

Observer Design Pattern - Comprehensive Guide

Table of Contents

1. Introduction
2. Pattern Structure and Components
3. UML Diagram
4. Real-world Examples
5. Implementation Details
6. Complete Code Example
7. Advanced Examples
8. Design Principles
9. Advantages and Disadvantages
10. When to Use and When to Avoid
11. Common Pitfalls
12. Related Patterns

Introduction

The Observer Design Pattern is a behavioral design pattern that defines a one-to-many dependency between objects. When one object (the subject) changes state, all its dependents (observers) are notified and updated automatically. This pattern promotes loose coupling between the subject and its observers.

Key Concept

The Observer pattern establishes a subscription mechanism that allows multiple objects to listen and react to events or state changes happening in another object. It implements a publish-subscribe model where observers register their interest in a subject.

Real-life Analogy

Think of a newspaper subscription service:

- **Publisher (Subject):** The newspaper company
- **Subscribers (Observers):** People who subscribe to the newspaper
- **Notification:** When a new edition is published, all subscribers are automatically notified
- **Dynamic Subscription:** People can subscribe or unsubscribe at any time
- **Independence:** Subscribers don't need to know about each other

Problem It Solves

Without the Observer pattern, objects would need to constantly check (poll) other objects for changes, leading to:

- Tight coupling between objects
- Inefficient polling mechanisms
- Difficulty in maintaining relationships
- Hard-coded dependencies

Example Problem: Imagine a stock price monitoring system where multiple displays (chart, table, alert system) need to update when stock prices change. Without Observer pattern, each display would need to constantly check for price updates, or the stock price class would need to know about all displays explicitly.

Pattern Structure and Components

The Observer pattern consists of four main components:

1. Subject Interface

- Defines methods for attaching, detaching, and notifying observers
- Maintains a list of observers
- Provides the contract for observable objects

2. Concrete Subject

- Implements the Subject interface
- Stores state that's of interest to observers
- Sends notifications to observers when state changes
- Maintains the list of observers

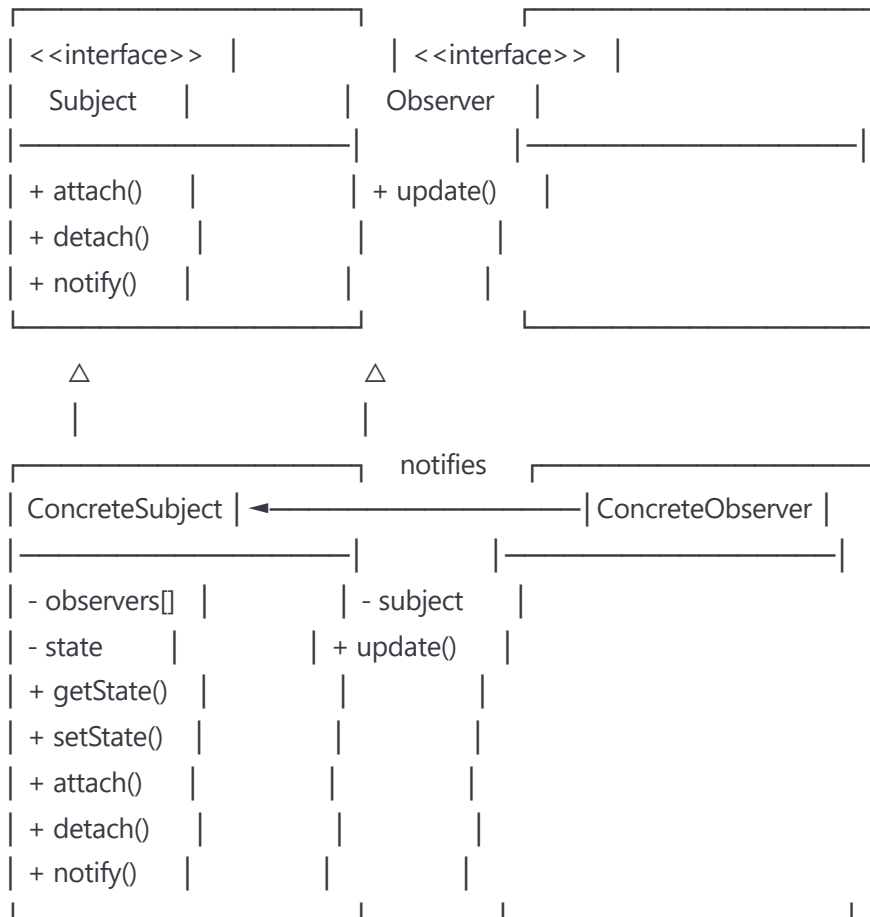
3. Observer Interface

- Defines the update method that subjects use to notify observers
- Establishes the contract for objects that want to be notified

4. Concrete Observers

- Implement the Observer interface
- Define specific update behavior when notified
- Maintain reference to subject for accessing state
- Can register/unregister themselves with subjects

UML Diagram



Real-world Examples

1. Model-View-Controller (MVC)

- **Model (Subject):** Data and business logic
- **Views (Observers):** UI components that display data
- **When model changes:** All views are automatically updated

2. Event Handling Systems

- **Button (Subject):** GUI button component
- **Event Listeners (Observers):** Functions that respond to clicks
- **When clicked:** All registered event listeners are triggered

3. Social Media Notifications

- **User Profile (Subject):** Person posting updates
- **Followers (Observers):** People following the user
- **When post is made:** All followers receive notifications

4. Stock Market Systems

- **Stock (Subject):** Individual stock price

- **Trading Apps (Observers):** Various applications monitoring the stock
- **When price changes:** All monitoring applications are updated

5. Weather Monitoring

- **Weather Station (Subject):** Collects weather data
- **Display Devices (Observers):** Phone apps, digital signs, websites
- **When data updates:** All devices show current conditions

Implementation Details

Design Considerations

1. Push vs Pull Model

- **Push:** Subject sends detailed information to observers
- **Pull:** Subject sends minimal notification; observers request details

2. Subject Independence

- Subjects shouldn't know specific observer types
- Use abstractions (interfaces) for loose coupling

3. Observer Management

- Provide methods to add/remove observers
- Handle observer lifecycle properly

4. Notification Timing

- Decide when to send notifications (immediately vs batched)
- Consider performance implications

Memory and Performance

- Each observer adds to the subject's memory footprint
- Notification time increases with number of observers
- Consider weak references to prevent memory leaks
- Batch notifications for efficiency when possible

Complete Code Example

News Agency Implementation

```
java
```

// 1. Observer Interface

```
public interface NewsObserver {  
    void update(String news, String category);  
    String getObserverName();  
}
```

// 2. Subject Interface

```
public interface NewsSubject {  
    void registerObserver(NewsObserver observer);  
    void removeObserver(NewsObserver observer);  
    void notifyObservers();  
    void setNews(String news, String category);  
}
```

// 3. Concrete Subject - News Agency

```
import java.util.*;  
import java.util.concurrent.CopyOnWriteArrayList;  
  
public class NewsAgency implements NewsSubject {  
    private List<NewsObserver> observers;  
    private String latestNews;  
    private String category;  
    private Map<String, Integer> categorySubscribers;  
  
    public NewsAgency() {  
        this.observers = new CopyOnWriteArrayList<>(); // Thread-safe  
        this.categorySubscribers = new HashMap<>();  
    }  
  
    @Override  
    public void registerObserver(NewsObserver observer) {  
        if (observer != null && !observers.contains(observer)) {  
            observers.add(observer);  
            System.out.println("Observer registered: " + observer.getObserverName());  
        }  
    }  
  
    @Override  
    public void removeObserver(NewsObserver observer) {  
        if (observers.remove(observer)) {  
            System.out.println("Observer removed: " + observer.getObserverName());  
        }  
    }  
  
    @Override  
    public void notifyObservers() {
```

```

System.out.println("\n=== NOTIFYING OBSERVERS ===");
System.out.println("News: " + latestNews);
System.out.println("Category: " + category);
System.out.println("Notifying " + observers.size() + " observers...\n");

for (NewsObserver observer : observers) {
    try {
        observer.update(latestNews, category);
    } catch (Exception e) {
        System.err.println("Error notifying observer " +
            observer.getObserverName() + ": " + e.getMessage());
    }
}

updateCategoryStats();
}

@Override
public void setNews(String news, String category) {
    this.latestNews = news;
    this.category = category;
    notifyObservers();
}

private void updateCategoryStats() {
    categorySubscribers.put(category,
        categorySubscribers.getOrDefault(category, 0) + 1);
}

public int getObserverCount() {
    return observers.size();
}

public String getLatestNews() {
    return latestNews;
}

public String getCategory() {
    return category;
}

public Map<String, Integer> getCategoryStats() {
    return new HashMap<>(categorySubscribers);
}
}

```

```

public class NewsChannelObserver implements NewsObserver {
    private String channelName;
    private String currentNews;
    private List<String> newsHistory;
    private Set<String> interestedCategories;

    public NewsChannelObserver(String channelName) {
        this.channelName = channelName;
        this.newsHistory = new ArrayList<>();
        this.interestedCategories = new HashSet<>();
        // Default interested categories
        this.interestedCategories.addAll(Arrays.asList("BREAKING", "POLITICS", "SPORTS"));
    }

    public NewsChannelObserver(String channelName, Set<String> categories) {
        this.channelName = channelName;
        this.newsHistory = new ArrayList<>();
        this.interestedCategories = new HashSet<>(categories);
    }

    @Override
    public void update(String news, String category) {
        if (interestedCategories.contains(category)) {
            this.currentNews = news;
            this.newsHistory.add(String.format("[%s] %s", category, news));
            System.out.println("📺 " + channelName + " broadcasting: " + news);
        } else {
            System.out.println("📺 " + channelName + " (ignoring " + category + " news)");
        }
    }

    @Override
    public String getObserverName() {
        return channelName;
    }

    public void addInterestedCategory(String category) {
        interestedCategories.add(category);
    }

    public void removeInterestedCategory(String category) {
        interestedCategories.remove(category);
    }

    public List<String> getNewsHistory() {
        return new ArrayList<>(newsHistory);
    }
}

```

```

}

public class OnlinePortalObserver implements NewsObserver {
    private String portalName;
    private String websiteUrl;
    private int articleCount;
    private Map<String, List<String>> categorizedNews;

    public OnlinePortalObserver(String portalName, String websiteUrl) {
        this.portalName = portalName;
        this.websiteUrl = websiteUrl;
        this.articleCount = 0;
        this.categorizedNews = new HashMap<>();
    }

    @Override
    public void update(String news, String category) {
        this.articleCount++;
        categorizedNews.computeIfAbsent(category, k -> new ArrayList<>()).add(news);

        System.out.println("🌐 " + portalName + " (" + websiteUrl + ") published: " + news);
        System.out.println("  Article #" + articleCount + " in " + category + " section");
    }

    @Override
    public String getObserverName() {
        return portalName + " Portal";
    }

    public int getArticleCount() {
        return articleCount;
    }

    public Map<String, List<String>> getCategorizedNews() {
        return new HashMap<>(categorizedNews);
    }
}

```

```

public class SocialMediaObserver implements NewsObserver {
    private String platformName;
    private int followerCount;
    private List<String> posts;
    private int engagementScore;

    public SocialMediaObserver(String platformName, int followerCount) {
        this.platformName = platformName;
        this.followerCount = followerCount;
    }
}

```



```

    this.posts = new ArrayList<>();
    this.engagementScore = 0;
}

@Override
public void update(String news, String category) {
    String post = createSocialPost(news, category);
    posts.add(post);

    // Simulate engagement based on category
    int engagement = calculateEngagement(category);
    engagementScore += engagement;

    System.out.println("📰 " + platformName + " posted: " + post);
    System.out.println("   Reached " + followerCount + " followers, +" +
        engagement + " engagement");
}

@Override
public String getObserverName() {
    return platformName + " Social";
}

private String createSocialPost(String news, String category) {
    String emoji = getEmojiForCategory(category);
    return emoji + " " + news + " #" + category.toLowerCase() + " #news";
}

private String getEmojiForCategory(String category) {
    switch (category) {
        case "BREAKING": return "🚨";
        case "SPORTS": return "⚽";
        case "POLITICS": return "🏛️";
        case "TECHNOLOGY": return "💻";
        case "WEATHER": return "☀️";
        default: return "📰";
    }
}

private int calculateEngagement(String category) {
    int baseEngagement = followerCount / 100;
    switch (category) {
        case "BREAKING": return baseEngagement * 3;
        case "SPORTS": return baseEngagement * 2;
        case "POLITICS": return (int)(baseEngagement * 1.5);
        default: return baseEngagement;
    }
}

```

```

    }

    public int getTotalEngagement() {
        return engagementScore;
    }

    public List<String> getPosts() {
        return new ArrayList<>(posts);
    }
}

// 5. Specialized Observer for Push Notifications
public class PushNotificationObserver implements NewsObserver {
    private String appName;
    private boolean criticalOnly;
    private int notificationsSent;
    private Map<String, Integer> categoryNotifications;

    public PushNotificationObserver(String appName, boolean criticalOnly) {
        this.appName = appName;
        this.criticalOnly = criticalOnly;
        this.notificationsSent = 0;
        this.categoryNotifications = new HashMap<>();
    }

    @Override
    public void update(String news, String category) {
        if (shouldSendNotification(category)) {
            sendPushNotification(news, category);
            notificationsSent++;
            categoryNotifications.put(category,
                categoryNotifications.getOrDefault(category, 0) + 1);
        } else {
            System.out.println("🚫 NewsApp Push Notifications (notification filtered for SPORTS)");
        }
    }
}

```

Observer removed: ESPN

=== NOTIFYING OBSERVERS ===

News: New AI breakthrough announced

Category: TECHNOLOGY

Notifying 2 observers...

🌐 TechCrunch (www.techcrunch.com) published: New AI breakthrough announced

Article #2 in TECHNOLOGY section

🔔 NewsApp PUSH [NORMAL]: New AI breakthrough announced

=====

Advanced Examples

Thread-Safe Observer Implementation

```
java
```

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadSafeNewsAgency implements NewsSubject {
    private final CopyOnWriteArrayList<NewsObserver> observers;
    private volatile String latestNews;
    private volatile String category;
    private final ExecutorService notificationExecutor;

    public ThreadSafeNewsAgency() {
        this.observers = new CopyOnWriteArrayList<>();
        this.notificationExecutor = Executors.newCachedThreadPool();
    }

    @Override
    public void registerObserver(NewsObserver observer) {
        observers.addIfAbsent(observer);
    }

    @Override
    public void removeObserver(NewsObserver observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        String currentNews = this.latestNews;
        String currentCategory = this.category;

        for (NewsObserver observer : observers) {
            // Asynchronous notification
            notificationExecutor.submit(() -> {
                try {
                    observer.update(currentNews, currentCategory);
                } catch (Exception e) {
                    System.err.println("Error notifying observer: " + e.getMessage());
                }
            });
        }
    }

    @Override
    public void setNews(String news, String category) {
        this.latestNews = news;
    }
}
```

```
this.category = category;
notifyObservers();
}

public void shutdown() {
    notificationExecutor.shutdown();
}
}
```

Observer with Filtering and Priorities

```
java
```

```

public interface PriorityObserver extends NewsObserver {
    int getPriority(); // Higher number = higher priority
    boolean isInterestedIn(String category);
}

public class FilteringNewsAgency implements NewsSubject {
    private List<PriorityObserver> priorityObservers;
    private String latestNews;
    private String category;

    public FilteringNewsAgency() {
        this.priorityObservers = new ArrayList<>();
    }

    @Override
    public void registerObserver(NewsObserver observer) {
        if (observer instanceof PriorityObserver) {
            PriorityObserver priorityObs = (PriorityObserver) observer;
            // Insert based on priority (higher priority first)
            int insertIndex = 0;
            for (int i = 0; i < priorityObservers.size(); i++) {
                if (priorityObservers.get(i).getPriority() < priorityObs.getPriority()) {
                    insertIndex = i;
                    break;
                }
            }
            insertIndex = i + 1;
        }
        priorityObservers.add(insertIndex, priorityObs);
    }

    @Override
    public void removeObserver(NewsObserver observer) {
        priorityObservers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        System.out.println("\n=== PRIORITY-BASED NOTIFICATION ===");
        System.out.println("Category: " + category);

        for (PriorityObserver observer : priorityObservers) {
            if (observer.isInterestedIn(category)) {
                System.out.println("Priority " + observer.getPriority() + ": ");
                observer.update(latestNews, category);
            }
        }
    }
}

```

```
    }  
}  
  
@Override  
public void setNews(String news, String category) {  
    this.latestNews = news;  
    this.category = category;  
    notifyObservers();  
}  
}
```

Event-Driven Observer with Custom Events

```
java
```

```
public class NewsEvent {
    private String news;
    private String category;
    private Date timestamp;
    private String source;
    private Map<String, Object> metadata;

    public NewsEvent(String news, String category, String source) {
        this.news = news;
        this.category = category;
        this.source = source;
        this.timestamp = new Date();
        this.metadata = new HashMap<>();
    }

    // Getters and setters
    public String getNews() { return news; }
    public String getCategory() { return category; }
    public Date getTimestamp() { return timestamp; }
    public String getSource() { return source; }
    public Map<String, Object> getMetadata() { return metadata; }

    public void addMetadata(String key, Object value) {
        metadata.put(key, value);
    }
}

public interface EventObserver {
    void onNewsEvent(NewsEvent event);
    String getObserverName();
}

public class EventDrivenNewsAgency {
    private List<EventObserver> observers;

    public EventDrivenNewsAgency() {
        this.observers = new ArrayList<>();
    }

    public void subscribe(EventObserver observer) {
        observers.add(observer);
    }

    public void unsubscribe(EventObserver observer) {
        observers.remove(observer);
    }
}
```



```
public void publishNews(String news, String category, String source) {  
    NewsEvent event = new NewsEvent(news, category, source);  
    event.addMetadata("publishTime", new Date());  
    event.addMetadata("observerCount", observers.size());  
  
    notifyObservers(event);  
}  
  
private void notifyObservers(NewsEvent event) {  
    for (EventObserver observer : observers) {  
        observer.onNewsEvent(event);  
    }  
}  
}
```

Design Principles

The Observer pattern adheres to several key design principles:

1. Loose Coupling

- Subjects and observers interact through interfaces
- Subjects don't need to know concrete observer types
- Observers can be added/removed independently

2. Open/Closed Principle (OCP)

- Open for extension: New observer types can be added
- Closed for modification: Existing subject code doesn't change

3. Single Responsibility Principle (SRP)

- Subjects manage their state and observer list
- Observers handle their specific update logic
- Clear separation of concerns

4. Dependency Inversion Principle

- Both subjects and observers depend on abstractions (interfaces)
- High-level modules don't depend on low-level modules

5. Publish-Subscribe Model

- Implements a clean publisher-subscriber relationship
- Supports one-to-many communication efficiently

Advantages and Disadvantages

✅ " + appName + " (notification filtered for " + category + ")");

```
    }  
}  
  
@Override  
public String getObserverName() {  
    return appName + " Push Notifications";  
}  
  
private boolean shouldSendNotification(String category) {  
    if (criticalOnly) {  
        return category.equals("BREAKING") || category.equals("EMERGENCY");  
    }  
    return true;  
}  
  
private void sendPushNotification(String news, String category) {  
    String priority = category.equals("BREAKING") ? "HIGH" : "NORMAL";  
    System.out.println("🔔 " + appName + " PUSH [" + priority + "]: " +  
        truncateNews(news, 50));  
}  
  
private String truncateNews(String news, int maxLength) {  
    return news.length() > maxLength ?  
        news.substring(0, maxLength) + "...": news;  
}  
  
public int getNotificationsSent() {  
    return notificationsSent;  
}  
  
public Map<String, Integer> getCategoryNotifications() {  
    return new HashMap<>(categoryNotifications);  
}  
}
```

// 6. Observable with Event Types (Advanced) public class AdvancedNewsAgency implements
NewsSubject { private Map<String, List<NewsObserver>> categoryObservers; private String latestNews;
private String category; private Date timestamp;

```

public AdvancedNewsAgency() {
    this.categoryObservers = new HashMap<>();
    this.timestamp = new Date();
}

public void registerObserverForCategory(NewsObserver observer, String category) {
    categoryObservers.computeIfAbsent(category, k -> new ArrayList<>()).add(observer);
    System.out.println("Observer " + observer.getObserverName() +
        " registered for " + category + " news");
}

@Override
public void registerObserver(NewsObserver observer) {
    // Register for all categories
    for (String category : Arrays.asList("BREAKING", "SPORTS", "POLITICS",
        "TECHNOLOGY", "WEATHER")) {
        registerObserverForCategory(observer, category);
    }
}

@Override
public void removeObserver(NewsObserver observer) {
    for (List<NewsObserver> observers : categoryObservers.values()) {
        observers.remove(observer);
    }
    System.out.println("Observer removed from all categories: " +
        observer.getObserverName());
}

@Override
public void notifyObservers() {
    List<NewsObserver> relevantObservers =
        categoryObservers.getDefault(category, new ArrayList<>());

    System.out.println("\n=== CATEGORY-SPECIFIC NOTIFICATION ===");
    System.out.println("Category: " + category);
    System.out.println("News: " + latestNews);
    System.out.println("Notifying " + relevantObservers.size() +
        " observers for " + category + "...\\n");

    for (NewsObserver observer : relevantObservers) {
        observer.update(latestNews, category);
    }
}

@Override

```

```
public void setNews(String news, String category) {  
    this.latestNews = news;  
    this.category = category;  
    this.timestamp = new Date();  
    notifyObservers();  
}  
  
public void removeCategory(String category) {  
    categoryObservers.remove(category);  
}  
  
public Set<String> getAvailableCategories() {  
    return new HashSet<>(categoryObservers.keySet());  
}  
}
```

Demonstration Class

```
```java
import java.util.*;

public class NewsAgencyDemo {
 public static void main(String[] args) {
 System.out.println("=== NEWS AGENCY OBSERVER PATTERN DEMO ===\n");

 // 1. Basic Observer pattern
 demonstrateBasicPattern();

 // 2. Dynamic observer management
 demonstrateDynamicManagement();

 // 3. Category-specific notifications
 demonstrateCategorySpecificNotifications();

 // 4. Observer statistics
 demonstrateObserverStatistics();
 }

 private static void demonstrateBasicPattern() {
 System.out.println("1. BASIC OBSERVER PATTERN:");
 System.out.println("-".repeat(40));

 NewsAgency agency = new NewsAgency();

 // Create observers
 NewsObserver cnn = new NewsChannelObserver("CNN");
 NewsObserver bbc = new NewsChannelObserver("BBC");
 NewsObserver reddit = new SocialMediaObserver("Reddit", 50000);

 // Register observers
 agency.registerObserver(cnn);
 agency.registerObserver(bbc);
 agency.registerObserver(reddit);

 // Publish news
 agency.setNews("Major earthquake hits California", "BREAKING");

 System.out.println("\n" + "=".repeat(50) + "\n");
 }

 private static void demonstrateDynamicManagement() {
```

```

System.out.println("2. DYNAMIC OBSERVER MANAGEMENT:");
System.out.println("-".repeat(45));

NewsAgency agency = new NewsAgency();

NewsObserver espn = new NewsChannelObserver("ESPN",
 Set.of("SPORTS"));
NewsObserver techCrunch = new OnlinePortalObserver("TechCrunch",
 "www.techcrunch.com");
NewsObserver pushApp = new PushNotificationObserver("NewsApp", true);

agency.registerObserver(espn);
agency.registerObserver(techCrunch);
agency.registerObserver(pushApp);

// Sports news
agency.setNews("Team wins championship", "SPORTS");

// Remove ESPN and add technology news
agency.removeObserver(espn);
agency.setNews("New AI breakthrough announced", "TECHNOLOGY");

System.out.println("\n" + "=".repeat(50) + "\n");
}

private static void demonstrateCategorySpecificNotifications() {
 System.out.println("3. CATEGORY-SPECIFIC NOTIFICATIONS:");
 System.out.println("-".repeat(45));

 AdvancedNewsAgency agency = new AdvancedNewsAgency();

 NewsObserver sportsObserver = new NewsChannelObserver("ESPN");
 NewsObserver techObserver = new OnlinePortalObserver("TechNews",
 "www.technews.com");
 NewsObserver generalObserver = new SocialMediaObserver("Twitter", 100000);

 // Register for specific categories
 agency.registerObserverForCategory(sportsObserver, "SPORTS");
 agency.registerObserverForCategory(techObserver, "TECHNOLOGY");
 agency.registerObserver(generalObserver); // All categories

 // Publish different types of news
 agency.setNews("New smartphone released", "TECHNOLOGY");
 agency.setNews("Football match results", "SPORTS");
 agency.setNews("Stock market update", "BUSINESS");

 System.out.println("\n" + "=".repeat(50) + "\n");
}

```

```

}

private static void demonstrateObserverStatistics() {
 System.out.println("4. OBSERVER STATISTICS:");
 System.out.println("-".repeat(30));

 NewsAgency agency = new NewsAgency();

 SocialMediaObserver twitter = new SocialMediaObserver("Twitter", 75000);
 SocialMediaObserver facebook = new SocialMediaObserver("Facebook", 120000);
 PushNotificationObserver pushApp = new PushNotificationObserver("NewsApp", false);

 agency.registerObserver(twitter);
 agency.registerObserver(facebook);
 agency.registerObserver(pushApp);

 // Publish various news
 agency.setNews("Breaking: Major policy announcement", "BREAKING");
 agency.setNews("Sports update: Championship finals", "SPORTS");
 agency.setNews("Tech news: New software release", "TECHNOLOGY");

 // Print statistics
 System.out.println("\n📊 STATISTICS:");
 System.out.println("Twitter engagement: " + twitter.getTotalEngagement());
 System.out.println("Facebook engagement: " + facebook.getTotalEngagement());
 System.out.println("Push notifications sent: " + pushApp.getNotificationsSent());
 System.out.println("Total observers: " + agency.getObserverCount());
}
}

```

## Expected Output:

=== NEWS AGENCY OBSERVER PATTERN DEMO ===

## 1. BASIC OBSERVER PATTERN:

-----  
Observer registered: CNN

Observer registered: BBC

Observer registered: Reddit Social

=== NOTIFYING OBSERVERS ===



News: Major earthquake hits California

Category: BREAKING

Notifying 3 observers...

 CNN broadcasting: Major earthquake hits California

 BBC broadcasting: Major earthquake hits California

 Reddit posted:  Major earthquake hits California #breaking #news  
Reached 50000 followers, +1500 engagement

=====

## 2. DYNAMIC OBSERVER MANAGEMENT:

-----  
Observer registered: ESPN

Observer registered: TechCrunch Portal


Observer registered: NewsApp Push Notifications


=== NOTIFYING OBSERVERS ===

News: Team wins championship

Category: SPORTS

Notifying 3 observers...

 ESPN broadcasting: Team wins championship

 TechCrunch (www.techcrunch.com) published: Team wins championship  
Article #1 in SPORTS section

