exposed.
* **Supports Robustness:**  
  **Controlled access** prevents invalid states and enforces correct usage through well-defined interfaces.

Information hiding = What is hidden.

Encapsulation = How data and behavior are combined and controlled.

**In conclusion**, my project applies the principle of **information hiding** by keeping all attributes **private** and following the **three-step process** to clearly distinguish between **constants** and **variables** at both the **class** and **instance** levels.

**Encapsulation** is achieved by pairing these **private attributes** with carefully designed methods that control how the **data is accessed** and **modified,** including **validation checks** to **maintain the integrity** of the object’s state. By clearly **separating** information hiding from encapsulation and applying both correctly, code becomes more reliable, easier to maintain, and secure.

**------------------------------------------------------------------------------------------------------------**

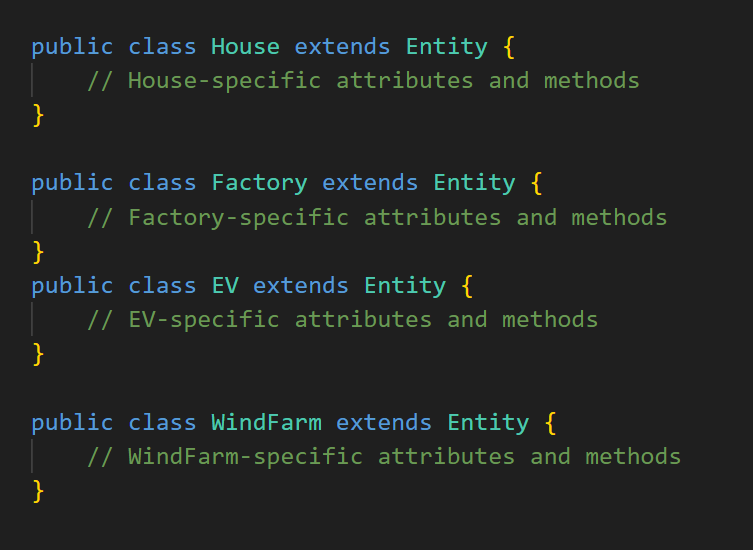
4.3 Inheritance:

Definition

**Inheritance** is an **object-oriented programming principle** that allows the creation of new classes based on existing ones. The new subclass inherits **properties** and **behaviors** from the superclass, enabling code reuse and establishing a hierarchical relationship between classes.

How I Applied It (Example 01)

In my project, I created **abstract base** class such as Entity to define common **attributes** and **behaviors**. I then extended these base classes with concrete subclasses which inherit and specialize the functionality.



Example 02

Just like House, Factory, EV, and WindFarm extend the Entity class, now various energy source types (e.g., SolarPanel, WindTurbine, Battery) also **extends** the**EnergySource** base class.

A screenshot of a computer program

AI-generated content may be incorrect.

Motivation for Using Inheritance

Inheritance allows me to **avoid code duplication** by defining common attributes and behaviors once in the **base classes** and **reusing** them across multiple specialized **subclasses** such as House, Factory, EV, SolarPanel, and WindTurbine. This promotes a **clear and logical organization** of related types, making the code easier to understand and maintain.

By extending base classes, I can **easily add new entity or energy source types** without modifying existing code. Subclasses can also **override or extend functionality** to implement specific behaviors, enabling flexibility and scalability in the project.

Overall, inheritance improves **code reuse, maintainability, and extensibility**, which are essential for managing complexity in a growing system.

Benefits Of Inheritance

* **Code Reusability:**  
  Inheritance allows subclasses to **reuse** code from their parent classes, eliminating redundancy and saving development time.
* **Improved Maintainability:**  
  Changes made in the superclass automatically propagate to subclasses, reducing the need for repetitive updates and simplifying maintenance.
* **Clear Hierarchical Structure:**  
  Inheritance establishes an **“is-a”** relationship between classes, organizing related classes into a logical hierarchy that is easier to understand.
* **Supports Polymorphism:**  
  Enables objects of **different subclasses** to be treated as instances of a common superclass, allowing flexible and extensible code.
* **Extensibility:**  
  New functionality can be added by creating subclasses without modifying existing code, supporting the open/closed principle.
* **Encapsulation Support:**  
  Inheritance helps **encapsulate** shared behavior in a base class, promoting cleaner and modular code design.

**------------------------------------------------------------------------------------------------------------**

4.4 Composition:

Definition

**Composition** is an **object-oriented design principle** where complex objects are built by combining **simpler objects**. Instead of inheriting from other classes, a class contains instances of other classes as components, modeling **“has-a”** relationships.

How I Applied It (Example 01)

In my project, each Entity contains an EnergySource and a Transformer as components rather than inheriting from them. This allows entities to be composed of different energy sources and transformers flexibly.

A screen shot of a computer

AI-generated content may be incorrect.

Each Entity object contains two important components: an EnergySource (such as solar panels, wind turbines, or hydro energy) and a Transformer (which connects the entity to the electrical grid).

I chose composition here because these components **are parts of an entity, not types of entities** themselves. Therefore, instead of using inheritance, I modeled this as a **“has-a”** relationship.

Example 02

A screen shot of a computer program

AI-generated content may be incorrect.

Similarly, the SmartGridManager class uses **composition** by maintaining **lists** of **Entity** and **Transformer** objects.

It “**has many”** entities and transformers but does not inherit from them. Instead, it manages and coordinates these components, further promoting modularity and separation of concerns.

Example 03

A computer screen shot of a computer

AI-generated content may be incorrect.

The WeatherEvent class demonstrates composition by holding references to key components: SmartGridManager TimeManager and WeatherStation.

Instead of inheriting from these classes, WeatherEvent **has** these objects as private members, allowing it to interact with and coordinate their behaviors.

Similar Usage in Other Classes

Many other classes in the project follow the same pattern, **holding references** to collaborators via **composition** rather than **inheritance**.

This consistent use of composition supports modular **design, separation of concerns**, and **easier maintenance** across the project.

Benefits Of Inheritance

* **Code Reuse:** Composition allows reuse of existing classes without relying on inheritance, making code more modular and maintainable.
* **Flexibility:** Components can be swapped or changed at runtime, enabling dynamic behavior and easier adaptation to new requirements.
* **Loose Coupling:** Classes are less dependent on each other, reducing the risk of side effects when modifying code.
* **Better Design:** Models real-world “has-a” relationships naturally, improving clarity and organization.
* **Improved Testability:** Smaller, focused components are easier to test in isolation.
* **Avoids Inheritance Issues:** Prevents problems like rigid hierarchies and the diamond problem common with deep inheritance chains.
* **Modularity and Extensibility:** Supports building complex systems from simple parts, making future extensions easier.

**------------------------------------------------------------------------------------------------------------**

4.5 Abstraction:

**Abstraction** is the principle of defining **common interfaces** or **abstract classes** to capture **shared behavior** and characteristics, while leaving the specific implementation details to the subclasses. It helps focus on ***what* an object does** rather than ***how* it does it.**

How I Applied It (Example 01)

**Entity** is the main base class for all types of entities such as houses, factories, electric vehicles (EVs), and wind farms. It contains common properties and methods shared by all these entities.

However, some specific behaviors, like how much energy each entity consumes, are left undefined in the base class and must be implemented by each subclass through an abstract method called getConsumptionAmount().

A black screen with text

AI-generated content may be incorrect.

Example 02

A black screen with text on it

AI-generated content may be incorrect.

**EnergySource** is an abstract class representing **different energy producers**. It requires subclasses to **implement produceEnergy**(), which varies depending on the source type (solar, wind, hydro):

Example 03

A screen shot of a computer code

AI-generated content may be incorrect.

**Event** is a key abstract class in the simulation that defines the structure for all events. It declares the**abstract** **process() method**, which each subclass implements to specify what happens when that event occurs. The event hierarchy is quite extensive, with 5-6 subclasses such as:

1. **ProductionEvent**: handles energy production by entities
2. **ConsumptionEvent:** manages energy consumption
3. **TradeEvent**: deals with energy buying and selling
4. **WeatherEvent**: updates weather conditions affecting production
5. **FaultEvent**: simulates faults like transformer failures
6. **RepairEvent**: simulates repairs to restore service

Benefits Of Inheritance

**Simplifies Code:** Hides complex details and shows only what’s necessary, making the code easier to understand.

**Promotes Reusability:** Provides common templates (abstract classes/interfaces) that different subclasses can use.

**Increases Flexibility:** Allows changing implementations without affecting other parts of the program.

**Improves Maintainability:** Makes it easier to update and fix code since details are separated from usage.

**Supports Modularity:** Helps organize code into clear, independent part

**------------------------------------------------------------------------------------------------------------**

4.6 Subtyping

**Subtyping** means that one type (a **subclass**) can be used in place of another type (its **parent class or interface**) without breaking the program.

How Interfaces Implemented by Entity (Example 01)

**IEnergyProducer:** Defines the ability to produce energy, implemented by entities like solar panels or wind farms.

**IEnergyConsumer:** Defines the ability to consume energy, implemented by all entities that use energy, such as houses and factories.

**ITradeable:** Defines the ability to trade energy with the grid, including methods for buying and selling energy.

**WeatherObserver:** Allows entities to receive updates about weather changes and adjust their behavior accordingly.

By implementing these interfaces, Entity objects can be handled **polymorphically** based on behavior rather than their concrete class.

A screen shot of a computer code

AI-generated content may be incorrect.

**Note:** Since Entity is an abstract class implementing these interfaces, it does not have to override all interface methods itself. Instead, it can defer the implementation to its concrete subclasses. However, any concrete subclass must override and implement all methods defined in the interfaces it inherits.

Demonstration of Subtyping and Multityping in Client Code (Example 2)

In the client program, I demonstrate subtyping and multityping by accessing a **single House object** through **different types of references:**

**Subtyping:** The House object is **referenced** as its superclass **Entity**, allowing access to general entity methods like **getId().**

**Multityping:** The same object is also referenced through **multiple interfaces** (IEnergyProducer, IEnergyConsumer, ITradeable, WeatherObserver), each representing a **different role** the entity can play.

A computer screen shot of a program

AI-generated content may be incorrect.

This design enables the **client to interact** with the object based on the role it needs at that moment, **without depending** on the **concrete House class**. It promotes flexibility and clean separation of concerns.

Benefits Of Subtyping

* **Flexibility:** Allows objects to be used in different contexts based on the interface or superclass they implement.
* **Code Reusability:** Enables sharing common behavior through superclasses or interfaces, reducing duplication.
* **Polymorphism:** Supports treating different objects uniformly, making the code more extensible.
* **Modularity:** Helps separate concerns by defining roles through interfaces, improving organization.
* **Type Safety:** Ensures objects conform to expected behaviors, reducing errors.
* **Easier Maintenance**: New subtypes can be added without changing existing code, supporting scalability.

**------------------------------------------------------------------------------------------------------------**

4.7 Polymorphism:

**Polymorphism** allows objects to be treated as instances of their parent class or interface, **enabling flexible and dynamic** behavior in the program. In my project, I used several types of polymorphism:

Type 1. Method Overloading in the Transformer Class

In my project, I used **method overloading** in the Transformer class to provide multiple ways to create **transformer objects** depending on the information available at the **time of instantiation.** This improves flexibility and readability.

A screen shot of a computer code

AI-generated content may be incorrect.

Explanation

Here, the two constructors have the same name Transformer **but different parameter** lists:

* One takes only the **id** and sets the transformer as **operational by default.**
* The other takes both id and operational status, **allowing more control.**

Benefits of Overloading?

* **Flexibility:** Allows creating objects or calling methods in different ways using the same method name, improving ease of use.
* **Improved Code Clarity:** Using the same method name for related operations makes the code more readable and intuitive.
* **Reduced Code Duplication:** Centralizes initialization or processing logic, helping avoid repetitive code.

Type 2. Parametric Polymorphism

I implemented parametric polymorphism by creating a generic class **EntityRegistry<T extends Entity>,** which acts as a **central container** to manage entities of any subclass of Entity.

A screen shot of a computer code

AI-generated content may be incorrect.

Explanation:

* The generic type parameter **<T extends Entity>** restricts T to be any subclass of Entity, ensuring type safety.
* This allows the registry to hold houses, factories, EVs, or any other entity subtype in a single structure.
* Methods like **register()** and **getEntities()** work uniformly for all entity types without needing separate code for each.
* Returning an **unmodifiable list** prevents external modification, preserving encapsulation.

Benefits of Generics:

* **Reusability:** The same registry class handles all entity types.
* **Type Safety:** Compile-time checks prevent adding invalid types.
* **Maintainability:** Centralized management of entities simplifies the codebase.

Type 3. Inclusion Polymorphism

In my project, I use **inclusion polymorphism** by storing all entities in a single **List<Entity>** within the SmartGridManager class:

A computer screen shot of text

AI-generated content may be incorrect.

Explanation

* This allows the **simulation** to manage diverse entity types: such as houses, factories, and electric vehicles uniformly. The list holds references of the base type Entity, but at runtime, each object behaves according to its actual subclass.
* Furthermore, the simulation employs an event-driven model where **different event types** (WeatherEvent, FaultEvent, ProductionEvent, etc.) inherit from a **common abstract base class Event**. Events are **scheduled** and processed in **chronological order.**

A screen shot of a computer

AI-generated content may be incorrect.

Here, although **e** is declared as type **Event**, the **overridden process()** method executed depends on the actual runtime type of the event object. This is the **dynamic dispatch behavior** enabled by **inclusion polymorphism**.

Why This Design

* **Inclusion Polymorphism**: Using **List<Entity>**enables the manager to handle all entity types transparently, without needing to know their specific classes. Similarly, processing events through the base Event type allows uniform handling of diverse event types.
* **Flexibility**: The system can handle objects of different concrete types through common superclass or interface references, enabling dynamic and adaptable behavior.
* **Extensibility:** New entity or event types can be added without modifying the existing management or processing code, supporting future growth.
* **Separation of Concerns**: Entities encapsulate their specific energy behaviors, while the manager and event calendar handle scheduling and control flow, promoting modularity and clean architecture.
* **Dynamic Behavior:** Method calls are resolved at runtime, ensuring that each object behaves according to its actual type.

Type 4. Coercion: Casting from Object to Entity

In my project, I demonstrated **coercion** by **explicitly casting** an object from a general type **(Object)** back to a more specific type **(Entity).** This is useful when dealing with generic or loosely typed references and needing to access subclass-specific methods or fields.

A screen shot of a computer code

AI-generated content may be incorrect.

Explanation:

* The variable obj is declared as **Object**, **the root superclass** of all classes in Java.
* Before casting, I use the **instanceof operator** to ensure obj is an instance of Entity to avoid a **ClassCastException.**
* After confirming, I cast obj back to **Entity** so I can call the getId() method specific to the Entity class.
* This explicit type conversion is called **coercion** or **type casting**.

Why This Is Useful:

1. **Casting** allows recovering the original type to access specialized behavior.
2. It demonstrates understanding of **Java’s type system** and **safe casting practices.**

4.8: Exceptions

Defination

An **exception** is an event that occurs during the execution of a program which disrupts the normal flow of instructions. It represents an unexpected or erroneous condition

Custom Exception: BatteryException

I created a **custom checked exception**, **BatteryException**, to specifically handle battery state violations such as:

A screen shot of a computer code

AI-generated content may be incorrect.

* **Battery Overflow:** Attempting to **store** more energy than the battery’s capacity.
* **Battery Underflow:** Attempting to **consume** more energy than is currently available.

These exceptions are explicitly **thrown** in the methods responsible for modifying battery levels:

A computer screen with colorful text

AI-generated content may be incorrect.

4.9: Exception Handling

Defination

Exception Handling is a programming mechanism that allows a program to detect, catch, and respond to runtime errors or exceptional conditions in a controlled way. It helps maintain normal program flow by managing errors gracefully instead of abruptly terminating the program.

Explanation

The methods produceEnergy(), consumeEnergy(), and tradeEnergy() **catch these exceptions internally** to handle errors gracefully:

* They log informative messages indicating the nature of the problem.
* They prevent the simulation from terminating abruptly.
* This approach allows the simulation to continue running and handle other entities or events without interruption.

Example from produceEnergy():

A screen shot of a computer program

AI-generated content may be incorrect.

Example from tradeEnergy():

A computer screen shot of a program code

AI-generated content may be incorrect.

Why Exception Handling Was Used

* Ensures the simulation handles **critical battery errors** without crashing.
* Validates battery operations to prevent invalid states like **overflow or underflow.**
* Allows graceful recovery by **logging errors** and **continuing execution**.
* Provides clear, **specific error messages** for easier debugging.
* Keeps validation separate from core logic for cleaner, maintainable code.

5.0: Observer Pattern

The **Observer pattern** is used to enable entities to **receive updates** about **weather changes dynamically** without tightly coupling them to the weather source. This design promotes loose coupling, modularity, and scalability.

Explanation

* The WeatherStation acts as the **subject (observable)** that maintains a list of observers and notifies them whenever the weather changes.
* The Entity class implements the WeatherObserver interface, allowing each entity to **react independently** to weather updates.

Subject

A computer screen shot of a program

AI-generated content may be incorrect.

* **Encapsulation:** The **weatherFactor** and **observer list** are private, protecting internal state.
* **Abstraction:** Observers only need to know about the **updateWeather() method**, not the internal workings of WeatherStation.

Entity (Observer)

A screen shot of a computer program

AI-generated content may be incorrect.

* **Polymorphism:** Multiple entity types can implement **WeatherObserver** and respond differently to weather updates.
* **Controlled State Update:** The latestWeather variable is updated only through the observer interface method, maintaining encapsulation.

Why This Pattern?

* It allows **dynamic registration and notification** of entities without modifying their code.
* Entities react to weather changes **independently and asynchronously**, improving flexibility.
* Facilitates **extensibility**: new entity types can be added without changing WeatherStation.

6.0: Simulation Results

Daily Summary

The daily summary provides a detailed snapshot of the simulation at the end of **each simulated day.**

The **printDailySummary()** method is responsible for generating this report. Below is an example output from a **typical day** in the simulation:

Example Day 01:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Entity | Produced (kWh) | Consumed (kWh) | Net (kWh) | Income ($) | Cost ($) | Profit ($) | Battery (kWh) |
| House1 | 34.26 | 22.75 | 11.50 | 14.50 | 0.00 | 14.50 | 17.50 |
| Factory1 | 30.26 | 38.89 | -8.63 | 16.93 | 0.00 | 16.63 | 17.50 |
| EV1 | 0.00 | 8.71 | -8.71 | 0.00 | 0.00 | 0.00 | 16.83 |
| WF1 | 21.33 | 0.00 | 21.33 | 21.91 | 0.00 | 21.91 | 17.50 |

This summary illustrates the **day-to-day dynamics of the system**, showing how **entities interact** with the **energy grid** and respond to changing conditions such as weather and demand.

cumulative summary

The **printTotalSummary()** method generates this comprehensive overview. Below is an example of the **cumulative summary output** at the end of the simulation:

Example Day 04:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Entity | Produced (kWh) | Consumed (kWh) | Net (kWh) | Income ($) | Cost ($) | Profit ($) | Battery (kWh) |
| House1 | 143.96 | 85.21 | 58.74 | 38.12 | 0.00 | 38.12 | 17.50 |
| Factory1 | 126.52 | 150.83 | -24.31 | 16.93 | 5.34 | 11.59 | 12.50 |
| EV1 | 0.00 | 35.68 | -35.68 | 0.00 | 11.5 | -11.59 | 12.50 |
| WF1 | 89.38 | 0.00 | 89.38 | 55.94 | 0.00 | 55.94 | 17.50 |

**SUMMARY**

This summary offers a big-picture view of the system’s performance and efficiency, helping to evaluate the **overall effectiveness** of the **energy management** and **trading strategies** implemented in the simulation.

7.0: Logic and Extensions

Core Simulation Logic

The simulation models a **peer-to-peer smart grid** where **multiple entities** such as houses, factories, and electric vehicles produce, consume, and trade energy. The simulation operates as **discrete event system:**

* Events (such as energy production, consumption, faults, and repairs) are **scheduled** and **managed** by an **EventCalendar (a priority queue).**
* Each event is **processed** in **chronological order**, and the system’s state is updated accordingly.
* Entities **interact** with the grid and each other by producing energy (influenced by weather), consuming energy, and trading energy based on their battery levels and market prices.

A screen shot of a computer screen

AI-generated content may be incorrect.

This structure allows for **flexible**, **realistic simulation** of asynchronous and **interdependent events.**

Strategic and Game-Theoretic Elements

**Trading Logic:** Each entity evaluates its **battery state** and **market price** to decide whether to **buy from** **or sell energy to the grid**. Thresholds for buying and selling are **dynamically** set based on battery capacity, simulating rational, self-interested behavior.

**Market Price:** The SmartGridManager sets a market price, and entities use this to calculate the financial outcome of trades.

**Weather Influence:** Weather factors, updated via the **Observer pattern**, directly affect renewable energy production, adding **realism** and **variability** to the simulation.

Extensions and Custom Features

* **Custom Exception Handling:**The**BatteryException** class is thrown when an entity attempts to **overfill** or **overdraw** its battery, ensuring robust error handling and state management.
* **Observer Pattern for Weather Updates:**  
  Entities implement WeatherObserver and are **notified** by the **WeatherStation**whenever weather changes, which in turn affects their energy production.
* **Strategy Pattern for Trading and Consumption:**  
  The **tradeEnergy()** and **getConsumptionAmount()** methods vary by entity type and battery capacity, encapsulating different strategies for different entities.
* **Fault and Repair Events:**  
  Transformers can experience **faults** (via FaultEvent) and be repaired (RepairEvent), affecting the ability of entities to trade energy and adding randomness and resilience testing to the simulation.
* **Extensible Entity Registry (Parametric Polymorphism):**  
  The **EntityRegistry<T extends Entity>** class allows for scalable management of heterogeneous entities using Java generics.

----------------------------------------------------------------------------------

## End