Project Tetrov (Tetris)

BTP600 - Junlian Xiang

Kabeer Rashid, Arsalan Khalid, Andre Granovsky, and Justin Flowers

April. 12, 2015

Table of Contents

[Overview 3](#_Toc416690777)

[Major Functionality 3](#_Toc416690778)

[Design Patterns Chosen 4](#_Toc416690779)

[The Strategy Design Pattern 5](#_Toc416690780)

[State pattern 7](#_Toc416690781)

[Factory Design Pattern 9](#_Toc416690782)

[Observer Pattern 10](#_Toc416690783)

[Adapter Pattern 11](#_Toc416690784)

[Conclusion 12](#_Toc416690785)

# Overview

We decided to choose Tetris as our game to implement. Tetris is a game of placing various 4-blocked shapes into a fixed grid. The object of it is to eliminate as many rows as possible without filling the grid. We realized to accomplish this we would need a game framework to handle much of the work of creating and displaying objects. We additionally wanted to do this project in Java, as all of us have experience with it. After much research, we found the Slick2D Java game framework. We ended up creating this project in Java using the Slick2D BasicGame framework and using their Rectangle object.

# Major Functionality

The major functionality we wanted to implement is:

* To be able to create shapes with dynamic size and colour
* To be able to move and rotate shapes in the Tetris grid
* For shapes to not allow illegal movement in the Tetris grid
* For shapes to stack up and remember their position
* For complete rows to remove themselves and lower the above rows
* A pause state
* A lost state

This would be accomplished through implementing these major classes:

* GameState
  + A control of what state (play, lost, or pause) the player is in
* Block
  + Represents a single block in a shape
* Shape
  + Represents a combination of blocks in a shape
* Grid
  + Represents the current game board of shapes on the screen
* ShapeFactory
  + Creates shapes for the PlayState

# Design Patterns Chosen

Since we knew we needed this functionality, we chose these design patterns:

* Strategy
  + To create Shapes as compositions of Blocks
  + To create the Grid as a container of Blocks
  + To create additional helper classes
* State
  + To control how the player may interact with the game depending on if they’re in the Play, Pause, or Lose state
* Factory
  + To create the various Shapes necessary in the PlayState
* Observer
  + To alert the PlayState of a Shape Collision
* Adapter
  + To hook our State pattern into the Slick2D framework

# https://lh3.googleusercontent.com/TjSx0bJsiiIQSxh35tvRaTDdL0wN9sAUveJdGOrAQ1glhZffvDxxJ-3g9Rp8USnXEy3ShZ_GEgLMsvCNeSFBNo-ERYnwN_iPTwKaBGMRQtVfxoHAwfj0rlWxFsgtZG-i4qyCk-IThe Strategy Design Pattern

*Strategy allows us to create classes based on broken down component classes. Since the Shape class encapsulates the Block class, we can decouple logic such that methods in a Shape are interchangeable regardless of its initialized Blocks.*

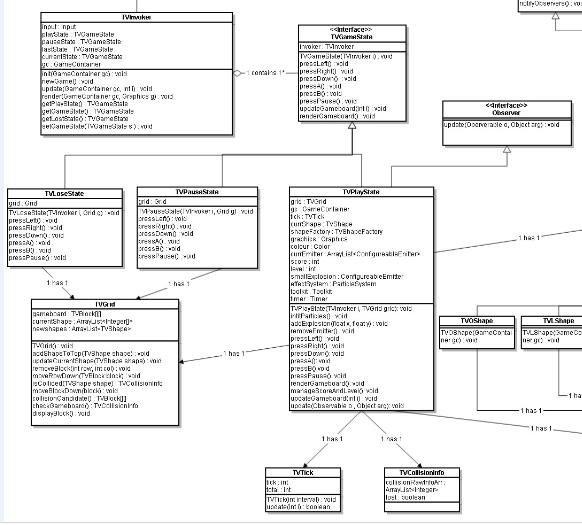
We chose to use the Strategy pattern for creating the Block and Shape classes because we felt it would make dynamic creation of Shapes with different block locations simpler than hard coding each one. It would also make collision simple as well, as Shape could simply ask its blocks if any had collided with the game board. Additionally, by defining an abstract Shape, we could decouple logic relating to handling rotation and collision from the Block class and have the Shape handle it all.

The first class we implemented using Strategy was the TVBlock class. This class extends the Slick2D Rectangle class and is responsible for moving and drawing a block. A TVBlock is composed of a Graphics object and a GameContainer object. The Graphics object allows it to draw itself to the current game canvas, and the GameContainer object allows it to dynamically create its width and height. Movement is based on the size of the block, such that movement always is in increments of the block’s height and width. To check collision, this class simply compares the x and y coordinates of itself vs a given block, returning true if the difference between the two is 0 (meaning no distance between them).

Next we designed and developed TVShape, the abstract base class which is responsible for containing and controlling a list of TVBlocks. This functionality includes shape rotation (clockwise and counterclockwise 90 degrees), shape movement (left, right, down), and collision handling. TVShape is composed of a 2D 4 by 4 TVBlock array which represents its current blocks. This class additionally extends Observable as it needs to notify the PlayState class of a collision. The Shape performs rotations by rotating its inner 2D array of blocks and moving them as necessary according to these changes. It performs movement by simply calling the move methods implemented in TVBlock on all its constituent blocks.

Strategy greatly helped in reducing the work in this section of the project, as it allowed us to abstract out all functionality from the concrete shape classes to the abstract shape classes. This allows the concrete shapes to only define where their blocks are located and what colour they are, making creation of different shapes simple. This additionally decouples movement and rotation logic from the concrete TVShape classes to the abstract class, making implementation of various unique concrete TVShape classes simple.

# State pattern

****

*The state design pattern is a behavioral software design pattern which attests to encapsulating varying behavior for the same routine based on an object’s state. All in all, state allows for a cleaner way for an object to dynamically change its method calls at runtime without resorting to large monolithic conditional statements, and is thus the heart of our game loop.*

We chose this pattern in our design to implement the “separation of concerns” in respective game states. We also chose it because it is a relatively simple system for changing how input is handled depending on state. Each state extends the TVGameState abstract superclass which in turn has a reference to the TVInvoker object. The TVInvoker object has state members to contain the current state and this game’s Play, Pause, and Win states.

.

As the game continues, the various states alert the TVInvoker to change its current state based on different conditions through the setGameState method. The concrete TVPlayState is the most important of these classes. It is responsible for the main game loop. This involves generating new shapes as necessary, containing and controlling the TVGrid class, drawing the current shape and blocks on screen, controlling the current shape, recording level and shape drop speed, and moving to the other two states.

TVGrid is a simple class which has the responsibility of knowing the current location of all blocks in relation to the standard Tetris grid (22 rows by 10 columns). It has public methods to add a shape to the top of TVGrid, update the current shape on TVGrid, and handle collision for an item on TVGrid. It is also responsible for removing complete rows and lowering incomplete rows above it. It does this by looking at fields in the TVShape class which contain the shape’s position relative to a 2D 22 by 10 TVBlock array. We use TVGrid to contain a record of all Shapes and Blocks on screen and to draw the current board state.

The TVShape classes notify the TVPlayState object when collision occurs. Once that happens the TVPlayState then updates the TVGrid about the current shape, checks the win/lose condition through the TVCollisionInfo class, and if they have lost sets the game state to TVLostState. The update(Observable o, Object arg) method is what actually handles passing this collision information to the grid from the Shape. It also increases level and score.

The key methods which every state must implement are related to handling input, resolving logic, and drawing to the screen. The various press methods (pressLeft, pressRight, etc) are used for handling user input. These methods should be called when the user presses the respective keyboard buttons for them. The updateGamebaord method is used in each state to handle logic based on current frame (which can be determined using the integer it takes in). Finally the renderGameboard method allows the state to actually draw something to the screen. The only differene between update and renderGameboard is when they’re called by TVInvoker: render at the beginning of a frame, update at the end. You’ll notice all of these methods are called in TVInvoker in different parts of the game loop, as they need to be called in different parts in order to be compatible with the Slick2D framework.

# https://lh3.googleusercontent.com/iEEsmhXzniPQWfqkF1qKeP5b0IsyqK6qFJVOpl5IDPWlI9KNMRoDAQdOLqUZhRNsvB8D01fLiUvdmNF_xgP7C1LFPA7RTsox-UrbZiKjy5VOYitfduiiKLZsXy1kizHXilvAMWAFactory Design Pattern

*The factory method pattern is a creational pattern which uses encapsulated methods to deal with the problem of creating objects without specifying the exact class of object that will be created. This is done by creating objects through calling a ‘factory’ method which utilizes the strategy method to build objects.*

In Tetrov, we utilized the Factory and Strategy design pattern to build different Tetris shapes at random. This allows the user to efficiently receive different shapes at runtime. The Factory design implementation occurs within TVShapeFactory.java, which contains a method known as public TVShape createShape(String shape). Notice that createShape accepts a string parameter which is utilized to determine what type of Tetris block to create. This is a common Factory approach which utilizes an if control structure to ensure that the correct class constructor is being called. In the UML diagram above, it shows that the TVShapeFactory class possess a 1 to 1 relationship with each type of object or Tetris block, and builds the block using its constructor and that constructor is triggered by the string given to createShape.

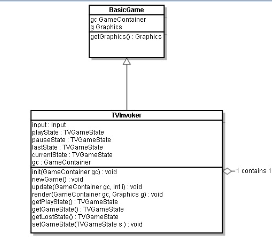
Once the correct object is created, it is returned for the client to use. Then, an RNG algorithm was developed to ensure that the client had access to a random shape as necessary. Note that the blocks come at random because of the original Tetris game design. The Random Number Generation algorithm is fairly simple, it utilizes the createShape method to create a static array containing each possible block. Then, the Java Random API is used to get a random index from the array, and then finally that index is returned to the PlayState.

# https://lh4.googleusercontent.com/W4rUfqn5ReJCP0w7P5E9K-_B-6dr6TT8SIdT_GaYZ5OhFo151j7CGVUAMRKqr9r1cef4tdE5o4efjhWxfEB9w-wsU8xbkKXbrepdngvubIfv6J_itWoSjqjWyiaZSVygY7gUOisObserver Pattern

*The observer pattern defines a one to many relationship. Because of this dependency we are able to notify all changes made in a Shape to the PlayState as necessary.*

In Tetrov we needed to update TVPlayState when a collision had occurred. Rather than have TVPlayState call the collision checking method each time a movement was made, we decided that TVShape should monitor its own movements and let TVPlayState know when it had collided. The design pattern we chose to perform this functionality was the Observer pattern. To make the Observer pattern work we defined TVShape as an Observable to notify the Observer TVPlayState when blocks were in collision. This collision check is performed in each movement command, and the TVShape uses setChanged and notifyObservers to alert TVPlayState of collision. TVPlayState then calls TVGrid and tells it that a collision has happened, passing it the updated TVShape object.

# Adapter Pattern



*The adapter design pattern allows the interface of an existing class to be used from another interface. It is often used to make existing classes work with others without modifying the source code, this is especially powerful within our main game loop.*

We needed to able to connect our design patterns to the Slick2D framework, and to do that we decided to use the Adapter pattern. Our TVInvoker is the class that connects the framework to our design patterns. It acts as the state machine controlling the various states, and uses Input listeners from Slick2D to connect to methods in the current state. For example:

if(input.isKeyPressed(Input.KEY\_A)) {

 currentState.pressLeft();

}

All key strokes received from Invoker are “routed” into currentState method calls. This connects our states to the framework in a concrete way. It additionally calls the methods implemented in each state in the correct part of Slick2D’s main loops, namely update and render. The only difference between these two methods is that one is called at the beginning of a frame (render) and the other is called at the end of a frame (update), meaning you want to draw and check logic in that order respectively. In that regards the TVInvoker is an object and class Adapter as it stores the class it needs to adapt and extends the class it is adapting to. This demonstrates how our code utilizes the Adaptor pattern to invoke methods on currentState, making two (otherwise incompatible) classes work together.

# Conclusion

In the end, using these 5 design patterns we were able to complete a Tetris clone. The hardest to implement were the Strategy and State patterns, as they had the most complex design involved around them. In a few scenarios behavior and data was misplaced as well as object ownership, yet during final stages of development we identified these issues and separated behavior within their corresponding class representations. All in all we learned a lot about the power of using design patterns and the issues associated with designing and implementing a complex project.