lectures # week-9

Week 9: Observer Pattern Implementation & MVC (Friday, March 6, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers concepts for Group Project Sprint 2.

Demo Code

Clone the lecture examples: TCSS305-Game

Lecture Preview 0:00

Today's agenda:

1. **Admin & Sprint 2 Q&A** — Sprint 2 questions, Sprint 3 requirements, Test 3 preview
2. **MVC architecture** — Model, View, Controller — how they map to Tetris
3. **PropertyChangeSupport & firePropertyChange** — Properties, events, and the sealed interface approach
4. **PropertyChangeListener implementation** — Switch expressions with pattern matching in view code
5. **Key listener strategies** — Map-based key bindings for flexible controls

Next Friday

Test 3. A test prep page with sample problems will be published over the weekend.

Admin Notes 0:00

Test 3 — Next Friday:

- Comparable in scope to Test 2

- Topics: GUI drawing/painting, event handling, Swing Timer, Observer design pattern (theory and implementation)
- Test prep page with sample problems published by end of weekend
- Study tip: create your own versions of the sample problems and share with your group

Sprint 2 Update:

- Canvas submission requirement removed (no way to submit there)
- Sprint 1 grades posted — check rubric comments for point deductions
- Monday: Sprint 2 group meetings with demos — come with Tetris running, pieces moving, no excuses about wrong branches

Sprint 3 Preview:

- Requirements are posted — read before Wednesday's class
- Friday of next week is Test 3, so Wednesday is your only day to ask Sprint 3 questions

---

Sprint 2 Q&A 3:35

Drawing Order: Fill Then Draw 4:52

If you're seeing drawing artifacts where pieces overlap or outlines appear inconsistent, the fix is straightforward: **always fill first, then draw**. Keep these two calls together with nothing else in between.

One Method for Drawing Blocks

Every Tetris piece is made of blocks. Consider a helper method like `drawBlock(Graphics2D g, int x, int y, Color fill, Color outline)` — one place where fill and draw happen together. This eliminates the problem of having fill/draw calls scattered across multiple locations.

Don't Worry About Rendering Performance 7:16

At worst, Tetris renders once per second (or maybe 10 times at high levels). This isn't stressing your system. Focus on the Observer design pattern, the architecture, and making your graphics look good — not micro-optimizing rendering.

Future Coursework

Performance-sensitive rendering becomes important in TCSS 490 Game Programming (Doctor Marriott, winter quarter). For this project, correctness and architecture matter more.

Loggers in Sprint 2 9:35

For debugging your view code, `System.out.println` or loggers are both fine — whatever is most convenient. These will be removed for production code anyway. Don't feel pressured to implement a full logging framework in the view.

For production applications, logging user interactions would be useful for crash reporting during alpha/beta testing. But for this frontend code, print statements are sufficient for debugging.

> **Use the Debugger**
>
> IntelliJ has a good debugger, though be aware that GUI programming is multi-threaded, which can make debugging trickier (different threads, different call stacks).

MVC: Model-View-Controller 12:41

Building on Wednesday's Observer pattern discussion, the full architecture is **MVC** — Model-View-Controller.

Views: Pure and Otherwise 13:10

The **model** (the Tetris game) is the **subject** — the class that fires property change events. The **views** are the **subscribers** — the classes that care about model state.

Some views are **pure views** — they only display model state and never need a direct reference to the model:

View	Pure View?	Why?
Board panel	Yes	Only displays the game board; doesn't call any model methods
Next piece panel	Yes	Only displays the next piece; no model interaction needed
Container (JFrame/JPanel)	No	Needs model reference for timer and key listener commands

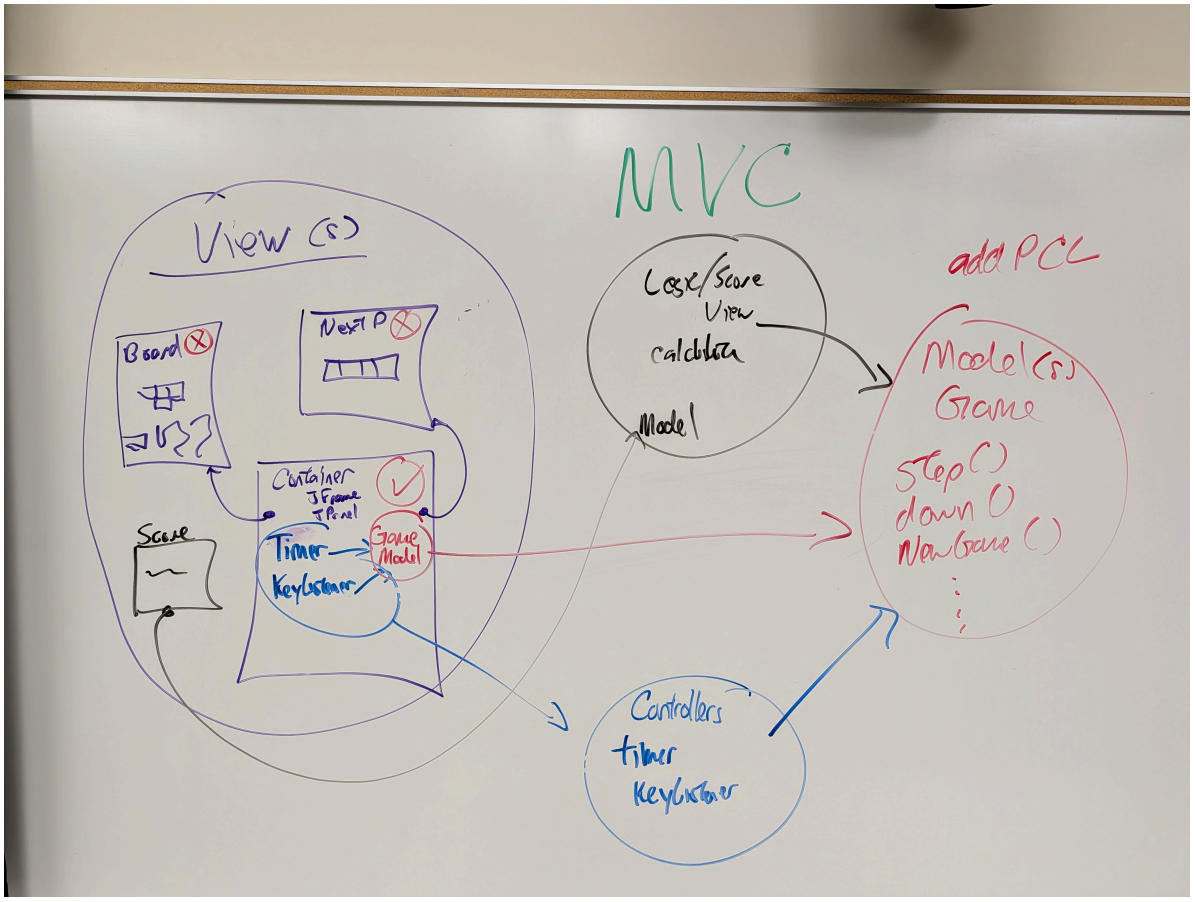
Pure views don't need the model — they just need to be `PropertyChangeListener`s registered with the model.

The Container: View + Controller 18:10

The container (your top-level JFrame or JPanel) holds everything together:

- **Board panel** and **Next Piece panel** (pure views)
- **Timer** — calls `game.step()` each tick
- **Key listener** — translates key presses to model commands (`step()`, `drop()`, `rotate()`)
- **Game model reference** — the one place the model is stored

---



Controllers 23:40

The timer and key listener are **controllers** — units that tell the model to do something:

- Timer ticks → `game.step()`
- Down arrow → `game.step()`
- Space bar → `game.drop()`
- Left/right arrows → `game.rotate()`

Controllers are not part of the Observer design pattern itself. The Observer pattern only describes the **model** and the **view**. The **controller** is what the MVC pattern adds — it's what changes the model's state.

In modern systems, controllers are typically embedded within view classes rather than being separate classes. Your timer and key listener live in the container class, which is itself both a view (responds to game state changes like pause/end) and a controller (sends commands to the model).

> ⚠ **The Container Is Also a View**
>
> When the model says "game paused" → stop the timer. When the model says "game ended" → stop the timer. The container must be a `PropertyChangeListener` too — otherwise your timer has no way to know when to stop.

Observer Pattern vs. MVC 26:50

Concept	Observer Design Pattern	Model-View-Controller
Describes	Model ↔ View relationship	Model + View + Controller
Missing piece	What changes the state?	Controller changes the state
Scope	A design pattern	A composite pattern / architectural pattern

MVC is implemented across virtually all modern GUI frameworks — desktop, mobile, and web.

Wiring It All Together 28:56

To make a panel an observer:

1. **Implement** `PropertyChangeListener` — the class must have `propertyChanged(PropertyChangeEvent)`
2. **Register with the model** — call `game.addPropertyChangeListener(boardPanel)`

The **container** is where both the model and the panels exist, so it's the natural place to wire them together:

```
// In the container constructor
myGame.addPropertyChangeListener(myBoardPanel);
myGame.addPropertyChangeListener(myNextPiecePanel);
myGame.addPropertyChangeListener(this); // container itself is a view
```

> **Keep All Registrations in One Place**
>
> If you register all property change listeners in one location and a change needs to happen, you know exactly where to look. Scattering registrations across multiple classes makes maintenance harder.

Sprint 3 Requirements 35:25

Sprint 3 focuses on polish and extra features:

1. **Game end handling** — When the game ends (by control or by filling up), the frontend handles it gracefully. No crashes. Inform the user the game is over.
2. **Score system** — The score lives in a **separate logic package** (not in the view, not in the model). The score class is both:
   - A **view** upon the game model (listens for cleared rows, new pieces, etc.)
   - A **model** for the score panel (fires its own property change events)
3. **Extra feature** — One additional feature on both frontend and backend. Required, not extra credit. Be creative — use the suggestions provided or come up with your own.

Score as Model and View 39:31

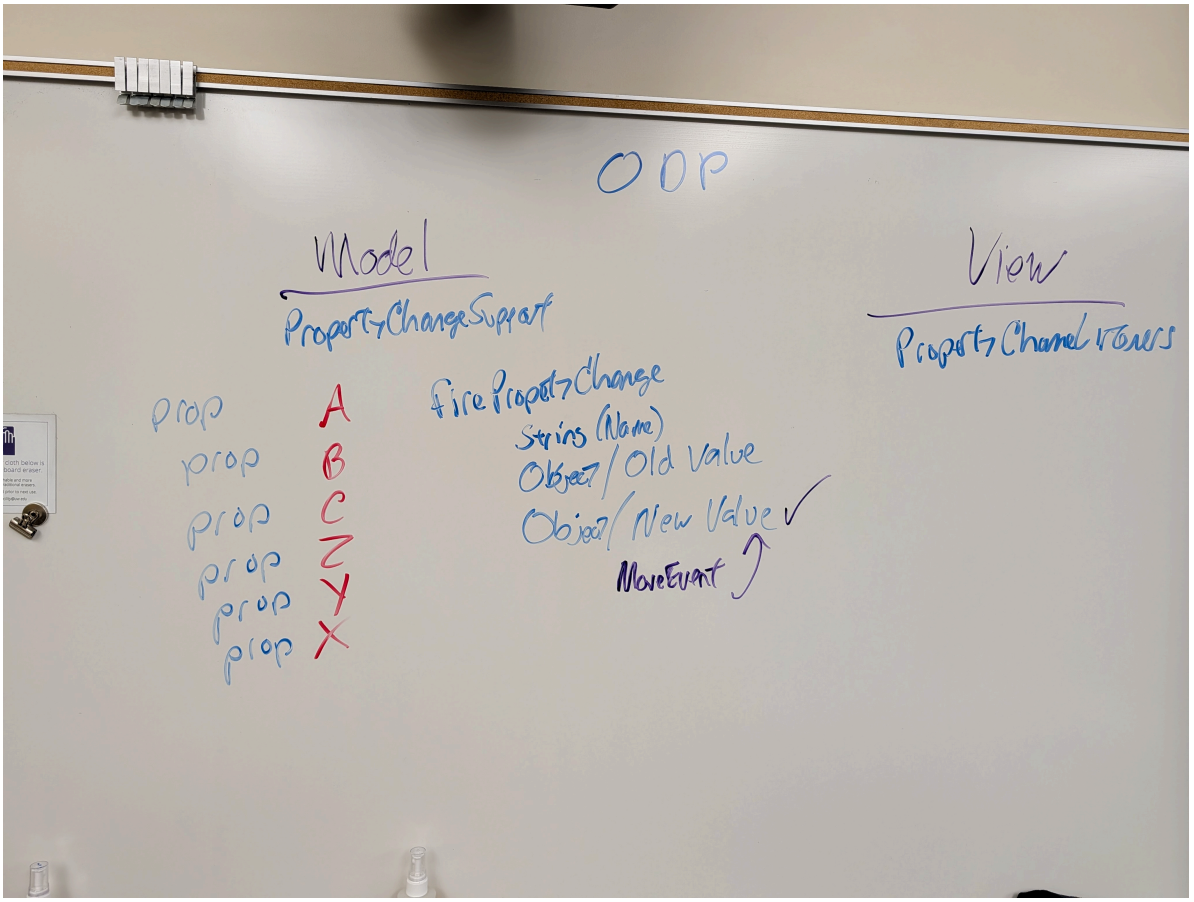
The score demonstrates the Observer pattern at a **micro scale** within the larger macro-scale architecture:

```
Game Model → (fires events) → Score Logic → (fires events) → Score Panel
             [score is a view]               [score is a model]
```

This nested observer pattern is very common in real applications.

PropertyChangeSupport Deep Dive 46:04

The model uses `PropertyChangeSupport` — composed (not extended). The Tetris game class has an instance field of type `PropertyChangeSupport`, not `extends PropertyChangeSupport`.



Properties 47:55

Each piece of mutable state that views care about is a **property**: next piece, current moving piece, frozen blocks, game state (running/paused/ended), etc.

firePropertyChange 50:27

The method takes three arguments:

Parameter	Type	Purpose
Property name	`String`	Identifies which property changed
Old value	`Object`	The previous value
New value	`Object`	The current value

> **String-Based Property Names Are Fragile**
>
> The compiler can't verify property name strings. If the model fires `"next piece"` but the listener checks for `"Next Piece"`, it silently fails. This is not a great design decision — and it's why we subvert it.

Subverting the Framework with Sealed Interfaces 53:04

Instead of relying on property name strings and casting `Object` values, the Tetris API defines a **sealed interface** for game events:

```
public sealed interface GameEvent
    permits MoveEvent, InvalidMoveEvent, NewGameEvent,
            NPCMoveEvent, CollisionEvent {
    // sealed = only these classes can implement this interface
}
```

Because the interface is sealed, only the explicitly permitted classes can implement it. This enables **exhaustive compile-time checking** in switch expressions.

Instead of inspecting the property name and casting the old/new values, you:

1. Check if the new value is a `GameEvent`
2. Use a switch expression to match on the specific event type

No string matching. No manual casting. Compile-time safety.

PropertyChangeListener Implementation 56:04

The Game Panel (Pure View) 59:27

The game panel `implements PropertyChangeListener` — making it a view. The `propertyChanged` method follows a consistent pattern:

```
@Override
public void propertyChanged(final PropertyChangeEvent theEvent) {
    if (theEvent.getNewValue() instanceof GameEvent event) {
        switch (event) {
            case GameEvent.MoveEvent moveEvent -> handleMove(moveEvent);
            case GameEvent.InvalidMoveEvent invalidMove -> { /* ignore */ }
```

```
            case GameEvent.NewGameEvent newGame -> handleNewGame(newGame);
            case GameEvent.NPCMoveEvent npcMove -> handleNPCMove(npcMove);
            case GameEvent.CollisionEvent collision -> handleCollision(collision);
        }
    }
}
```

Key modern Java features used here:

Pattern Matching for instanceof 1:02:00

```
if (theEvent.getNewValue() instanceof GameEvent event) {
    // 'event' is automatically cast to GameEvent
    // Only in scope within these braces
}
```

The `instanceof` check and cast happen in one expression. The variable `event` is scoped to the `if` block — you can't use it outside the braces because the runtime check wouldn't be guaranteed.

Switch Expressions with Type Matching 1:03:33

The modern switch expression differs from the old switch statement:

Feature	Old Switch Statement	Modern Switch Expression
Syntax	`case VALUE:` (colon)	`case TYPE var ->` (arrow)
Matches on	Values (1, 2, "hello")	Types (`MoveEvent`, `NewGameEvent`)
Evaluates to	Nothing (statement)	A value (expression)
Fall-through	Yes (needs `break`)	No

Because `GameEvent` is sealed, the compiler performs an **exhaustive case check**. If you forget to handle one event type, you get a **compile-time error** — not a runtime bug.

> **Avoid `default` in Sealed Type Switches**
>
> You *can* add a `default` case, but don't. The `default` will catch any new event types added later, which means you lose the compile-time exhaustiveness check. If a new event is added to the model, you **want** the compiler to force you to handle it explicitly.

Handling Events You Don't Care About 1:07:01

Your next piece panel doesn't care about move events. But you still need explicit cases for exhaustiveness:

```
case GameEvent.MoveEvent moveEvent -> { /* not relevant to this panel */ }
case GameEvent.CollisionEvent collision -> { /* not relevant to this panel */ }
```

This is preferred over `default` because:

- New events trigger compile-time errors (you must consciously decide to handle or ignore them)
- The intent is clear — you explicitly chose to do nothing

> **Use Helper Methods**
>
> Keep the switch expression clean. Each case should do one thing — if it needs to do three things, extract a helper method. Don't put curly braces with multiple statements in switch cases.

The Controller's propertyChanged 1:08:11

The controller class follows the same structure — it's also a `PropertyChangeListener`. Even though it's primarily a controller (sending commands to the model), it responds to model state changes too (enabling/disabling buttons, starting/stopping timers).

Same pattern: check `instanceof GameEvent`, switch on type, handle or ignore each case explicitly.

Key Listener Strategies 1:11:04

The `KeyListener` interface has three methods: `keyPressed`, `keyReleased`, `keyTyped`. For Tetris, you want **`keyPressed`** — that's what fires when a key is held down.

The If-Else Approach 1:12:00

The straightforward approach — `if` the key is W, do this; `if` the key is S, do that. Works fine, but:

- Gets unwieldy with many keys
- Makes runtime key rebinding very difficult

The Map-Based Approach 1:13:21

A `Map<Integer, Runnable>` maps key codes to actions:

```
private final Map<Integer, Runnable> myKeyMap = new HashMap<>();

// Setup
myKeyMap.put(KeyEvent.VK_LEFT, myGame::left);
myKeyMap.put(KeyEvent.VK_RIGHT, myGame::right);
myKeyMap.put(KeyEvent.VK_DOWN, myGame::step);
myKeyMap.put(KeyEvent.VK_SPACE, myGame::drop);

// In keyPressed:
final Runnable action = myKeyMap.get(theEvent.getKeyCode());
if (action != null) {
    action.run();
}
```

Using `Runnable` as the value type enables **method references** for clean, concise mappings. Maps can be modified at runtime, making dynamic key rebinding straightforward.

> **Extra Feature Idea**
>
> Dynamic key rebinding using the map-based approach would count as a frontend extra feature for Sprint 3.

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