Name: Viet Hoang Pham
Student ID: 104506968

Design Report for Tetris

Design Overview

Short 6-minute video talking about my Tetris custom program:
https://drive.google.com/file/d/1GK1tSVRb80M0J5ckxRpsm1i2wdEY3yGN/view?usp=sharing

Note: The video talks about what the custom program can do, not how it can do it. Structure and implementation explanation will be detailed in this report. The game also plays music and sound effects but it is reduced to the minimum volume in the video.

Key points about what the custom program can do (Summary of the video):

The program implements the classic game Tetris. The key game logics are:

- The player will be presented with a grid-based game board where Tetriminos fall from the top of the screen.
- The game contains seven Tetrimino shapes which are called I, J, L, O, S, T, and Z.
- The player can use keyboard controls to move, rotate, and drop the Tetriminos.
- The game introduces levels of the game, which indicates the speed the game moves the Tetriminos down over time. As the game progresses, the level increments lead to the Tetrominos fall faster, increasing the difficulty.
- The player must place these Tetriminoes to create lines without gaps. Once a line is completed, it is cleared automatically, and the lines above it will be moved down.
- There are 4 game modes in the game: Classic, NES, 40 Lines, and Blitz with different game rules.
- The queue is introduced to show the player what are the next pieces. The amount of queued Tetriminos being displayed is based on game mode (1 in Classic, NES; 5 in 40 Lines, Blitz).
- 2 queue randomness algorithms are introduced: The first one is completely random, where a type of Tetrimino has the chance of being created 5 times in a row. The second one is 7-bag random, where a type of Tetrimino can only appear once in every 7 turns (the algorithm randoms the permutations of 7 Tetriminoes instead of the Tetriminoes themselves).
- The randomness algorithm is chosen based on the game mode: In Classic and NES, the game chooses completely random, while the game chooses 7-bag random algorithm in Blitz and 40-Lines game modes.
- The player can also hold a Tetrimino in Blitz and 40-Lines game mode, which means that when the current Tetrimino is not what they want, they can always hold/swap that Tetrimino.
- Depends on each game mode, the objective of the game is that you will get points for the number of lines cleared (in Classic, NES, Blitz) based on the number of lines cleared, or have the fastest time possible to clear 40 lines (in 40 lines).

- Normally, the game ends when the tetrominoes stack up to the top of the playing field without any space left for new pieces. But it can also end based on some game modes such as when the game time reaches 2 minutes in Blitz or clears 40 Lines in 40 Lines
- The player can save their scores with their name, which will be recorded into the BestScore.json file. The recorded scores are always sorted before being saved and rewritten.
- The best score/fastest time is always displayed during the game. It's loaded from the mentioned JSON file.
- The game also plays background music, and it changes depending on the current state.
- The game has 5 main states with different displays and controls:
    - Main Menu where the player can choose the game mode they want to play with different game rules. They can also click on the Help button to open Help state.
    - Help where the player can read about the controls, shortcuts, and game modes configurations/rules in the game.
    - Ingame where the player plays Tetris (the game logic).
    - Pause Menu where the player can pause the current game. They can either continue their current game, restart to have a new one, or return to the Main Menu state.
    - Game Over where the player can have a look at their top 5 scores/times of each game mode they are playing. They can type their name to save their current scores in this state.

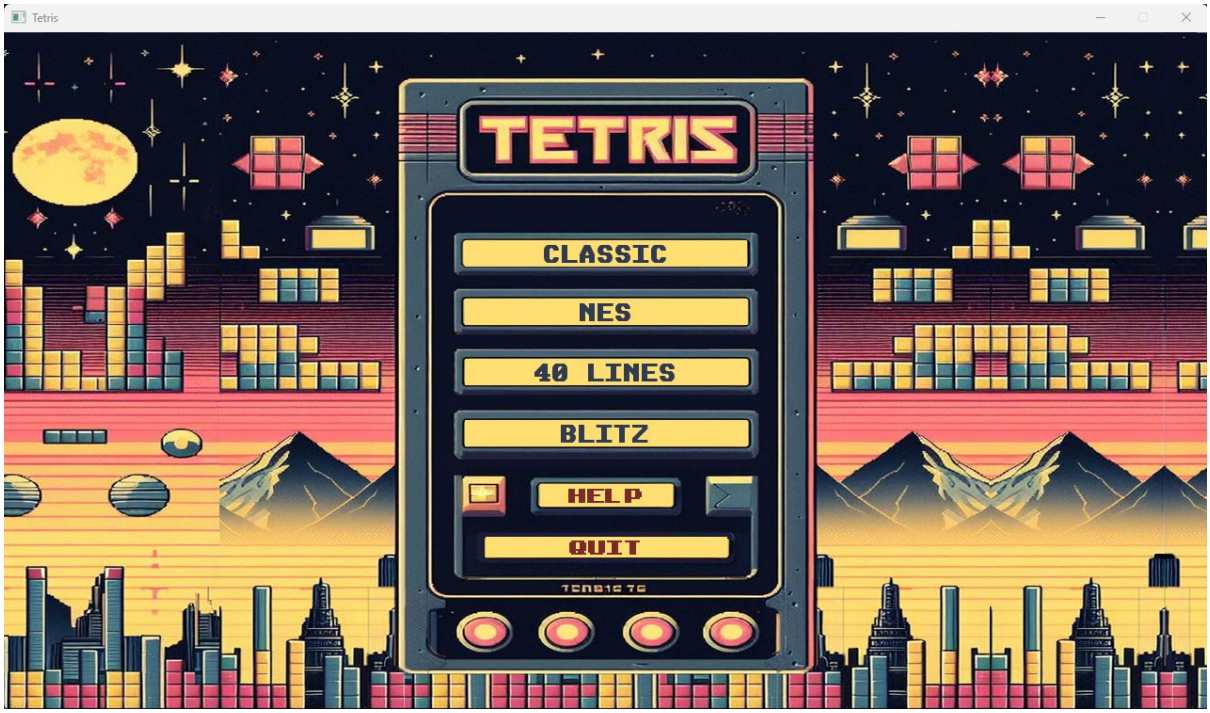

Figure 1: Main Menu of Tetris Custom Program

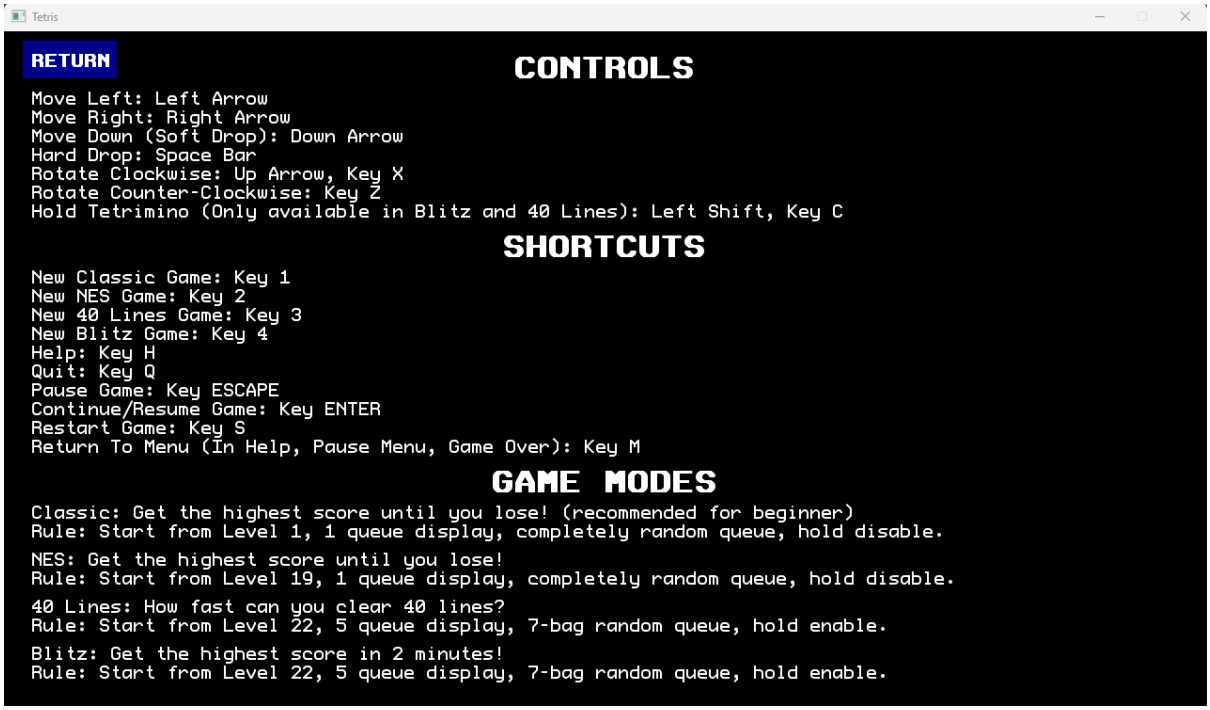

Figure 2: Help State of Tetris Custom Program

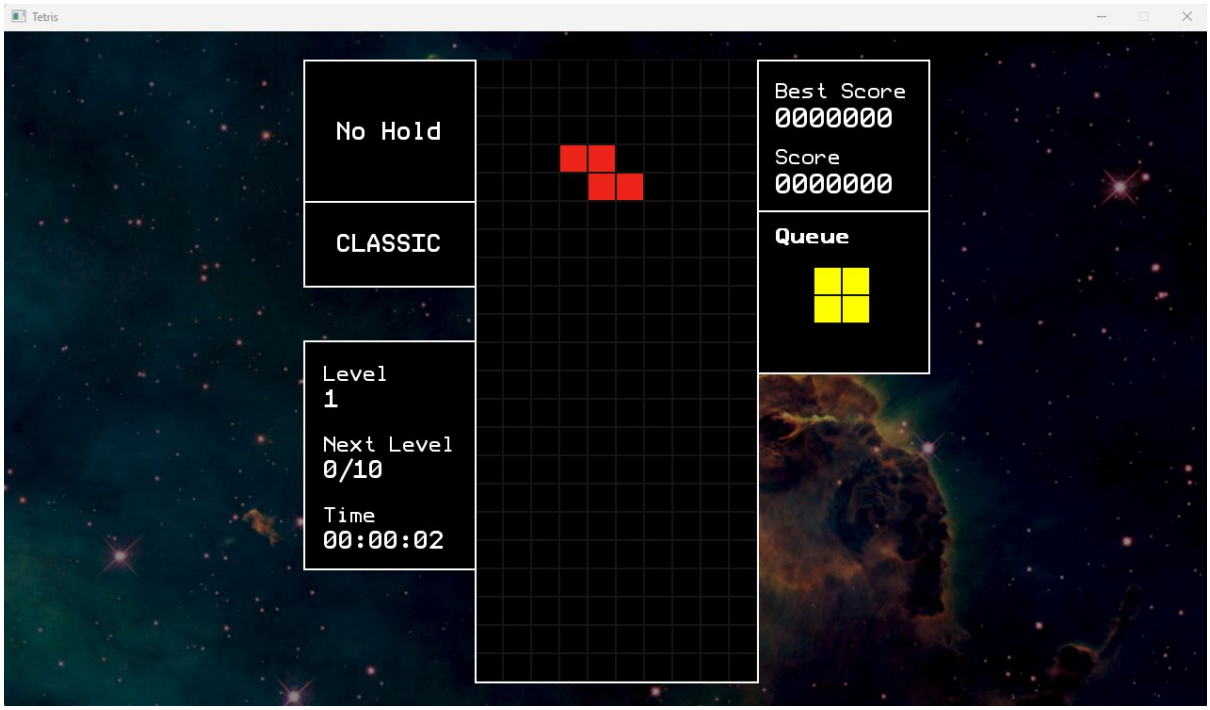

Figure 3: Classic Game Mode of Tetris Custom Program

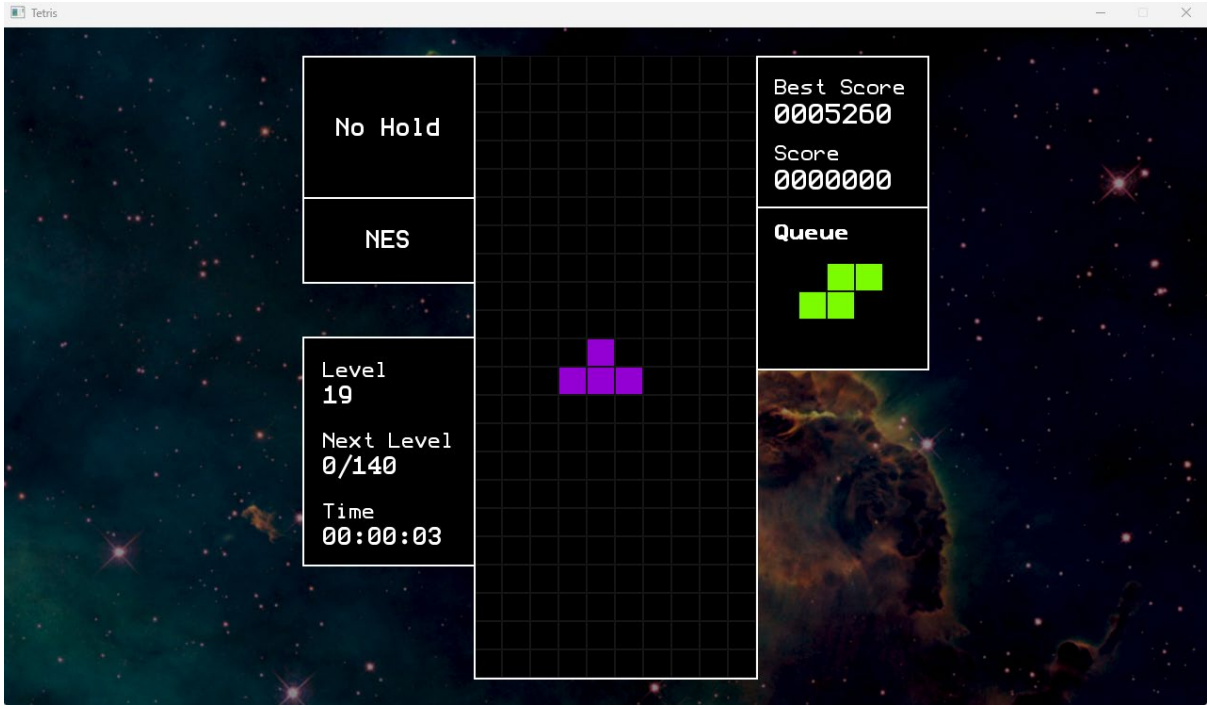

Figure 4: NES Game Mode of Tetris Custom Program

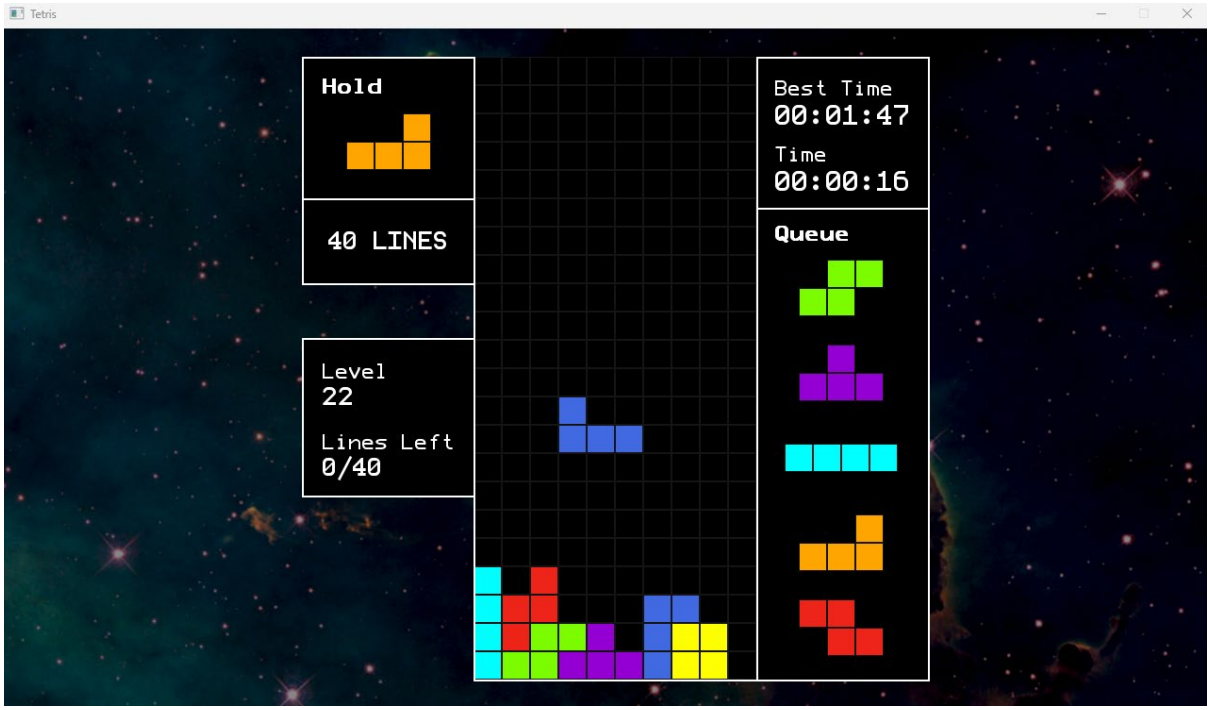

Figure 5: 40 Lines Game Mode of Tetris Custom Program

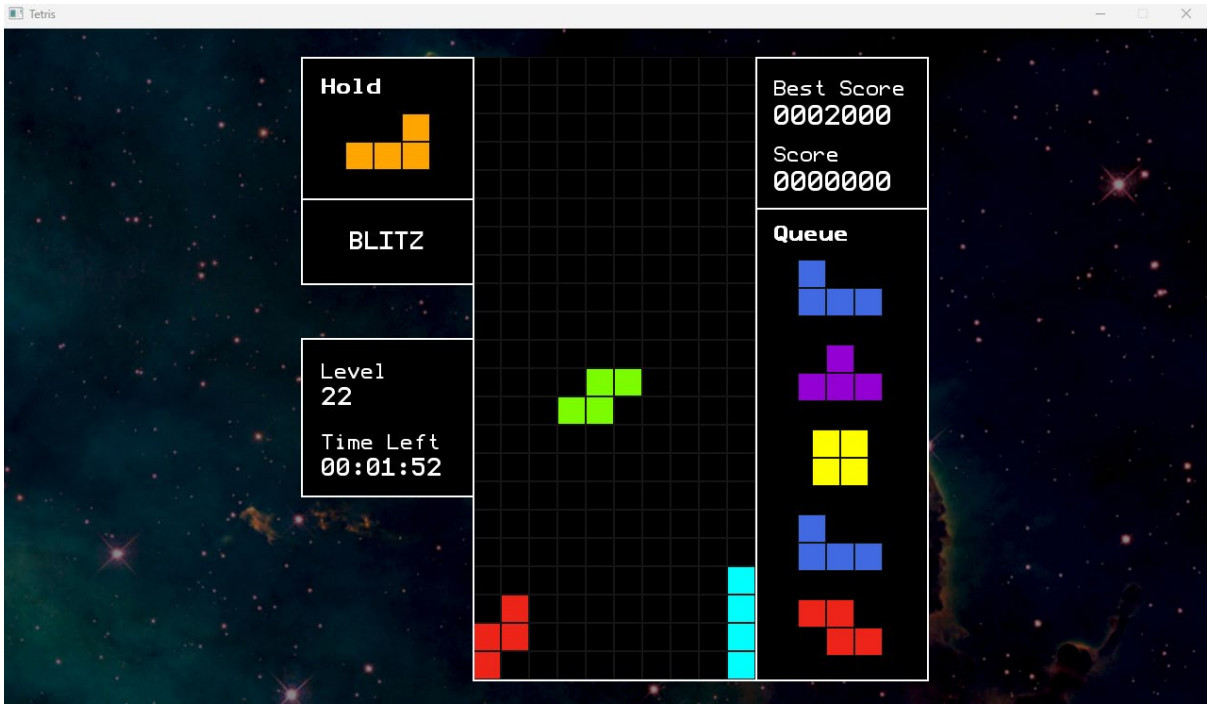

Figure 6: Blitz Game Mode of Tetris Custom Program

Figure 7: Pause Menu of Tetris Custom Program

Figure 8: Top 5 Times in Game Over of Tetris Custom Program (40 Lines Game Mode)

Figure 9: Top 5 Scores in Game Over of Tetris Custom Program (Blitz Game Mode)

Design Structures and Patterns in Tetris Custom Program:

The class diagram is provided in another document, but it can be downloaded for better resolution here:
https://drive.google.com/file/d/1j9ZdAgpNF1pIphT0vjdP8fgfKclRO5ci/view

1. Observer – Event Manager:

- Tetris Custom Program represents the MVC architecture, where the Observer pattern is used to implement an event-driven architecture, decoupling the Model, View, and Controller components. The Event Manager (Observer) acts as the "middle-man" to handle the communication between these components, where one component reacts and changes based on the game events posted by other components instead of directly depending on each other. Changes in one component do not require modifications in others since the listeners update based on the game events, making the system easier to maintain and extend. New listeners like Game Logic can also be added at runtime, by providing a queue of Attach, Detach, and Game Events to avoid concurrent problems.

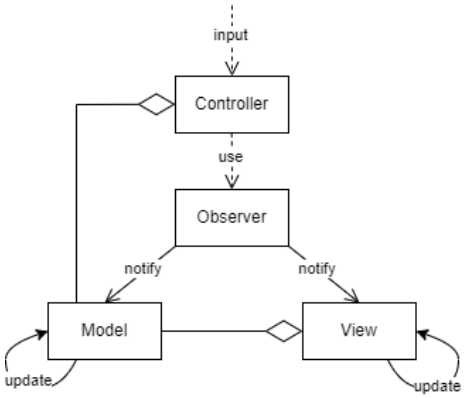

Figure 10: Tetris Initial MVC Architecture with Observer Pattern flow chart

- This “backbone” of the Tetris Custom Program structure enhances the maintainability, and scalability, allowing space for easy extension in the future. In fact, aside from Model, View, and Controller, some components like GameAudioProcessor and GameInputProcessor are not in the plan at the beginning, and they are easily added and extend the program functionalities, which are the testaments to my claim. A disadvantage of using this design pattern is that we have to know about the orders of listeners in the list or implement a queue attach, detach, or events (like what I did in the Tetris custom program) to avoid concurrent problems.

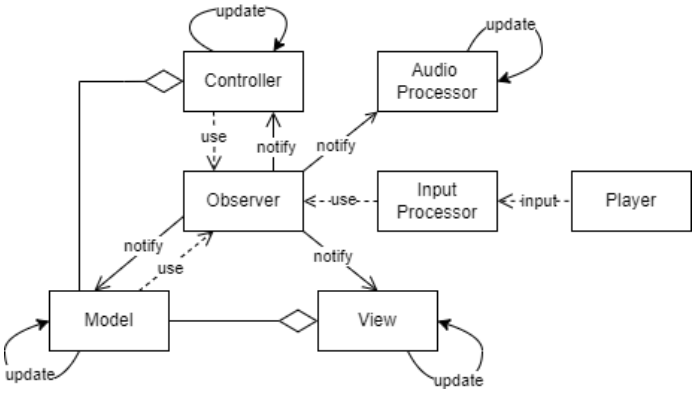

Figure 11: Tetris Final Overall Structure with Observer Pattern flow chart

2. State – Game State:

- The State design pattern is a behavioral design pattern that allows an object to change its behavior when its internal state changes. The Tetris game states are modeled using the State design pattern to handle different game states such as Main Menu, Help, Ingame, Pause Menu, and Game Over. It allows for clean separation of state-specific behavior and easy transitions between different game states. Each state has distinct behaviors and transitions that can be encapsulated within separate state classes: Menu State can transition to Help State or Ingame State, and Ingame State can transition to Pause Menu and Game Over State. Pause Menu and Game Over state can transition to Ingame State or Menu State depending on the game events.
- By implementing this design pattern, each state will have its information (game buttons), display (View), and controls (Controller) making the code more modular, maintainable, and extendable whenever we create a new state (for example a setting configuration state). This design pattern did not introduce any disadvantages to my Tetris custom program, and in fact it's extremely useful as I can create new states for extension very easily.

3. Decorator – Game Logic Decorator:

- The Decorator design pattern is a structural design pattern that allows behavior to be added to an object without affecting the behavior of other objects from the same class. In the Tetris Custom Program, the Decorator pattern is used to extend the functionalities of the GameLogic class, such as implementing different game-ending conditions (e.g., a Blitz game mode that ends after 2 minutes or a 40 Lines game mode that ends after clearing 40 lines) and adding the ability to hold Tetriminos.

-The Decorator pattern offers a flexible and modular approach to extending the functionalities of the GameLogic class in the Tetris game, as a game mode can attach 2 minutes game over condition with Hold Tetriminos ability, and another game mode may only attach 2 minutes game over condition, avoiding excessive subclasses (think about having m * n subclasses which can be reduced to m + n decorators) and code duplication (such as both Blitz and 40 Lines game mode can hold tetrimino).

4. Factory Methods – Game State Factory, Game Command Processor Factory:

The Factory Method design pattern is a creational pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. For example, the Factory Method pattern is used to create different game states with complex configurations in the Tetris Custom Program. Each state (Main Menu, Help, Ingame, Pause Menu, Game Over) has distinct information like game buttons. By using the Factory Method pattern, the creation logic for these states is encapsulated and separated, making it easier to manage and extend. In other words, the client (the logic such as Game Model) knows "what" state to transition to, but doesn't need to know "how" to create them (how to dependency inject via constructors to game state…) since it's the job of the factory.

5. Builder – Button Builder

The Builder design pattern is a creational pattern that allows for the step-by-step construction of complex objects. It provides a way to construct a complex object by specifying its type and content through a construction process. The Builder pattern is used to create immutable game buttons. These buttons have a range of properties such as size, color, border size, border color, and text, making the telescoping constructor pattern cumbersome and error-prone. It may become something like this:

```
public CircleButton(string name, int x, int y, int radius, int borderSize, Color color, Color borderColor)
public CircleButton(string name, int x, int y, int radius, Color color) : this(name, x, y, radius, 0, color, Color.White)
public CircleButton(string name, int x, int y, int radius) : this(name, x, y, radius, 0, Color.Black, Color.White)
public CircleButton(string name, int radius) : this(name, 0, 0, radius, 0, Color.Black, Color.White)
```

or

```
new CircleButton("Submit", 100, 200, 20, 2, Color.Cyan, Color.Black)
```

There's no way we can tell which value is it assigned to without looking/remembering the constructor order. Meanwhile, the Builder pattern streamlines the creation process, making it more readable and maintainable:

```
new CircleButtonBuilder("Submit").SetPosition(100, 200)
                                 .SetRadius(20)
                                 .SetColor(Color.Cyan)
                                 .SetBorderSize(2)
                                 .SetBorderColor(Color.Black)
                                 .Build(),
```

It provides a clear way to construct complex objects, ensures immutability, and allows for easy customization, making the custom program easier to manage and extend.

6. Command – Game Commands:

- In the Tetris Custom Program, the Command pattern is used to handle keyboard and mouse inputs. Each command corresponds to a specific game action, and the Event Manager (following the mentioned Observer pattern) posts different game events based on these commands. The Command pattern decouples the input handling from the game logic. The GameController does not need to know how each command is executed; it only needs to invoke the command.
- The Command design pattern enhances the flexibility, maintainability, and extensibility of the Tetris game's input handling mechanism. Decoupling the input handling from the game logic enables a clear separation of concerns, making the code easier to maintain and debug. The only disadvantage of using this in the Tetris custom program is that the program did not fully utilize the Command design pattern functionalities because the Command design pattern is very useful if the requested action is very complex, but in the case of the Tetris custom program, its only job is posting events to other listeners or enable SplashKit reading text.

7. Strategy – RandomStrategy, Extra/Queue/Score/HoldIngameDrawStrategy, …:

- The strategy pattern encapsulates each algorithm or behavior as an object, called the strategy, that implements a common interface or abstract class. In the Tetris Custom Program, the Strategy pattern is used to implement different queue random algorithms, and various display strategies for the queue, score, hold, extra display in the Ingame Drawer, and top five display in the Game Over Drawer. This setup ensures that different algorithms and display formats can be easily interchanged and extended, resulting in a more modular and easily extendable functionality. New strategies for queue randomization or display can be introduced without modifying the existing code. A disadvantage of using this design pattern in the Tetris custom program is that there is always a need to use a "simple static factory" to choose a specific strategy, which introduces a bunch of factory classes.

8. Singleton – Best Score Manager:
- The Singleton pattern is used for the BestScoreManager class. This class is responsible for loading the best scores from a JSON file and saving the player's score to the JSON file. The Singleton pattern ensures that there is only one instance of BestScoreManager throughout the game, coordinating all actions related to score management. The instance of BestScoreManager is created only when it is first needed, optimizing resource usage through lazy initialization, and encapsulating the score management logic.

9. Mediator – Game Logic
- The Mediator design pattern is a behavioral pattern that facilitates communication between different components of a system by introducing a mediator object. The GameLogic class in the Tetris custom program acts as a mediator that decouples the EventManager from Grid, Tetrimino, ScoreProcessor, QueueProcessor, LevelProcessor, and TickProcessor. The GameLogic class centralizes interactions between various components as it receives events from the EventManager and decides how to process them with the grid, Tetrimino, score, queue, level, and tick processors. Moreover, the components themselves don't need to directly depend on each other like Grid doesn't need to know anything about Tetrimino and vice versa. The mediator pattern also simplifies interactions between objects, reducing potential errors and dependencies. All communication is funneled through the mediator (GameLogic), making it easier to trace and debug. The only disadvantage of this design pattern is that the mediator class can quickly become a "god object" and this is true in the GameLogic class also as it's slightly larger than any classes in the custom program.

Key Classes and Methods of the Tetris Custom Program

Table 1: Cell details

Responsibility	Methods
Know its size, positions in the grid, its occupied boolean, and color.	
Know how to draw itself, draw the borders	Draw() DrawBorderLeft() DrawBorderRight() DrawBorderBottom() DrawBorder()

Table 2: Tetrimino details <<abstract>>

Responsibility	Methods
Know its shape, positions in the grid, rotation state, and ID.	

Know how to move, rotate themselves	Move() Rotate()
Tell the cell to draw itself	Draw(int x, int y)

Table 3: Grid details

Responsibility	Methods
Know its number of rows, and columns to initialize a matrix of Cells.	
Know if there's a full/empty row, move rows down when a row is cleared, if a requested cell is inside the grid and valid.	IsValid(int row, int col) IsInside(int row, int col) ClearFullRows() IsRowFull() MoveRowsDown()
Tell the cell to draw itself	Draw(int x, int y)

Table 4: GameLogic details

Responsibility	Methods	Notes
Know current Grid, current Tetrimino, hold Tetrimino, current Score Processor, current Queue Processor, current Level Processor, current Tick Processor		This class is a Mediator that decouples the Event Manager from dependencies like Grid, Tetrimino, Score Processor, Queue Processor, Level Processor, and Tick Processor as it helps centralize these dependencies' communication.
Know how to tell the dependencies to change itself based on the Game Events	Update(GameEvent e) MoveLeft() MoveRight() MoveDown() HardDrop() RotateCW() RotateCCW() LockTetrimino() ClearProcessing() TetriminoFits() Run()	

Tell Grid and Tetrimino to draw itself	Draw(int x, int y)	

Table 5: GameLogicDecorator details <<abstract>>

Responsibility	Methods
Know current Game Logic	
Know current exposed Game Logic methods	Update(GameEvent e) Run() Draw() GetNextTetrimino() Subclasses know how to extend these exposed methods functionalities

Table 6: ScoreProcessor details

Responsibility	Methods
Know its score	
Know how to add scores based on the number of lines cleared	Add(int numLineCleared)

Table 7: LevelProcessor details

Responsibility	Methods
Know its level, and number of cleared lines	
Know how to add a number of cleared lines and check for the next level requirements for the number of lines to clear to reach the next level.	Add(int numLineCleared) GetNextLevelClearedLine() NumberOfLineToNextLevel() GetLevelTicks() CheckLevel()

Table 8: TickProcessor details

Responsibility	Methods
Know its current timers	
Know how to start, resume, pause, or stop its timers	Start() Pause() Resume()

	Stop()

Table 9: QueueProcessor details

Responsibility	Methods
Know its current queue, and random strategy	
Know how to delegate random task to the RandomStrategy, if the queue is empty, and return tetriminos in the queue.	GetTetrimino() GetQueue(int amount) IsGeneratedNeeded()

Table 10: RandomStrategy details <<interface>>

Responsibility	Methods
Know how to generate random tetriminos	Generate() Subclasses know how to randomize tetriminoes

Table 11: GameModel details

Responsibility	Methods
Know its current game state, and game logic.	
Know what state to transition to, and post the game event to the Event Manager. Delegate update tasks to the current game state	Update(GameEvent e) TransitionTo(IGameStateFactory gameStateFactory)

Table 12: GameState details <<abstract>>

Responsibility	Methods
Know the current game buttons	
Know how to update based on game events	Update(GameEvent e) Subclasses know how to extend this method

Table 13: Button details <<abstract>>

Responsibility	Methods
Know its location, its name, its color, its border size, and border color.	
Know how to draw itself	Draw() DrawBorder() Subclasses know how to extend this method

Table 14: Button Builder details <<abstract>>

Responsibility	Methods
Know its location, its name, its color, its border size, and border color.	
Know how to construct immutable button based on current information step-by-step	SetPosition(int x, int y) SetColor(Color color) SetBorderSize(int size) SetBorderColor(Color color) Subclasses know how to extend this method

Table 15: GameInputProcessor details

Responsibility	Methods
Listen to the raw inputs, tell the event manager to post the KeyBoardInputEvent to the listeners	Update(GameEvent e) Listening(int code)

Table 16: GameAudioProcessor details

Responsibility	Methods
Know its current music list, and current music index in the list	
Play music and sound effects based on the game events	Update(GameEvent e) PlayMusic() PlaySoundEffect()

Table 17: IEventListener details <<interface>>

Responsibility	Methods
An interface for all listeners	Update() Their subclasses will have their implementation of the Update() method.

Table 18: EventManager details

Responsibility	Methods
Know their listeners	
Know how to add, remove listeners from the list, or from the queue	Attach (IEventListener) Detach (IEventListener) QueueAttach() QueueDetach() QueueEvent()
Know how to post/notify listeners of an event.	Post(GameEvent e)

Table 19: GameController details

Responsibility	Methods
Know game buttons in game model, current game command processor	
Know how to react on the game events, and delegate execution task to the current game command processor	Update(GameEvent e)

Table 20: GameCommandProcessor details

Responsibility	Methods
Know its current game commands list	
Know how to react to the game events messages	Execute(EventManager, string[] messages, gameButtons)

to invoke current game commands	KeyboardHandler(EventManager, keyCode) MouseHandler(EventManager, gameButtons, mousePosition)

Table 21: GameCommand details <<abstract>>

Responsibility	Methods
Know its key input, button pressed to be invoked	
Know how to post the game events	Execute(EventManager) Subclasses know how to extend this method

Table 22: GameView details

Responsibility	Methods
Know about game model and its current drawer	
Know how to react on the game events, and delegate drawing task to the current drawer	Update(GameEvent e) Draw() Initialize()

Table 23: Drawer details <<abstract>>

Responsibility	Methods
Know about the background color or image	
Know how to draw current game model	Draw(GameModel) Subclasses know how to extend this method.

Table 24: IExtraIngameDrawStrategy details <<interface>>

Responsibility	Methods
Know how to draw extra information for Ingame Drawer	DrawExtra(GameModel) Subclasses know how to extend this method.

Table 25: IHoldIngameDrawStrategy details <<interface>>

Responsibility	Methods

Know how to draw hold tetrimino for Ingame Drawer	DrawHold(GameModel) Subclasses know how to extend this method.

Table 26: IQueueIngameDrawStrategy details <<interface>>

Responsibility	Methods
Know how to draw queued tetrimino for Ingame Drawer	DrawQueue(GameModel) Subclasses know how to extend this method.

Table 27: RecordIngameDrawStrategy details <<abstract>>

Responsibility	Methods
Know how to draw best record for Ingame Drawer	DrawRecord(GameModel) Subclasses know how to extend this method.

Table 28: BestRecordGameOverDrawStrategy details <<abstract>>

Responsibility	Methods
Know how to draw top 5 records for GameOver Drawer	DrawBestRecord(GameModel) Subclasses know how to extend this method.

Table 29: All Factories details

Responsibility	Methods
Know how to create/construct a specific object, encapsulates the creation process from the logic.	Create()

Table 30: BestScoreManager details <<singleton>>

Responsibility	Methods
Know how to load, save, sort scores to JSON file. Check if the JSON file is existed	Save(gameLogic, playerName) LoadAll() Load(Mode mode) LoadBestScore() LoadBestTime() SortScores() CheckExistedJson() CreateDefaultJson()