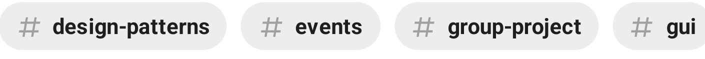
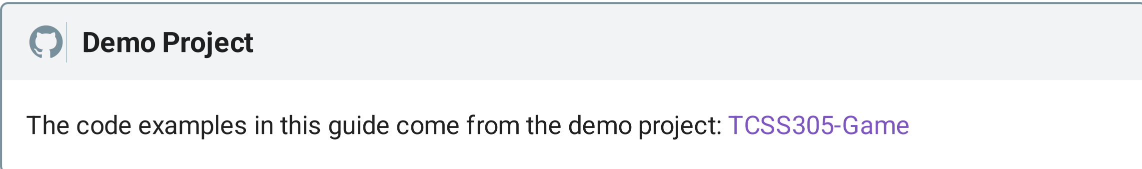


The Observer Pattern

TCSS 305 Programming Practicum



The Observer pattern is one of the most widely used design patterns in software development. It defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically. This guide covers the pattern's history, its implementation using Java's `PropertyChangeListener` framework, and how modern Java features like sealed classes and records create type-safe event systems.

---

1 Brief History

The Observer pattern was first catalogued by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) in their seminal 1994 book *Design Patterns: Elements of Reusable Object-Oriented Software*. However, the concept of observers watching for state changes predates this formalization--it's rooted in event-driven programming models from the 1980s and earlier GUI frameworks.

The pattern is also known as **Publish-Subscribe** or the **Dependents** pattern. Java has provided several implementations of this pattern over the years:

1.1 Java's Observer Implementations

`java.util.Observable` and `java.util.Observer` (JDK 1.0 - Deprecated in Java 9): Java's first built-in observer framework required subjects to extend the `Observable` class. This was problematic because it forced inheritance (preventing you from extending other classes), wasn't thread-safe, and provided limited control over what data was sent to observers. The API was deprecated in Java 9 due to these design flaws.

`java.beans.PropertyChangeListener` (JDK 1.1 - Still widely used): This framework was designed specifically for JavaBeans and Swing GUI components. Instead of requiring inheritance, it uses a delegation model with `PropertyChangeSupport` to manage observers. This is the approach our codebase uses because it's flexible, thread-safe, and purpose-built for notifying about property changes in components.

`java.util.concurrent.Flow` (Java 9+): The newest observer implementation follows the Reactive Streams specification, providing backpressure support for asynchronous data streams. While powerful for complex reactive applications, it's significantly more complicated than `PropertyChangeListener` and isn't tailored for Swing GUI applications. For our use case--simple property change notifications in a Swing application-- `PropertyChangeListener` is the better choice.

---

2 The Problem It Solves

Imagine you're building a weather station that collects temperature data. Multiple devices need to react to temperature changes: a display screen should update the current temperature, an alert system should warn if it gets too hot or cold, and a logging system should record the data.

The naive approach would have the weather station directly call methods on each device whenever the temperature changes. But this creates tight coupling--the weather station needs to know about every device that cares about temperature. If you want to add a new device, you have to modify the weather station code. If a device is removed, you risk errors. The weather station becomes a tangled mess of dependencies.

You need a way for the weather station to announce "the temperature changed" without knowing or caring who's listening. Devices should be able to subscribe to updates when they're interested, and unsubscribe when they're not--all without the weather station having to track them explicitly.

---

3 How the Pattern Solves It

The Observer pattern introduces two key roles:

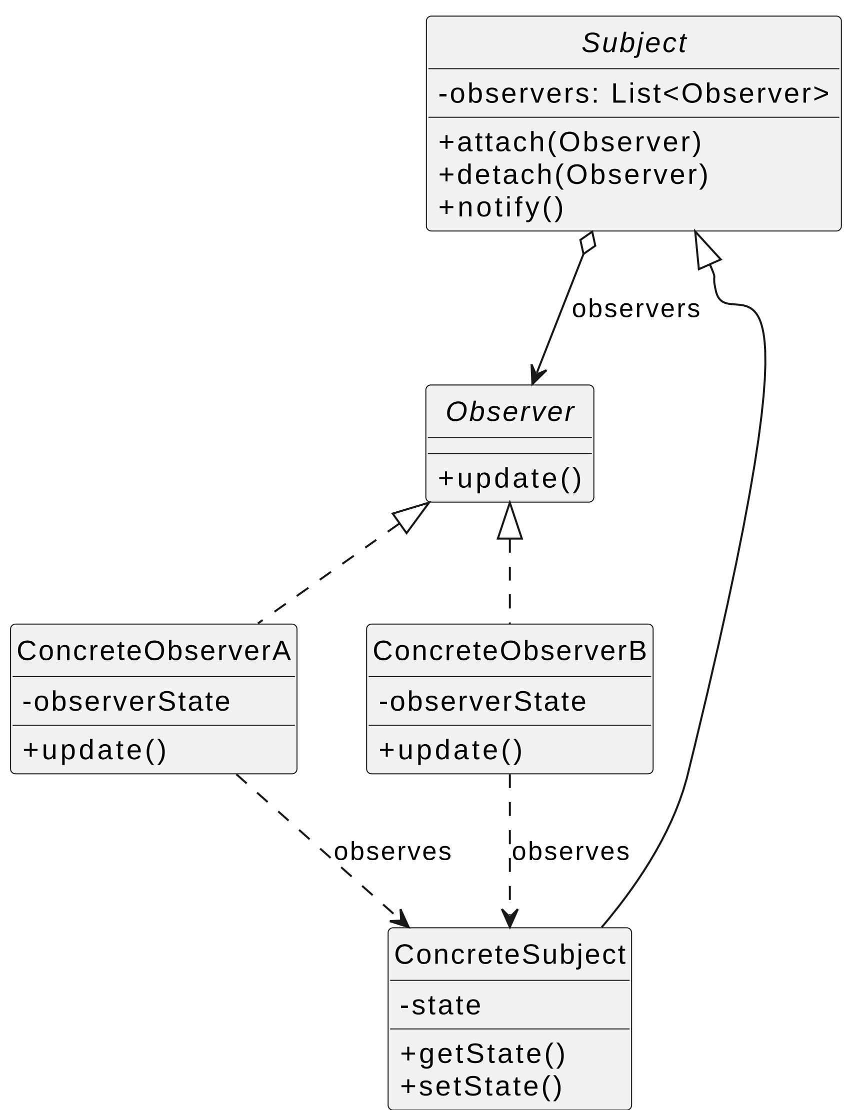
Subject (Observable): The object that holds state and notifies others when it changes. The subject maintains a list of interested observers and provides methods to register/unregister them. When state changes, the subject loops through its observers and notifies each one.

Observer: Objects that want to be notified of changes. Each observer implements a standard interface with an update/notification method. When the subject calls this method, the observer receives information about what changed and can react accordingly.

This decouples the subject from the observers. The subject only knows "I have a list of objects that implement the Observer interface." Observers can come and go dynamically. The subject announces changes without knowing what observers will do with that information--and that's exactly how it should be.

The key insight: **push notifications instead of continuous polling**. Rather than each device constantly checking "has the temperature changed yet?", the weather station pushes updates only when changes occur.

3.1 UML Class Diagram - Generic Observer Pattern



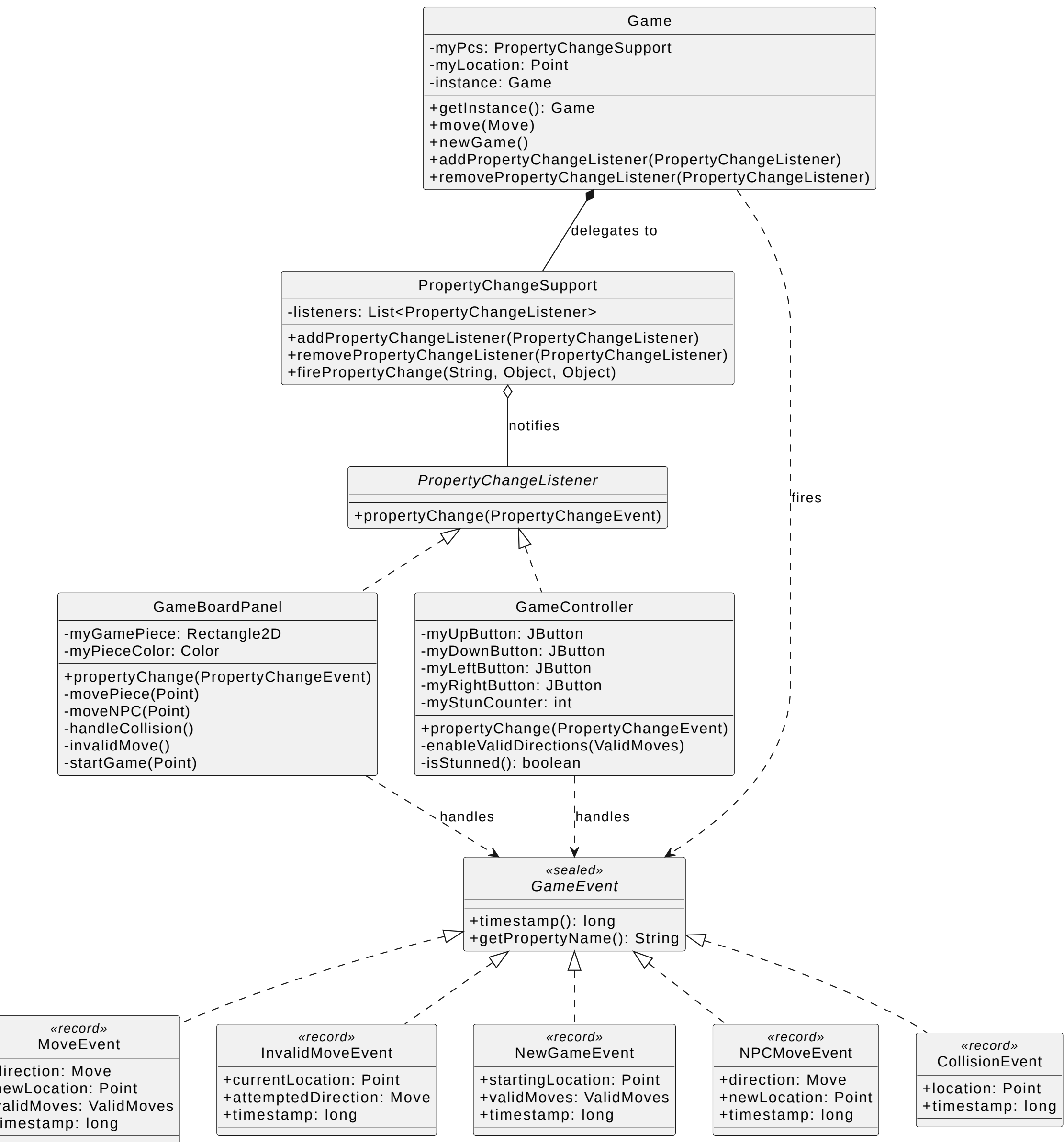
In this classic structure, the `Subject` maintains a collection of `Observer` instances. When state changes in a `ConcreteSubject`, it calls `notify()`, which loops through all registered observers and calls their `update()` method. Each `ConcreteObserver` can then query the subject for its new state or receive the state as a parameter.

---

4 Technical Implementation in Our Codebase

This project demonstrates a modern take on the Observer pattern using Java's `PropertyChangeListener` framework enhanced with type-safe sealed events (a Java 21 feature).

4.1 UML Class Diagram - Our Implementation



This diagram shows how our implementation uses delegation (`Game` delegates observer management to `PropertyChangeSupport`) and sealed event types (`GameEvent` with five record implementations) to create a type-safe, modern Observer pattern.

4.2 The Subject: Game Class

The `Game` class is our subject--it holds the game state (piece location) and notifies observers when that state changes. See `Game.java` for the full implementation.

Using PropertyChangeSupport

Instead of implementing our own observer registration and notification system (maintaining a list of observers, iterating through them, etc.), we delegate this responsibility to Java's built-

in `PropertyChangeSupport` class. This helper class is part of the JavaBeans framework and does all the heavy lifting for us:

- Maintains the list of registered observers
- Provides thread-safe add/remove methods
- Handles the notification loop when we fire events
- Allows filtering by property name (though we don't use this feature)

Here's how we set it up in the `Game` constructor (lines 63-66):

```
public final class Game implements PropertyChangeEnabledGameControls {
    private final PropertyChangeSupport myPcs;
    private Point myLocation;

    private Game() {
        myPcs = new PropertyChangeSupport(this);
        myLocation = STARTING_LOCATION;
        // ...
    }
}
```

The `PropertyChangeSupport(this)` constructor takes a reference to the subject object (the `Game` instance). This reference is included in property change events so observers can identify which object changed.

Notifying Observers

When the game state changes, we create a typed event and ask `PropertyChangeSupport` to notify all registered observers. Here's the `move()` method (lines 111-129):

```
@Override
public void move(final Move theMove) {
    if (myMoveValidators.get(theMove).getAsBoolean()) {
        myLocation = myMovements.get(theMove).get();
        final GameEvent event = new GameEvent.MoveEvent(
            theMove,
            myLocation,
            calculateValidDirections(),
            GameEvent.now()
        );
        myPcs.firePropertyChange(event.getPropertyName(), null, event);
    }
}
```

The call to `myPcs.firePropertyChange()` triggers the notification process. `PropertyChangeSupport` loops through all registered observers and calls their `propertyChange()` method with information about what changed. We don't need to write this loop ourselves-- `PropertyChangeSupport` handles it.

Exposing the Observer Registration Interface

The `Game` class implements the `PropertyChangeEnabledGameControls` interface (lines 196-216), which requires methods for adding and removing observers. We simply delegate these calls to our `PropertyChangeSupport` instance:

```
@Override
public void addPropertyChangeListener(final PropertyChangeListener
theListener) {
    myPcs.addPropertyChangeListener(theListener);
}

@Override
public void removePropertyChangeListener(final PropertyChangeListener
theListener) {
    myPcs.removePropertyChangeListener(theListener);
}
```

This is the delegation pattern in action: `Game` implements the interface, but `PropertyChangeSupport` does the actual work.

4.3 The Observer Interface

Observers implement Java's standard `PropertyChangeListener` interface, which requires one method:

```
void propertyChange(PropertyChangeEvent evt);
```

Our project has two observers: `GameController` and `GameBoardPanel`, both implementing this interface.

4.4 Observer 1: GameBoardPanel (The View)

The `GameBoardPanel` listens for game state changes and updates the visual display. See `GameBoardPanel.java` (line 23 for class declaration).

```
public class GameBoardPanel extends JPanel implements PropertyChangeListener {

    @Override
    public void propertyChange(final PropertyChangeEvent theEvent) {
        if (theEvent.getNewValue() instanceof final GameEvent event) {
            switch (event) {
                case final GameEvent.MoveEvent moveEvent ->
                    movePiece(moveEvent.newLocation());
                case final GameEvent.InvalidMoveEvent invalidEvent ->
                    invalidMove();
                case final GameEvent.NewGameEvent newGameEvent ->
                    startGame(newGameEvent.startingLocation());
```

```java
                case final GameEvent.NPCMoveEvent npcMoveEvent ->
                    moveNPC(npcMoveEvent.newLocation());
                case final GameEvent.CollisionEvent collisionEvent ->
                    handleCollision();
            }
        }
    }
}
```

> **Modern Java Feature: Pattern Matching for Switch**
>
> This code uses **pattern matching for switch** (Java 21) combined with sealed types to enable exhaustive, type-safe event handling. The compiler verifies that all possible `GameEvent` subtypes are handled, eliminating the risk of missing a case. Sealed classes restrict which types can implement an interface, and pattern matching lets `switch` expressions destructure those types directly--no casting required.

4.5 Observer 2: GameController

The `GameController` listens for the same events but responds differently--it enables/disables directional buttons based on valid moves. See `GameController.java` (line 32 for class declaration).

```java
@Override
public void propertyChange(final PropertyChangeEvent theEvent) {
    if (theEvent.getNewValue() instanceof final GameEvent event) {
        switch (event) {
            case final GameEvent.MoveEvent moveEvent ->
                enableValidDirections(moveEvent.validMoves());
            case final GameEvent.NewGameEvent newGameEvent ->
                enableValidDirections(newGameEvent.validMoves());
            case final GameEvent.NPCMoveEvent e -> { }  // Handled by
GameBoardPanel
            case final GameEvent.CollisionEvent e -> {
                if (!isStunned()) {
                    myStunCounter = STUN_DURATION;
                }
            }
            default -> { }
        }
    }
}
```

4.6 Wiring It Together

Observers register themselves with the subject during setup. See `GameController.java` (lines 179-205):

```java
public static void createAndShowGUI() {
    final Game game = Game.getInstance();
    final GameController pane = new GameController(game);

    // Register the controller as an observer
    game.addPropertyChangeListener(pane);

    // Register the view as an observer
    final GameBoardPanel gamePanel = new GameBoardPanel();
    game.addPropertyChangeListener(gamePanel);
}
```

Now when the game state changes (via `move()`, `newGame()`, etc.), both observers are automatically notified. The `Game` class doesn't know what the observers will do with the notification--it just announces the change.

4.7 Type-Safe Events with Sealed Classes

This codebase modernizes the classic Observer pattern with Java 21's sealed classes. Instead of string-based property names (`"position"`, `"gameOver"`), we use sealed event types. See `GameEvent.java` (lines 1-129).

> **Modern Java Feature: Sealed Classes and Records**
>
> This implementation uses **sealed classes** (Java 17) combined with **records** (Java 16) to create type-safe event hierarchies. Sealed classes restrict which types can implement an interface, giving the compiler full knowledge of all possible subtypes. Records provide concise, immutable data carriers with automatically generated `equals()`, `hashCode()`, and `toString()` methods.

```java
public sealed interface GameEvent permits
        GameEvent.MoveEvent,
        GameEvent.InvalidMoveEvent,
        GameEvent.NewGameEvent,
        GameEvent.NPCMoveEvent,
        GameEvent.CollisionEvent {

    record MoveEvent(
        Move direction,
        Point newLocation,
        ValidMoves validMoves,
        long timestamp
    ) implements GameEvent { }

    record InvalidMoveEvent(
        Point currentLocation,
        Move attemptedDirection,
        long timestamp
    ) implements GameEvent { }
```

```
    record NewGameEvent(
        Point startingLocation,
        ValidMoves validMoves,
        long timestamp
    ) implements GameEvent { }

    record NPCMoveEvent(
        Move direction,
        Point newLocation,
        long timestamp
    ) implements GameEvent { }

    record CollisionEvent(
        Point location,
        long timestamp
    ) implements GameEvent { }
}
```

This gives us:

- **Compile-time type safety**: The compiler knows all possible event types
- **Exhaustiveness checking**: Switch statements must handle all cases
- **IDE support**: Autocomplete shows available events and their data
- **Self-documenting**: Event types clearly show what data they carry

---

5 Benefits and Tradeoffs

5.1 Benefits

- **Loose coupling**: Subject and observers are independent; you can add/remove observers without modifying the subject
- **Dynamic relationships**: Observers can register/unregister at runtime
- **Broadcast communication**: One subject can notify multiple observers simultaneously
- **Separation of concerns**: Business logic (subject) is separated from presentation/reaction logic (observers)

5.2 Tradeoffs

- **Unexpected updates**: Observers may trigger cascading updates, causing performance issues

- **Memory leaks**: Forgetting to unregister observers can prevent garbage collection
- **Order dependency**: If observers rely on a specific notification order, you have hidden coupling
- **Debugging complexity**: The flow of execution jumps between subject and observers, making stack traces harder to follow

---

6 Common Pitfalls

Forgetting to Unregister Observers

When an observer is no longer needed, you must explicitly remove it via `removePropertyChangeListener()`. Otherwise, the subject maintains a reference, preventing garbage collection. This is especially important with GUI components that get created and destroyed frequently.

Modifying State in Observer Notifications

If an observer modifies the subject's state during a notification, you can create infinite loops or unpredictable update cascades. Observers should react to state changes, not cause new ones.

Assuming Notification Order

Java's `PropertyChangeSupport` notifies observers in registration order, but relying on this creates hidden coupling. Each observer should be independent.

Sending Too Much or Too Little Data

Our sealed event records include all relevant data (`newLocation`, `validMoves`) so observers don't need to query the subject. This avoids race conditions and simplifies observer logic. The classic mistake is sending just a "something changed" notification, forcing observers to ask "what changed?"--which can get the wrong answer if state changed again.

---

7 Related Patterns

Model-View-Controller (MVC): Observer is the backbone of MVC. The Model is the subject, Views are observers. This project uses MVC with `Game` as the Model, `GameBoardPanel` as the View, and `GameController` as the Controller. See MVC Pattern.

Mediator: Both patterns decouple objects, but Mediator centralizes communication through a mediator object, while Observer uses a distributed publish-subscribe approach. Use Observer when subjects and observers have a one-to-many relationship; use Mediator when you have many-to-many communication needs.

Strategy: Our codebase combines Observer with Strategy--the `Game` class uses strategy for move validation while using observer for state notifications. See Strategy Pattern.

Summary

Concept	Key Point
Observer Pattern	Defines a one-to-many dependency so that when one object changes state, all dependents are notified automatically
Subject	The object that holds state and notifies observers when it changes (e.g., `Game`)
Observer	Objects that register with a subject to receive notifications (e.g., `GameBoardPanel`, `GameController`)
PropertyChangeSupport	Java's built-in helper class that manages observer registration and notification -- use delegation, not inheritance
PropertyChangeListener	The standard Java interface for observers, requiring a single `propertyChange()` method
Sealed Event Types	Modern Java approach using sealed interfaces and records for compile-time type safety and exhaustiveness checking
Push vs. Poll	Observers receive pushed notifications instead of continuously polling the subject for changes

Concept	Key Point
Loose Coupling	The subject knows nothing about its observers beyond the interface they implement

---

Further Reading

External Resources

- JavaBeans PropertyChangeListener Tutorial -- Official Oracle documentation on PropertyChangeListener
- JEP 409: Sealed Classes -- The Java feature that enables type-safe event hierarchies

---

References

Primary Texts:

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Chapter 5, pp. 293-303.
- Freeman, E., & Robson, E. (2020). *Head First Design Patterns* (2nd ed.). O'Reilly Media. Chapter 2: The Observer Pattern.
- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press.

Language Documentation:

- Oracle JDK 25: `PropertyChangeListener` -- Interface for receiving property change events
- Oracle JDK 25: `PropertyChangeSupport` -- Helper class for managing property change listeners
- Oracle JDK 25: `PropertyChangeEvent` -- Event object for property changes

Additional Resources:

- Alexander, C. (1977). *A Pattern Language*. Oxford University Press. -- The architectural work that inspired software design patterns.

This guide is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.