lectures # week-9

Week 9: Observer Design Pattern (Wednesday, March 4, 2026)

Lecture Recording

Watch on Panopto

Related Assignment

This lecture covers concepts for Group Project Sprint 2.

Lecture Preview 0:09

Today's agenda:

1. **Test 2 review** — Brief discussion of results and curve
2. **Sprint 2 requirements & design** — Separation of concerns, command-only API, where the model lives
3. **Observer design pattern** — The tight coupling problem and the polymorphic solution
4. **Property Change framework** — Java's implementation of the Observer pattern for Swing

Coming Friday

Implementation of the Observer pattern using `PropertyChangeListener` with code examples.

Admin Notes 0:00

Rest of the Quarter:

Week	Monday	Wednesday	Friday
9 (this week)	Group meetings (Sprint 1)	Observer pattern (today)	Observer implementation
10	Last group meeting (Sprint 2)	Wrap-up / functional programming	**Test 3**
Finals	—	Project reflection due (Canvas)	—

- Sprint 3 due **Sunday night, Week 10** (before finals week).
- Sample problems for Test 3 will be published over the weekend.
- No final group meeting for Sprint 3 — the last group meeting covers Sprint 2.

Test 3 Topics: GUI drawing/painting, animation, Swing Timer, Observer design pattern (theory and implementation with `PropertyChangeListener`). Programming is cumulative — expect to need fundamentals like for loops, reading unit tests, etc., even though they weren't taught in recent weeks.

Grades: All grades are now up to date in Canvas. The group project weight (40%) is currently calculated from only Sprint 0 (5 pts) and Sprint 1 (10 pts) out of the total 40 points — so Canvas's weighted average may look skewed. There are still 25 points of group project work remaining.

Test 2 Curve: Class average (excluding non-participants) was just under 80%. Everyone except the two perfect scores received +1.5 points. Section 4 (implementation question) was given full marks (10/10) to all students due to confusing question design.

Grading Scale Reminder: Final grades are determined by the weighted average in Canvas mapped to the grade equivalence table at the bottom of the syllabus. The scale will never be narrowed, but may be widened at the end of the quarter if needed.

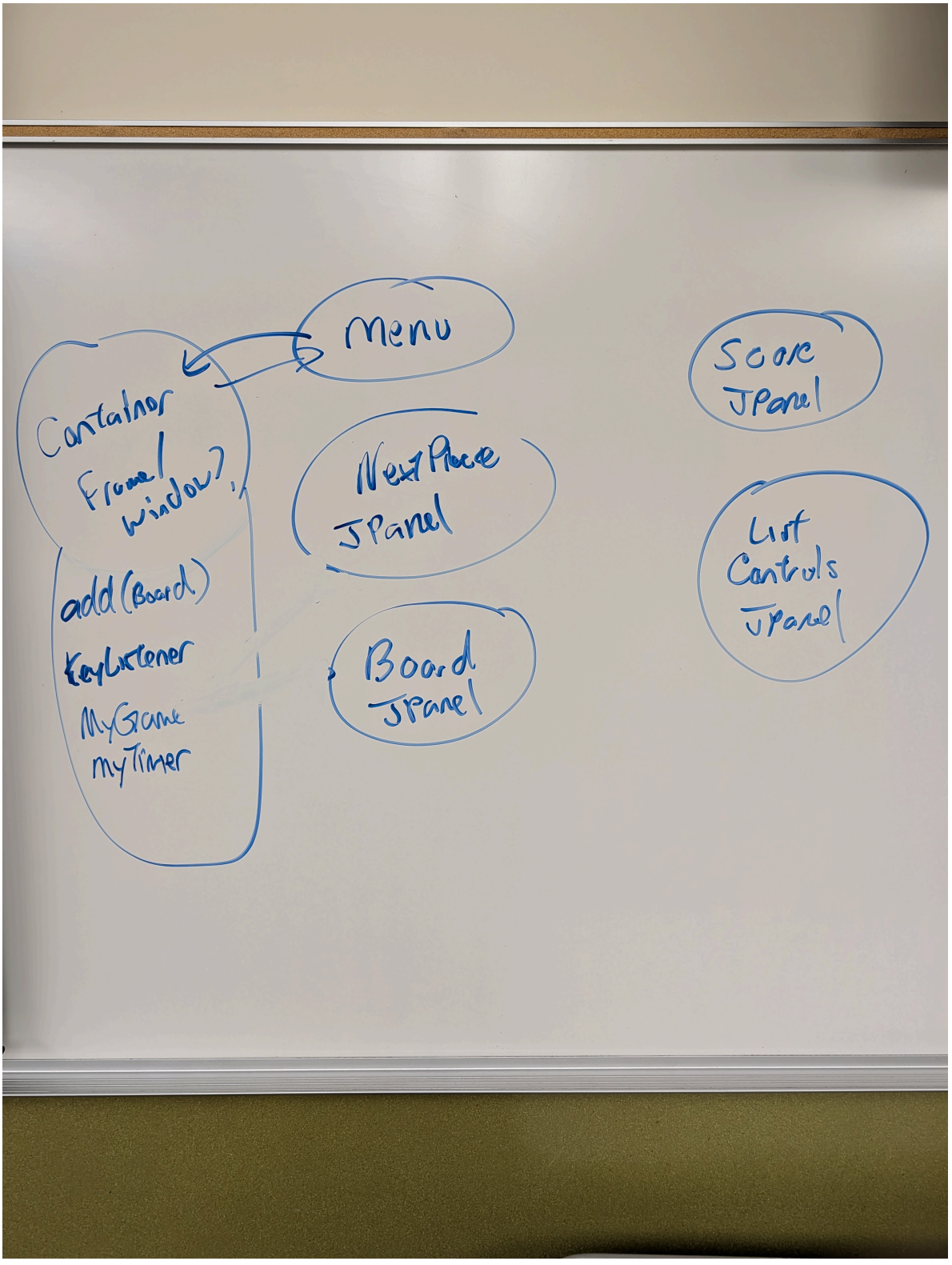
AI Tutor Beta: The course chat tutor is now embedded in the Sprint 2 assignment page. It has awareness of the course, the assignment, and prerequisites. Limited to 20 messages per session (API costs come out of the instructor's pocket). Submit bug reports if you encounter issues.

Sprint 2 Design 20:25

Sprint 2 requirements: pieces moving down the screen, frozen blocks at the bottom, next piece changing for all seven tetrominoes (centered), keyboard controls.

Separating Responsibilities 21:08

Looking at a running Tetris application, six distinct responsibilities emerge:



Responsibility	Description
Container (Frame/Window)	Holds everything; owns the game model, timer, and key listener

Responsibility	Description
Menu	Menu bar functionality
Board (JPanel)	Renders the active game board with moving pieces
Next Piece (JPanel)	Displays the upcoming tetromino
Score (JPanel)	Shows the current score
List Controls (JPanel)	Displays the control key mappings (informational only)

Could you put all of this in one class? Yes. Should you? No. Each responsibility should be its own class. The next piece panel has nothing to do with the score. The controls display has nothing to do with the board. Separate your concerns.

Each Panel Is a JPanel

Board, Next Piece, Score, and Controls each `extends JPanel`. The container instantiates these objects and adds them to itself. Clean separation through composition.

The Command-Only API 28:34

Every public method in the game model returns `void`. This creates a **command-only API** — you can tell the model to do things (`step()`, `left()`, `right()`, `rotate()`, `drop()`), but you cannot ask it for information.

This has major design implications:

- Your board panel **does not need** a reference to the model
- Your next piece panel **does not need** a reference to the model
- Your score panel **does not need** a reference to the model
- **Only the container** needs the model reference (for commands via key listener and timer)

Student Question

Q: If the panels can't query the model, how do they know what to display?

A: That's exactly the problem the Observer design pattern solves — keep reading.

Where Things Live 29:52

The **container** is the natural home for:

- **The game model** — only place that needs to send commands
- **The timer** — calls `step()` on the model each tick
- **The key listener** — translates key presses into model commands (left, right, rotate, drop, pause)

The individual panels (board, next piece, score) receive information through the Observer pattern — they never touch the model directly.

> **Work With What the Model Gives You**
>
> The model's coordinate system and piece representation may not match your expectations. Don't try to "fix" the data — transpose it for your display. You can't change the jar file. In the real world, you'll often build a frontend on a backend you didn't design.

What Is a Design Pattern? 38:13

Across different systems, languages, and frameworks, certain types of problems keep appearing. When developers notice the same problem and the same style of solution recurring across System A, System B, and System C, that solution gets formalized as a **design pattern**.

Key characteristics:

- **Language-agnostic** — the Observer pattern exists in Java, C#, JavaScript, Python, and web architectures
- **Implementation-agnostic** — the pattern describes the *concept*; specific frameworks provide concrete implementations
- **Problem-first** — if you don't have the problem, you don't need the pattern

The concept of design patterns originated in **architecture** (actual building design, not software), where architects noticed recurring solutions to common structural problems — like how to arrange doors along a hallway or distribute windows across a building facade.

> **Reading**
>
> See Introduction to Design Patterns for a deeper overview.

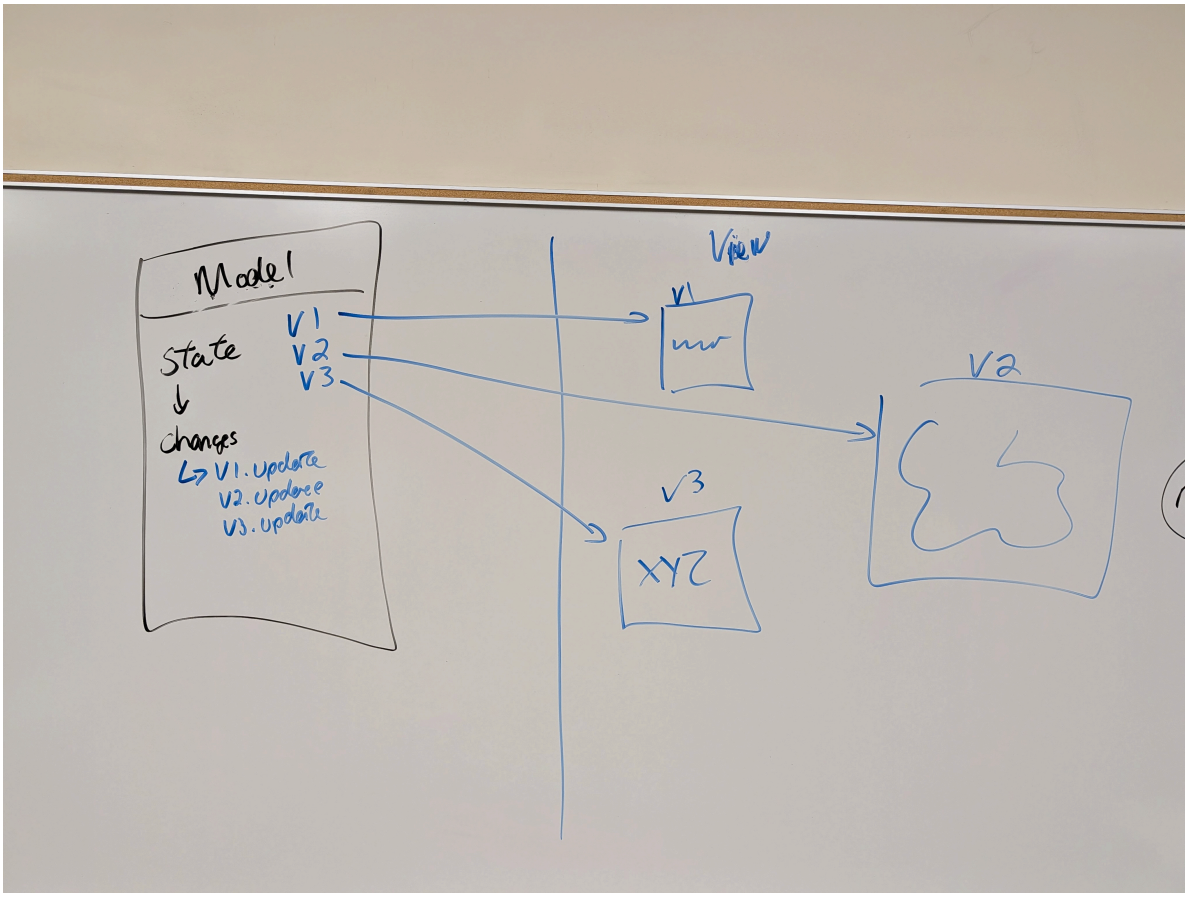
The Observer Design Pattern 44:48

Why Separate Model and View? 45:05

A simple calculator can live in one file — it has minimal state (two operands, an operator, a result). But as applications grow in complexity (Road Rage, Tetris), separating application **state** (model) from the **presentation** (view) becomes essential for organization and maintainability. The view might not even be a GUI — remember, Sprint 0 used a console interface.

The Problem: Tight Coupling 50:44

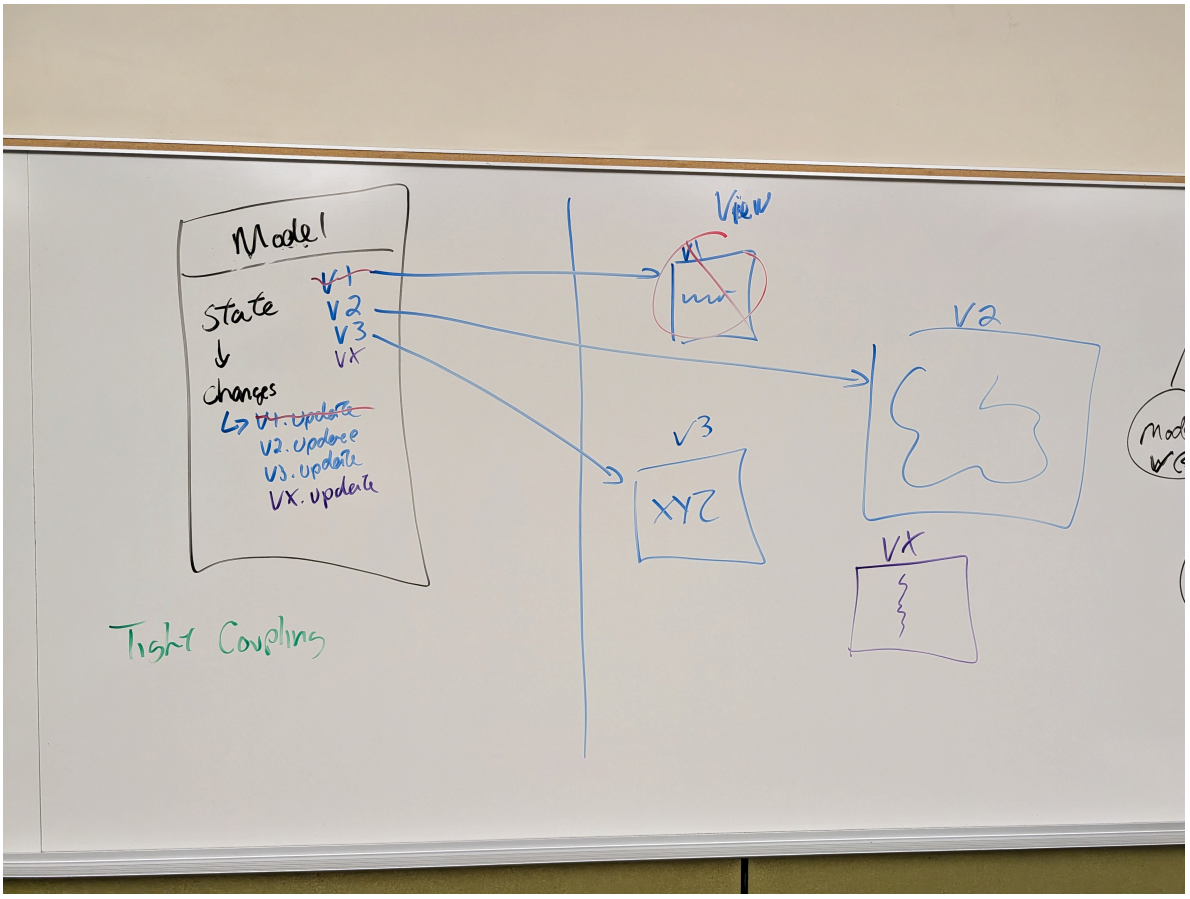
Consider a model with state that multiple views need to display. In a naive approach, the model stores explicit references to every view:



When state changes, the model calls `v1.update()`, `v2.update()`, `v3.update()`. This works — until requirements change.

Why Tight Coupling Breaks Down 55:37

The client gives feedback: "V1 sucks — get rid of it. Add VX instead. And plan for more views in the future."



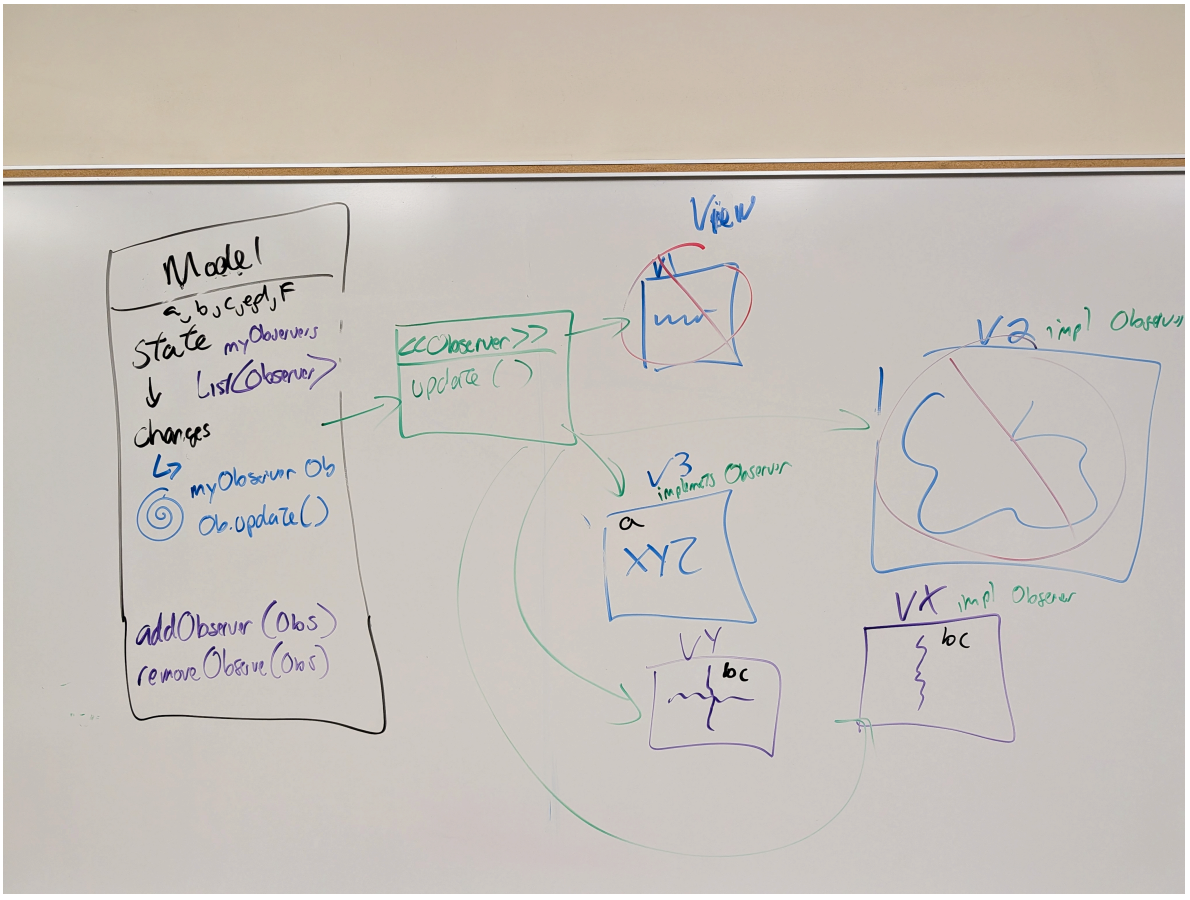
Now you must:

1. Go into the **model code** and delete all V1 references
2. Add explicit VX references to the model
3. Add `vx.update()` calls everywhere state changes
4. Figure out which of the 10 state properties VX cares about (even though you're the model developer and shouldn't need to know)
5. Recompile the model

Every view change forces model changes. That's **tight coupling** — and it's the problem the Observer pattern solves.

The Solution: Polymorphism 59:28

The Observer pattern introduces an **interface** between the model and its views:



The pattern works in five steps:

1. **Define an `Observer` interface** with an `update()` method
2. **All views implement `Observer`** — guaranteed to have `update()` via the compiler
3. **The model stores a `List<Observer>`** instead of explicit view references
4. **The model exposes `addObserver(Observer)` and `removeObserver(Observer)`**
5. **When state changes**, loop through observers and call `ob.update()` on each

Why this works:

Scenario	Tight Coupling	Observer Pattern
Add a new view	Modify model code, recompile	Call `model.addObserver(newView)` — no model changes
Remove a view	Modify model code, recompile	Stop calling `addObserver` for it — no model changes

Scenario	Tight Coupling	Observer Pattern
View doesn't care about a state change	Model must know what each view cares about	View simply ignores the update

> ⚠ **The Model Still Has References**
>
> The model still holds references to views — but **only through the Observer interface**. It doesn't know about board panels, score panels, or next piece panels. It only knows about "observers." This is loose coupling through polymorphism.

How This Applies to Tetris 1:08:00

The Tetris model is in a **jar file** — you literally cannot modify it. But the observer infrastructure is already built in. Your panels become observers, register themselves with the model, and receive updates when state changes. That's how your board panel knows pieces are moving even though the API is command-only.

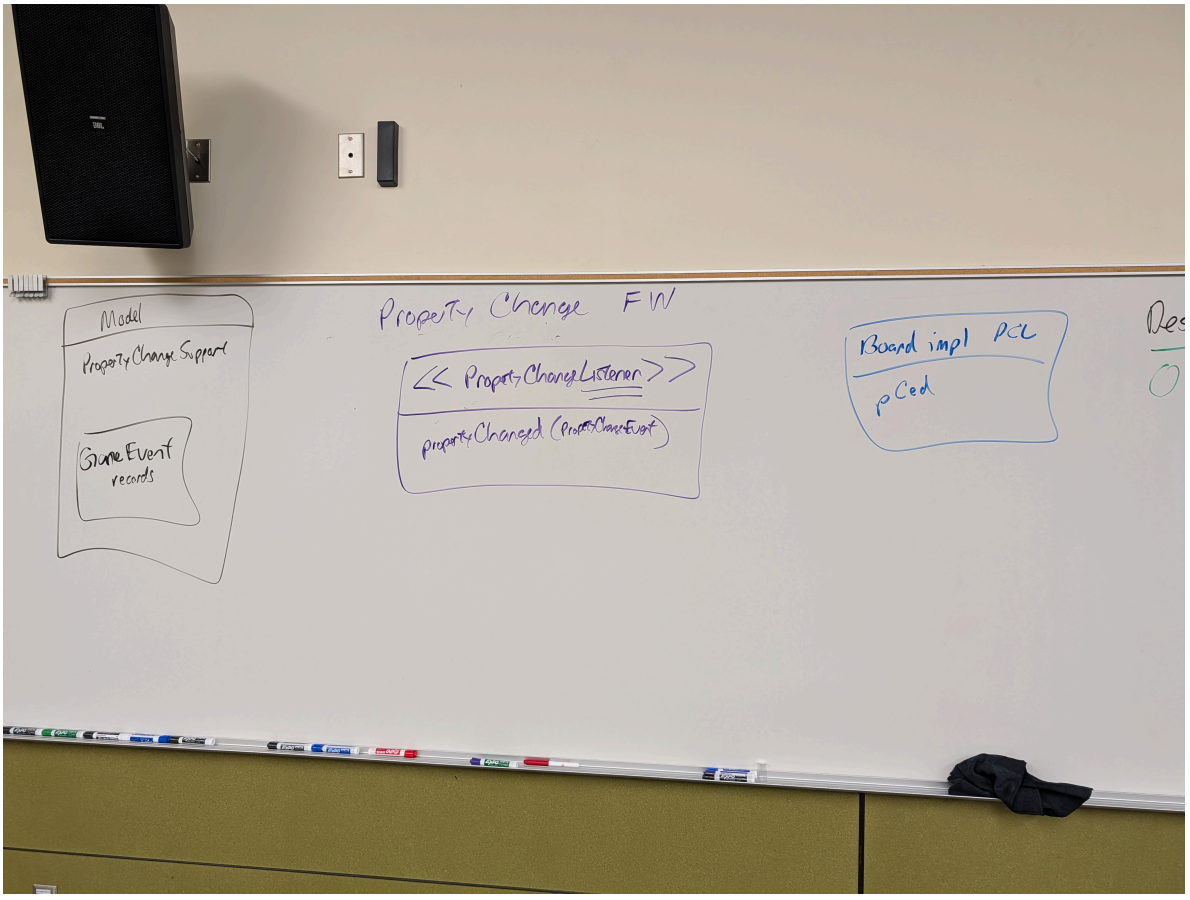
Connecting to Event Listeners 1:13:21

If this pattern feels familiar, it should. **Event listeners are observers.** When you add an `ActionListener` to a button, you're registering as an observer of that button's click events. The Observer design pattern is the foundation of Java's entire event handling system.

> 🔥 **Reading**
>
> See The Observer Pattern for the full theory and code examples.

The Property Change Framework 1:11:45

Java's implementation of the Observer pattern for our Swing applications uses the **Property Change framework**.



View Side: PropertyChangeListener 1:12:20

The observer interface in this framework is `PropertyChangeListener`. Your view classes implement it:

```
public class BoardPanel extends JPanel implements PropertyChangeListener {

    @Override
    public void propertyChanged(PropertyChangeEvent theEvent) {
        // Inspect theEvent to determine what changed
        // Update display accordingly
    }
}
```

The `PropertyChangeEvent` object wraps the information about what changed. Inside it, you'll find game-specific `GameEvent` records that tell you exactly what happened — piece moved, next piece changed, frozen blocks changed, etc.

Model Side: PropertyChangeSupport 1:17:08

The model uses `PropertyChangeSupport` to manage its list of observers and fire events:

Component	Role
`PropertyChangeSupport`	Stores the list of listeners; handles add/remove/fire
`firePropertyChange(...)`	Notifies all registered listeners of a state change
`GameEvent` (interface)	Defines the types of changes (piece moved, next piece changed, etc.)
Records implementing `GameEvent`	Carry the specific data for each event type

This Is Already Built Into the Model

The Tetris model jar already has this observer infrastructure. You don't build the model side for Sprint 2 — you build the **view side**. Your panels implement `PropertyChangeListener`, register themselves with the model, and handle the events they care about.

Sprint 3 Preview

In Sprint 3, you'll need to make modifications to the model. That's when you'll work with `PropertyChangeSupport` and `firePropertyChange` directly.

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