Object Oriented programming and software engineering – Lab 17

Adam Korytowski – 2025

I Design patterns

Design patterns are standard, reusable solutions to common problems that arise in software design. They represent best practices that have been discovered and refined over time by experienced software developers. Rather than solving a problem from scratch each time, design patterns provide a proven framework for tackling common challenges in a structured way.

Design patterns are not pieces of code; rather, they are conceptual templates that guide developers on how to structure their code to solve specific types of problems. They help to create more maintainable, scalable, and flexible software.

There are 23 classic design patterns, most notably cataloged in the book *"Design Patterns: Elements of Reusable Object-Oriented Software"* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. These patterns fall into three main categories:

- Creational Patterns: Deal with object creation mechanisms (e.g., Singleton, Factory Method).
- Structural Patterns: Concerned with organizing classes and objects to form larger structures (e.g., Adapter, Composite).
- Behavioral Patterns: Focus on communication between objects (e.g., Observer, Strategy).

II Observer design pattern

The Observer Pattern is used when we need to allow one object (the subject) to notify multiple other objects (the observers) about changes in its state. The key idea is to establish a relationship between the subject and its observers, so that when the subject's state changes, all registered observers are automatically updated.

What is it for?

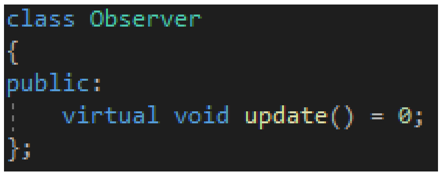
- Loose coupling: The subject does not need to know the details of its observers. It only needs to notify them of any changes, and observers take care of how to handle those changes.

- Dynamic interaction: Observers can be added or removed at runtime without changing the subject's code, making it flexible and extensible.
- Multiple views: The Observer Pattern is often used in scenarios like user interfaces, where a change in the underlying model (e.g., a character's stats in a game) needs to be reflected across multiple views (e.g., a UI displaying the character's health and mana).

Example: Applying Observer in Game Code:

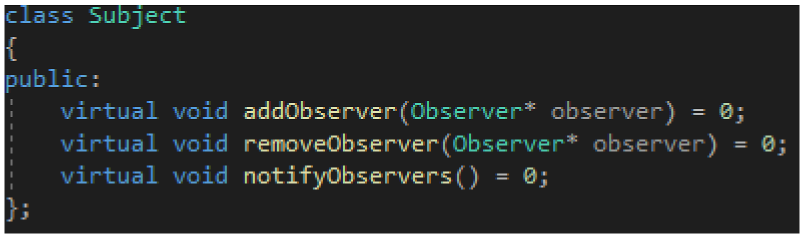
Imagine you have a base class Character with derived classes like Warrior and Mage. The Observer Pattern can be applied if you want other parts of the program (such as the UI, a logging system, or a battle log) to be notified whenever a change happens to the character's stats (e.g., health, mana, status effects).

1. Observer Interface:
   Every observer needs to implement the update method, which will be called whenever the observed object changes.



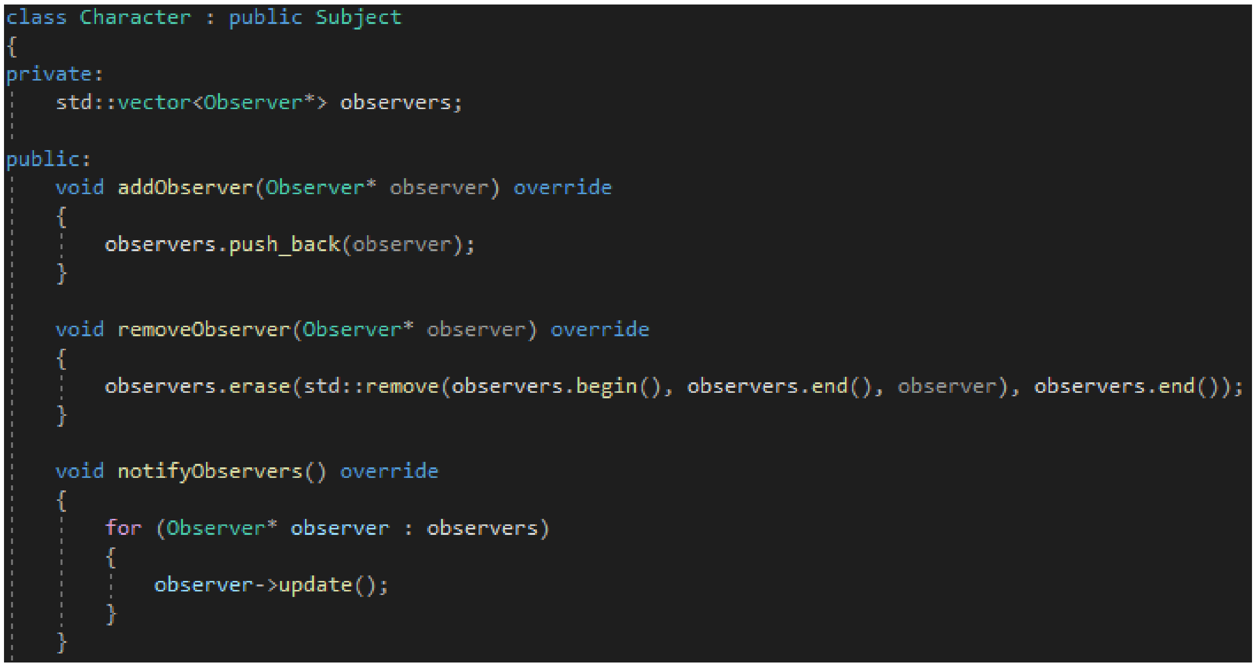
```cpp
class Observer
{
public:
    virtual void update() = 0;
};
```

2. Subject Interface:
   The Character (or derived classes like Warrior and Mage) will act as the subject. It will have methods to add, remove, and notify observers.

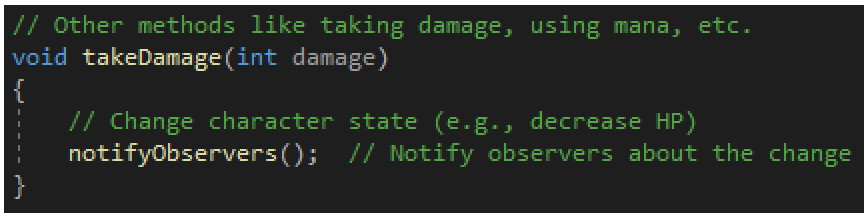


```cpp
class Subject
{
public:
    virtual void addObserver(Observer* observer) = 0;
    virtual void removeObserver(Observer* observer) = 0;
    virtual void notifyObservers() = 0;
};
```

3. Character Class (and subclasses like Warrior and Mage):
   The character will manage the observers and notify them when a change occurs (e.g., damage taken, stat changes).

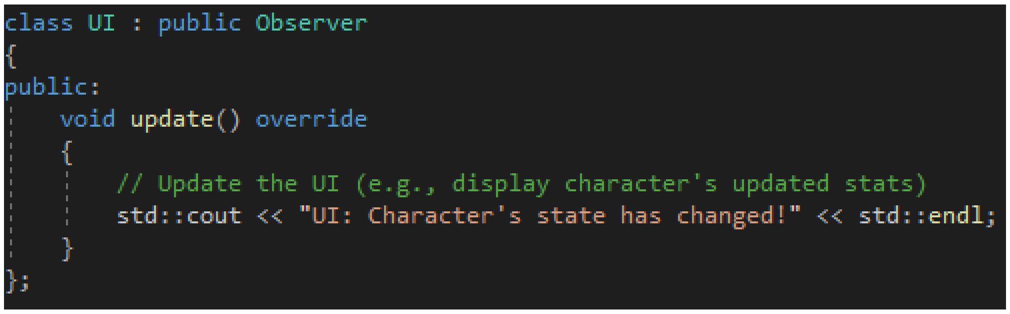


```cpp
class Character : public Subject
{
private:
    std::vector<Observer*> observers;

public:
    void addObserver(Observer* observer) override
    {
        observers.push_back(observer);
    }

    void removeObserver(Observer* observer) override
    {
        observers.erase(std::remove(observers.begin(), observers.end(), observer), observers.end());
    }

    void notifyObservers() override
    {
        for (Observer* observer : observers)
        {
            observer->update();
        }
    }
```



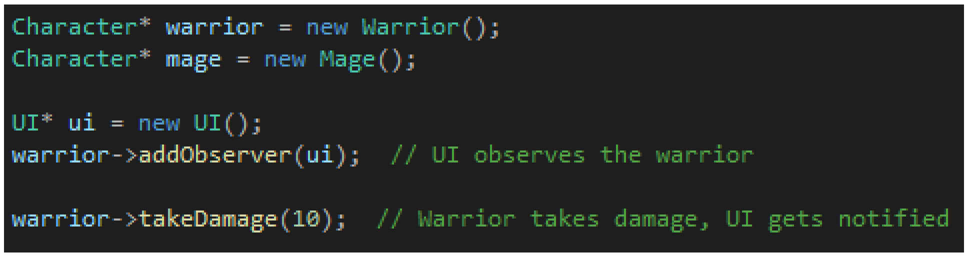
```cpp
// Other methods like taking damage, using mana, etc.
void takeDamage(int damage)
{
    // Change character state (e.g., decrease HP)
    notifyObservers();  // Notify observers about the change
}
```

4. Observer (e.g., UI or Log):
   The observer will react to the state change of the character, such as updating the UI or logging the change.



```cpp
class UI : public Observer
{
public:
    void update() override
    {
        // Update the UI (e.g., display character's updated stats)
        std::cout << "UI: Character's state has changed!" << std::endl;
    }
};
```

5. Usage Example:



```cpp
Character* warrior = new Warrior();
Character* mage = new Mage();

UI* ui = new UI();
warrior->addObserver(ui);  // UI observes the warrior

warrior->takeDamage(10);  // Warrior takes damage, UI gets notified
```

III Tasks:

- Implement Observer design pattern in your existing code (6 pts)
- Update your code in your git repository (2 pts)