Software Design Document

Earthquake Alert System

Use Case: Earthquake Detection & Notification (Observer Pattern)

Team Members:

- İsmail Altar Büyükdoğan
- Talha Çetin
- Samet Bozdoğan
- Yakup Çakır

Document-Specific Task Matrix:

See ‘docs/TASK_MATRIX.md’

Table of Contents

System Overview

The Earthquake Alert System is a simplified backend simulation designed to demonstrate the Observer Design Pattern in a real-world-inspired use case. The system models how various components (observers) can be notified automatically when an earthquake is detected by a central subject (sensor).

This implementation does not include a frontend, external API integration, or database storage. Instead, it focuses purely on backend logic, making the internal structure of the design pattern clearly visible and suitable for academic presentation and understanding.

The simulated scenario includes:

- An earthquake sensor that detects seismic events
- A notification service that alerts users
- A logger that records earthquake data
- A map module that updates a visual layer (simulated)

All observer components are updated in real time through a decoupled subscription mechanism, showcasing the scalability and maintainability benefits of the Observer Pattern.

System Context

This project simulates a backend part of a possible earthquake alert system. In a real scenario, the system might connect to live earthquake data from an official source like a government agency. However, in this version, we manually simulate earthquake events.

The system has one central part: the earthquake sensor. When a simulated earthquake happens, the sensor informs all the services (observers) that are interested in this event. These services then respond in their own way — like sending an alert, logging it, or updating a map.

The whole system runs locally and is not connected to the internet, a database, or a user interface. The main goal is to show how the different parts can work together using the Observer Design Pattern.

Key Features and Functionality

- The system can simulate an earthquake with details like location, magnitude, depth, and time.

- When an earthquake is simulated, all registered observers are notified.

- Each observer has a different job:

  - The notification service prints an alert message.
  - The logger service records the earthquake details in a structured format.
  - The map updater adds a marker (simulated) based on the earthquake data.

- Observers can be added or removed at any time without changing the sensor itself.

- Everything runs in the terminal using TypeScript, and all responses are printed to the console.

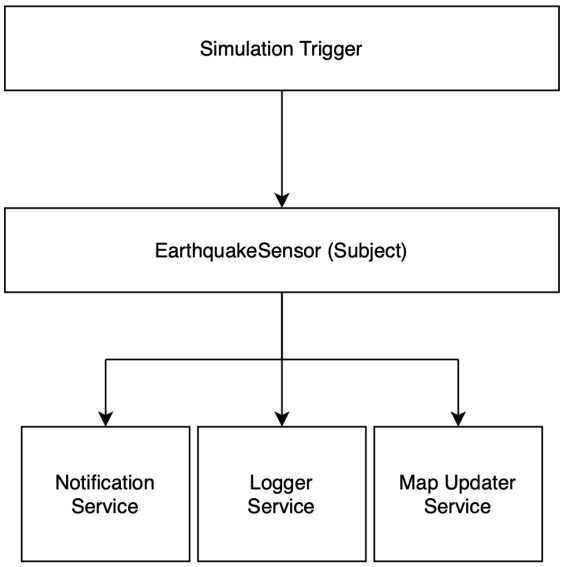
Assumptions and Dependencies

- The earthquake data is not coming from a real API. It is created manually in the simulation.
- All services (observers) are assumed to be local functions that react instantly.
- The system runs in a local Node.js environment using TypeScript.
- There is no frontend, no database, and no external services involved.
- It is assumed that all observers are trusted and do not fail during updates (no error handling is included in this version).
- The goal is not to build a full application, but to clearly show how the Observer Pattern works in a simple way.

Architectural Design

System Architecture Diagram (High-Level)

The system has a simple structure. It includes one subject (the earthquake sensor) and three observers (notification, logger, map updater). All parts run inside a single process. There are no networks, databases, or UI layers.



This shows how one central unit (sensor) notifies multiple independent services.

Architectural Pattern

This project uses the Observer Design Pattern. It is not a full software architecture like MVC or microservices. Instead, it focuses on showing a reusable design solution for how components can communicate without being tightly connected.

Rationale for Architectural Decisions

The Observer Pattern fits well because:

- One event (earthquake) leads to many different reactions.
- Each observer handles a separate task (single responsibility).
- New services can be added without changing the existing code.

Keeping everything simple (no servers or UI) makes it easier to focus on the design pattern itself.

Component Design

Subsystems and Modules

EarthquakeSensor (Subject): Detects earthquakes and notifies all registered observers.

NotificationService (Observer): Prints a message that simulates alerting the users.

LoggerService (Observer): Logs earthquake data with details like location and severity.

MapUpdaterService (Observer): Pretends to update a map by storing the event as a marker.

Responsibilities of Each Component

EarthquakeSensor: Keeps a list of observers and calls their ‘update()’ methods when an earthquake is detected.

Observers: Each one focuses on one task and handles the update in its own way.

Interfaces Between Components

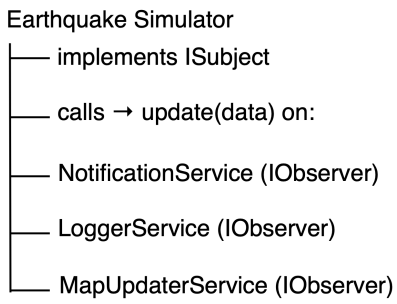
All observers implement a shared interface called ‘IObserver’, which has a single method: ‘update(data)’.

The sensor uses the ‘ISubject’ interface, which includes ‘addObserver()’, ‘removeObserver()’, and ‘notifyObservers()’ methods.

This setup ensures that the sensor and observers are loosely connected, which makes the system easier to change or grow.

Component Diagram

Since this is a simple simulation, the relationships can be described like this:



Data Design

Data Model / ER Diagram

The system uses a single TypeScript interface to represent earthquake data:

```
interface EarthquakeData {
  magnitude: number;
  location: string;
  timestamp: Date;
  depth: number;
}
```

This structure is used to pass information from the subject to all observers.

There is no database or entity relationship model in this simulation. All data is kept in memory and printed to the console.

Data Storage

No external data storage is used.

Observers like MapUpdaterService store data in local arrays for demonstration only.

When the program ends, all data is lost.

Data Flow

EarthquakeSensor creates an EarthquakeData object.

This object is passed to each observer through the update(data) method.

Observers handle the data as needed (logging, displaying, etc.)

Data Validation Rules

There is no strict validation in this version.

It is assumed that all fields (magnitude, location, timestamp, depth) are present and correct.

Design Patterns

Applied Design Pattern

This project uses the Observer Design Pattern.

- The main subject is ‘EarthquakeSensor’.

- All observers (NotificationService, LoggerService, MapUpdaterService) implement the ‘IObserver’ interface.

- The sensor keeps a list of observers and notifies them when an earthquake is detected.

Context and Justification

This pattern is a good choice for this scenario because:

- One central event (earthquake detection) leads to multiple actions in different parts of the system.

- Each observer works independently and has one clear job.

- New observers can be added easily without touching the existing code.

- The subject does not need to know what each observer does — it just calls ‘update()’.

This helps us build a system that is more modular, easier to test, and easier to expand in the future.

Implementation Notes

- The project is written in TypeScript and runs on Node.js using ‘ts-node’.
- No external libraries are used — everything is written with native TypeScript features.
- The Observer Pattern is implemented using interfaces (‘ISubject’, ‘IObserver’) and classes.
- The system runs locally through a single entry file: ‘src/index.ts’.
- Earthquake simulation is done by manually calling ‘detectEarthquake()’ with sample data.
- All outputs are printed to the console. There is no database, frontend, or API.

This setup is designed to keep the focus on the design pattern, without distractions from unrelated technologies.

User Interface Design

This project does not include a user interface at this stage.

All outputs are printed to the terminal for demonstration purposes. The system is focused on backend behavior and the implementation of the Observer Pattern.

In future versions of the project, a mobile interface using React Native may be added. That interface could connect to this backend and visually display alerts, logs, and map updates.

External Interfaces

Currently, this system does not use any external interfaces.

Earthquake data is manually simulated inside the code for demonstration purposes. No real-time APIs, web services, or databases are connected.

In future versions, the system is expected to connect to official earthquake data providers such as AFAD (Turkey), Kandilli Observatory, or USGS (United States Geological Survey). These APIs would allow the system to receive real earthquake data and trigger observers automatically, without manual input.

Performance Considerations

This version of the system is a simple simulation and does not have any performance constraints. All processing is local and runs instantly using console outputs.

However, in future versions where the system connects to real-time earthquake data from external APIs (e.g., AFAD, Kandilli, USGS), performance will become more important. The system will need to:

- Handle data updates in near real-time
- Avoid blocking delays when receiving or processing earthquake data
- Notify all observers quickly, especially if used in a real alert system

As new features are added, scalability and low-latency response will be key concerns to ensure reliability.

Error Handling and Logging

This version of the system does not include advanced error handling. It is assumed that all components behave as expected and no runtime errors occur.

- There are no try-catch blocks or recovery mechanisms.

- If an observer fails, it will affect the system unless handled externally.

For logging:

- The ‘LoggerService’ class acts as a basic logging component.
- It prints structured earthquake data to the console.
- The logs include information like magnitude, location, timestamp, and calculated severity.

In a future version, error handling should be improved to catch unexpected failures (e.g., network errors, null data). Logging can also be expanded to use files or external logging services.

Design for Testability

The system is designed with simplicity and modularity in mind, which makes it easy to test.

- Each observer class (‘NotificationService’, ‘LoggerService’, ‘MapUpdaterService’) has a single responsibility and can be tested independently.

- The subject class (‘EarthquakeSensor’) can be tested by simulating an earthquake and checking if all observers are updated correctly.

- Since no external systems are used, the logic can be tested without needing a network connection, database, or frontend.

In future development, formal unit tests (e.g., using Jest or Mocha) can be added to verify the behavior of each class under different scenarios.

Deployment and Installation Design

This system is not deployed on any server or cloud environment. It is intended to be run locally for educational purposes.

Environment Configuration

Node.js must be installed on the local machine.

TypeScript and ts-node are used for compiling and running the code.

Packaging and Dependencies

No runtime dependencies are used.

Development dependencies include:

- ‘typescript’
- ‘ts-node’

This setup keeps the project lightweight and easy to run in any environment that supports Node.js.

Change Log

- Created basic project structure using TypeScript
- Implemented the Observer Pattern with:
  1. EarthquakeSensor (Subject)
  2. NotificationService (Observer)
  3. LoggerService (Observer)
  4. MapUpdaterService (Observer)
- Simulated earthquake detection and observer updates
- Printed structured outputs to the console
- Documented the system design and responsibilities

Future Work / Open Issues

This project is a simplified simulation focused on demonstrating the Observer Pattern. In future versions, the system can be expanded in several ways:

- Connect to real earthquake data sources such as AFAD, Kandilli, or USGS APIs
- Improve error handling, including recovery from failed observer updates
- Add a frontend application using React Native to show alerts and maps visually
- Save earthquake logs to a database or file system for persistence
- Write unit tests to verify each component
- Add different observer types, such as email or SMS alert services

These improvements would help turn this simulation into a more realistic and useful system.