



Design Patterns

(Observer + Bridge)

CSE 4513: SOFTWARE ENGINEERING & OBJECT ORIENTED DESIGN

27 January, 2026



The Team

Anjim Hossain

220041101

Mahiul Kabir

220041109

Ridita Alam

220041110

Nayeemul Hasan

220041125

Saba Atharique

220041130

Sadman Shahrier

220041133

Tasnif Emran

220041135

Alfi Shahrin

220041153

Obidit Islam

220041154

Madina Akbari

220041162

O B S E R V E R

Behavioural Design Pattern - Why?

- Focuses on object interaction, communication, and responsibility
- Observers react automatically to state changes without the need to know data-types
- Promotes event-driven behavior, such as notifications and updates

What Problems are Solved?

- Objects become dependent on each other - change in one object forces change in others
- Program needs to manually inform objects when state changes occur
- Adding or removing objects dynamically becomes difficult

Real Life Example – YouTube Subscriptions

- **Subject:** YouTube Channel
Observers: Subscribers
- All subscribers get notified when the channel uploads a new video
- **Naive solutions:**
 1. Constant polling of channel to check for updates
 2. Hard code every subscriber – requires knowledge of data type

Traditional Approach

```
class Subscriber {
public:
    void notifyMe(string msg) {
        cout<<"Subscriber received: "<<msg<<endl;
    }
};

class Channel {
    Subscriber s1, s2;
public:
    void uploadVideo() {
        s1.notifyMe("New video uploaded!");
        s2.notifyMe("New video uploaded!");
    }
};

int main() {
    Channel youtube;
    youtube.uploadVideo();
    return 0;
}
```

Efficient Approach

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

class Subscriber : public Observer {
public:
    void update(string msg) {
        cout<<"Subscriber received: "<<msg<<endl;
    }
};

class Channel {
    vector<Observer*> subs;
public:
    void subscribe(Observer* o) {subs.push_back(o);}
    void uploadVideo() {
        for(Observer* o: subs) o->update("New video uploaded!");
    }
};

int main() {
    Channel youtube;
    Subscriber s1, s2;
    youtube.subscribe(&s1);
    youtube.subscribe(&s2);
    youtube.uploadVideo();
    return 0;
}
```

When to use Observer Pattern?

- When multiple other objects depend on one object's state
- When implementing event-handling systems
- When observers need automatic updates

BRIDGE

Structural Design Pattern - Why?

- Focuses on how classes and objects are connected and how responsibilities are distributed across structures
- Bridge splits a class into two independent hierarchies: Abstraction and Implementation - They are connected by composition, not inheritance. This is a structural decision.
- Controls complexity caused by multiple dimensions of variation

What Problems are Solved?

- Using inheritance, the number of classes grows multiplicatively. The bridge breaks due to this explosion.
- Bridge introduces loose coupling, which does not break abstraction if there are any changes in implementations
- Allows both abstraction and implementation to evolve independently, which was dependent before

Real World Example - Message System

● Message Dimensions

- Types: Text, Email, Voice
- Senders: SMS, EmailServer, WhatsApp

● Naive Approach:

- TextViaSms
- TextViaWhatsApp
- VoiceViaWhatsApp
- EmailViaSMTP

*This would cause
inheritance explosion*

● Bridge separates:

- What the message is → Abstraction
- How it is sent → Implementation

Traditional Approach

```
class Message {  
public:  
    virtual void send(const string& msg) = 0;  
    virtual ~Message() {}  
};
```

```
class TextViaSMS : public Message {  
public:  
    void send(const string& msg) override {  
        cout << "Sending SMS Text: " << msg << endl;  
    }  
};
```

```
class TextViaEmail : public Message {  
public:  
    void send(const string& msg) override {  
        cout << "Sending Email Text: " << msg << endl;  
    }  
};
```

```
class EmailViaSMS : public Message {  
public:  
    void send(const string& msg) override {  
        cout << "Sending SMS Email: " << msg << endl;  
    }  
};
```

```
class EmailViaEmail : public Message {  
public:  
    void send(const string& msg) override {  
        cout << "Sending Email Email: " << msg << endl;  
    }  
};
```

- One base Class

- Concrete Classes for every combination causing inheritance explosion

Efficient Approach

Implementation Hierarchy:

```
#include <iostream>
using namespace std;

class MessageSender {
public:
    virtual void sendMessage(const string& msg) = 0;
    virtual ~MessageSender() {}
};
```

Concrete Implementation:

```
class SMSSender : public MessageSender {
public:
    void sendMessage(const string& msg) override {
        cout << "Sending SMS: " << msg << endl;
    }
};

class EmailSender : public MessageSender {
public:
    void sendMessage(const string& msg) override {
        cout << "Sending Email: " << msg << endl;
    }
};
```

- Decouples sending mechanisms from message type - new senders allowed without touching message classes

- Avoids class explosion like: TextViaSMS, TextViaEmail etc

- We can add:
 - WhatsAppSender
 - TelegramSenderwithout touching Message, TextMessage or EmailMessage

Abstraction:

```
class Message {
protected:
    MessageSender* sender;

public:
    Message(MessageSender* s) : sender(s) {}
    virtual void send(const string& msg) = 0;
};
```

Refined Abstraction:

```
class TextMessage : public Message {
public:
    TextMessage(MessageSender* s) : Message(s) {}

    void send(const string& msg) override {
        sender->sendMessage("Text: " + msg);
    }
};

class EmailMessage : public Message {
public:
    EmailMessage(MessageSender* s) : Message(s) {}

    void send(const string& msg) override {
        sender->sendMessage("Email: " + msg);
    }
};
```

```
MessageSender* sms = new SMSSender();
Message* msg = new TextMessage(sms);
msg->send("Hello!");
```

- MessageSender* sender; – this line is the reason this is bridge

- This means Message has a MessageSender

- This switches from inheritance → Composition

- This defines “What kind of message is this? – do not care whether it’s via SMS or WhatsApp.

- That Responsibility is bridged



THANK YOU

