assignment # group-project

Sprint 3 – Polish & Extra Features

School of Engineering and Technology, University of Washington Tacoma
TCSS 305 Programming Practicum, Winter 2026
Value: 10% of the course grade

Due Date

Sunday, 15 March 2026, 23:59:59

Description

Sprint 3 is the final sprint. You will polish your Tetris game into a complete, playable experience by implementing end game handling, a scoring system with levels, a key binding display, and two extra features — one front-end and one back-end. This sprint rewards both solid engineering and creativity.

Learning Objectives

By completing this sprint, you will:

- React to model state changes (`GameState.OVER`) and provide clear user feedback
- Implement a front-end scoring algorithm driven by back-end events
- Use `javax.swing.Timer` delay adjustments to implement game leveling
- Extend the back-end model with new `GameState` values and `PropertyChangeEvent` firing
- Apply the Observer Design Pattern in a feature you design and implement
- Deliver a polished, complete application as a team

Before You Begin

Ensure you have completed:

- [x] All Sprint 2 requirements (game controls, timer, PropertyChangeListeners)
- [x] Merged `sprint2` into `main`
- [x] Reviewed the Back-End API Documentation
- [x] Discussed extra feature choices with your team
- [x] Completed the Project Setup below (replace the JAR with source code)

🔧 Project Setup — Back-End Source Code

Sprint 3 requires you to modify the back-end model (Requirement 6). To do this, you need the **source code** instead of the compiled JAR.

Step 1: Download the Source

Download the back-end source code: ⬇ tetris-model-source.zip

Step 2: Remove the JAR from Your Project

1. In IntelliJ, open **File → Project Structure → Libraries**
2. Find `tetris-model-1.0.0.jar` and **remove** it
3. Click **OK** to apply

> ⚠ **Warning**
>
> If you skip this step, IntelliJ will use the compiled JAR classes instead of your source code changes. Your back-end modifications will not take effect.

Step 3: Add the Source Files

1. Extract the zip file — it contains the folder `edu/uw/tcss/model/` with 9 Java files
2. Copy the `edu` folder into your project's `src/` directory
3. Your project structure should look like this:

```
src/
├── edu/
│   └── uw/
│       └── tcss/
│           ├── app/
│           │   └── ... (your existing code)
│           ├── model/          ← from the zip
│           │   ├── GameControls.java
```

```
│           │   ├── GameEvent.java
│           │   ├── TetrisGame.java
│           │   └── ... (6 more files)
│           └── view/
│               └── ... (your existing code)
```

4. Verify IntelliJ recognizes the source files — you should see them in the `edu.uw.tcss.model` package with no compilation errors

Step 4: Verify

Run your game. It should behave **exactly the same** as before — the source code is identical to what was compiled into the JAR. If anything breaks, double-check that the JAR has been removed and the source files are in the correct location.

> **Tip**
>
> Commit the source files to your `sprint3` branch so all team members have them.

Requirements

Requirement 1: Branch Setup

Create a branch from your `main` branch called `sprint3`. All Sprint 3 work should be committed to this branch.

> **Caution**
>
> Do NOT commit, merge, or alter the `sprint3` branch after the due date/time. I will ignore any commits after the deadline.

Requirement 2: End Game

> **Guide**
>
> - The Observer Pattern — Listening for GameState changes from the model
> - Animation with javax.swing.Timer — Stopping the timer when the game ends

React to `GameState.OVER`:

- It must be **apparent to the user** that the game is over
- Add a **special effect** when the game ends — this is an opportunity for creativity (visual effect, animation, sound, dialog, etc.)
- Provide an option for the user to **end the current game** (menu item, key press, or both)
- The event handler should **ONLY** call a single method in the `GameControls` API

> ⚠ **Front-End Agnosticism**
>
> The front-end should remain agnostic about **HOW** the game ended. It simply reacts to the `GameState.OVER` event — it does not check board state, piece positions, or any other condition to determine if the game is over.

Requirement 3: Key Binding Display

> 📖 **Guide**
>
> Handling Key Events — Review key binding patterns from Sprint 2

Display a guide showing which keys are bound to which actions. This may be:

- Displayed in the gameplay window (e.g., in the information panel)
- In a menu item (e.g., Help > Controls)
- Another accessible location

The display must be visible and easy to find during gameplay.

Requirement 4: Score Keeping and Display

> 📖 **Guide**
>
> - The Observer Pattern — Listening for model events, implementing PropertyChangeSupport, and defining sealed event types
> - Animation with javax.swing.Timer — Adjusting timer delay for level-based speed increases

Display Requirements:

The front-end must display:

- Current score
- Current level
- Total lines cleared
- Indication of when the next level will be reached

Score Logic Class:

Create a dedicated score logic class that acts as an intermediary between the model and the view. This class:

- **Observes the model** — registers as a `PropertyChangeListener` on the `TetrisGame` to listen for piece freeze and line clear events
- **Maintains state** — tracks current score, level, and lines cleared (this class has state but no GUI code)
- **Fires its own events** — implements `PropertyChangeSupport` and fires `PropertyChangeEvent`s that the view listens for (score updated, level changed, etc.)
- **Defines a sealed event interface** — create your own sealed interface with record implementations to define the score-related events this class fires, following the same pattern as the back-end's `GameEvent` sealed interface

This is the Observer Design Pattern applied at a smaller scale: the model notifies the score logic, and the score logic notifies the view. Since this class isn't part of the back-end model and isn't a view component, place it in its own package (e.g., `logic`) to reflect its role as an intermediary.

> **Tip**
>
> Review the sealed event types section of the Observer Pattern guide. The back-end uses a sealed `GameEvent` interface with record implementations like `MoveEvent` and `CollisionEvent`. Your score logic class should follow this same pattern — define a sealed interface (e.g., `ScoreEvent`) with records for each type of score-related change. Examples: `ScoreChanged`, `LevelChanged`, `LinesClearedChanged`, etc.

Leveling:

- Define requirements to level up (standard: lines cleared; alternatives: score, combination, etc.)
- As the game levels, the current piece must advance faster (decrease the Timer delay)

- Determine the speed increase rate — it should be noticeable but still playable

Scoring Algorithm:

Event	Points
Piece freezes	+4 points

Level	1 line	2 lines	3 lines	4 lines
1	40	100	300	1200
2	80	200	600	2400
3	120	300	900	3600
...				
10	400	1000	3000	12000
n	40 x n	100 x n	300 x n	1200 x n

Requirement 5: Required Extra Feature — Front-End (Choose 1)

You must implement **one** of the following front-end features. Additional features may be considered for extra credit.

Sound/Music

- Add sound effects for actions (move, rotate, drop, line clear, game over, etc.)
- Add music for gameplay, intro, and/or end game
- Include music files in your Git repository
- Add UI components and/or key bindings to control music and sounds (on/off, mute, etc.)
- If key bindings control sound/music, include them in the Key Binding Display
- Must work on both Mac and Windows

User Changeable Key Bindings

- Add UI components to allow key binding changes at runtime
- Key bindings must change at runtime (no restart required)
- Changes must be reflected in the Key Binding Display
- Persistence across program restarts is optional but looked upon highly

Persistent High Score Keeping

- Track multiple high scores
- Prompt user for 3-character initials upon achieving a high score
- High scores must persist across program restarts (file I/O)
- Include option to view the high score list
- Include option to reset all high scores

User Changeable Color Themes

- Implement at least 3 user-selectable color themes
- Each theme must alter: game board and next piece panel (at minimum)
- Add UI components to change theme
- Selected theme must persist across program restarts

Other

- **Ask first** — requirements will be determined based on the idea
- Must be approved before implementation

Requirement 6: Required Extra Feature — Back-End (Choose 1)

> **Guide**
>
> - The Observer Pattern — Publishing new GameState changes via PropertyChangeSupport
> - Java Enums — Adding new enum constants to GameState

You must implement **one** of the following back-end features. Additional features may be considered for extra credit.

> **Important**
>
> This requirement requires modifying the back-end source code. If you haven't completed the Project Setup above, do that first.

> **Note**
>
> If implementing multiple features with overlapping `GameState` values, discuss with the instructor.

Panic Mode

Add new `GameState` values based on frozen block height:

- `GameState.WORRY` — at least one block above halfway up the board
- `GameState.PANIC` — at least one block above 3/4 up the board

Requirements:

- These states function the same as `RUNNING` (pausing still works)
- Publish state changes via the Observer Design Pattern (`PropertyChangeEvent`)
- Add an API option to enable/disable Panic Mode when starting a new game
- Add a front-end UI component to toggle Panic Mode on/off
- Update JavaDoc for all back-end changes
- Front-end must **visually change** depending on mode (different colors, sounds, visual effects, etc.)

Hyper Piece Mode

Add a new `GameState` value based on random chance:

- `GameState.HYPER` — entered randomly when a new current piece is chosen (~1 in 10 pieces)

Requirements:

- This state functions the same as `RUNNING` (pausing still works)
- Publish state changes via the Observer Design Pattern (`PropertyChangeEvent`)
- Add an API option to enable/disable Hyper Piece Mode when starting a new game
- Add a front-end UI component to toggle Hyper Piece Mode on/off

- Update JavaDoc for all back-end changes
- Front-end must **visually change AND speed up/down** (must be noticeable to the player)

Other

- **Ask first** — must require `PropertyChangeEvent` framework changes and front-end visual/functional changes
- Must be approved before implementation

Requirement 7: Codebase Requirements

- Correct **all** linting tool warnings (Checkstyle, IntelliJ inspections)
- If suppressing any warnings, document the reasoning with comments

> **Tip**
>
> Run your linting tools early and often. Don't leave this until the last minute — fixing warnings in a large codebase takes more time than you expect.

Requirement 8: Daily Meetings (3 Required)

Hold at least **3** daily meetings during this sprint. Create a Google Doc to document your meetings. Each meeting should contain:

- Date and time
- Members who attended
- Location/method (Discord voice/video, Zoom, in-person)
- Each group member answers:
    a. What have you done since the last meeting?
    b. What will you do before the next meeting?
    c. Is there anything the group can do to help you?

Put the link to the Google Doc in your `README.md` (with view access).

> **Tip**
>
> Sprint 3 has more requirements and more moving parts. Three meetings is the minimum — consider meeting more often, especially as the deadline approaches.

Requirement 9: Executive Summary

Add the following sections to your `executive-summary.md`:

- **Sprint 3 Contributions** — for each group member, describe their contribution to Sprint 3
- **Sprint 3 Meetings** — link to meeting minutes, describe other meetings (where/when/how, what was discussed), describe primary forms of communication besides formal meetings
- **Sprint 3 Comments** — document any issues, ideas, code weirdness, etc.
- **Extra Features** — document which extra features you implemented (one front-end, one back-end)
- **Extra Credit** — document any additional extra features to be considered for extra credit

> **Important**
>
> Keep all previous sprint entries intact. The executive summary is a living document.

Guide Reference

Guide	Description
The Observer Pattern	PropertyChangeListener, PropertyChangeSupport — used for end game, scoring, and back-end extra features
Animation with javax.swing.Timer	Adjusting timer delay for leveling, stopping on game over
Handling Key Events	Key binding display and optional user-changeable key bindings
Java Enums	Adding new GameState enum constants for back-end extra features

Guide	Description
Custom Painting with Java 2D	Visual effects for end game and Panic/Hyper modes
Model-View-Controller (MVC)	Maintaining clean separation as you add scoring and extra features
Linters and Code Quality	Resolving all Checkstyle and IntelliJ warnings

---

🚀 Submission and Grading

> **Important**
>
> This is the final sprint. Plan your time carefully — extra features, scoring, end game, and linting all need to be complete.

Please see the rubric in Canvas for a breakdown of the grade for this sprint.

Submitting Your Work

1. Ensure all Sprint 3 requirements are committed and pushed to your `sprint3` branch
2. Perform one merge from `sprint3` into your `main` branch
3. Push the merge to the remote repository

I will look at both the `sprint3` and `main` branches. I will ignore any commits after the due date.

> **Tip**
>
> You can commit and push to your `sprint3` branch as often as you like. Only perform one merge into `main` at the end.