Project 1 report

Thursday 2:15 workshop
Group 13
Ryan Goh
Yuxin Miao
Danny Ngo

P1: concerns about current design

Our team has identified multiple concerns regarding the current design of Tetris Madness which hinders extension and violates GRASP design principles.

1. The various tetris piece classes (I, J, L etc) had many shared behaviours and lack of polymorphism which resulted in a lot of repeated code. The repetition of methods such as canRotate() and advance() makes the pieces difficult to edit.
   There is no superclass to extend when introducing the new pieces (P,Q and +) and would result in more duplicate code.

2. The Tetris class has too many responsibilities and is bloated. It spans nearly 400 lines and is responsible for generating UI elements, interacting with the library and handling game logic. The class has low cohesion.

3. Since there is low cohesion in the class Tetris, the variation point of different game difficulty levels (Easy/ Medium/ Madness) would not be well protected and this may lead to undesirable impact on UI and initialisation behaviour during implementation.

P2: refactoring current code

1. We refactored the duplicate code concerning the many tetris piece classes. Although I J L etc shared a lot of behaviours, they did not reuse the code. We did this by utilising the polymorphism pattern. We created a superclass called TetrisPiece, which contains 4 tetroBlocks, and all of the specific pieces extended. This way, each piece only needed to define their shape matrix (r[][]) in a helper function, createLocationMatrix(), called in the constructor and could use the superclass' methods This superclass also utilises the protected variation principle which will make it easier to apply the madness extensions as we will be able to reuse the functionality of TetrisPiece in the new pieces (P,Q,+) without affecting other elements of the game.

2. Next, we used GRASP principles as a guideline to refactor the Tetris class. The Tetris class in the old game is split into 3 separate more cohesive classes called TetrisInitialise, UIController, and Levels. These classes have more focused responsibilities which will result in better cohesion while keeping coupling low.

3. The class 'Levels' is a superclass which will be extended by the current 'easy' level. This utilises the Polymorphism pattern to allow various levels to share duplicate behaviour. This further acts as a protected variation which facilitates future extension of the 'medium' and 'madness' difficulty level and prevents core level behaviour independent of level (ie removeFilledLine) from getting affected undesirably.

4. Levels is also identified as the creator of the tetris piece because it closely uses and contains new tetris pieces as the game is being played. This aims to achieve low coupling and high cohesion.

5. The UIcontroller class is an indirection that handles UI components and gameplay logic to avoid direct coupling between classes. The UIcontroller class also acts as a controller and information expert, controlling the gameplay logic and assigning responsibilities to objects.

Proposed class	responsibility	justification
TetrisInitialise	Interacting with libraries and provided utilities including: JGameGrid game engine, JSwing gui toolkit, TetrisComponents and PropertiesLoader.	These components are considered external to the scope of our modifications. The principle of protected variation suggests we should isolate these aspects. This helps improve the stability of the system. This allows us to alter the gameplay without affecting the initialisation process.
UIController	Intermediary between the UI components and the gameplay logic.	This uses the grasp pattern of a controller. It provides an interface for Levels to send and receive information from GameGrid objects without directly accessing it. This provides an indirection which reduces coupling. This is an example of pure fabrication, as none of the classes in the domain are suited to handle the responsibility of communicating with the UI without reducing their cohesion, a purely artificial class is created to maintain cohesion

Levels	Core gameplay logic	This class would be provided the necessary information to run the gameplay logic. It would become the information expert to clear rows and the creator for new tetris pieces

method	New class	justification
Tetris()	Tetris	the original Tetris constructor remains in the initialiser and acts as the point of entry for Driver.
initProperties	Levels	All 3 of the new classes have access to properties in order to extract the parameters they require. Only Levels requires all properties and therefore this method is placed in Levels
Act()	Tetris	“heartbeat” function provided by the JGameGrid library. Tetris is responsible for passing messages received from the library to the UIController.
startBtnActionPerformed()	Tetris	Uses the libraries to initialise JGameGrid and Jswing objects.
getDelayTime	Tetris	Helper which provides the delay for startBtnActionPerformed()
start()	UIController	Called by Tetris whenever a new round begins (when the start button is hit or when the game begins) to perform round start tasks such as displaying the first block
newFallSpeed	UIController	Sets up a new fall speed. A responsibility of UI controller as it relates to the speed at which the UI renders
getSpeedMultiplier()	Levels	A helper to newFallSpeed. Levels is the expert which knows how much faster blocks should fall since Levels knows the difficulty
Moveblock()	UIController	Receives keys from UI and passes it to the respective tetrisPiece in Levels. Passing user input to game objects is the responsibility of a controller
Showscore()	UIController	Displays score onto the screen. Part of the controller’s responsibility is to prevent Level from directly accessing GameGrid when displaying scores
Gameover()	UIController	Displays the game over screen on the UI. Another task involving UI

createRandomTetrisBlock	Level	By the creator principle, since Level contains Piece and closely uses Piece, it should create the new Piece.
setCurrentTetrisBlock	Level	Part of the gameplay logic and therefore responsibility of Levels
removeFilledLine	Level	Part of the gameplay logic. As the creator and owner of tetrisPieces, Level is the information expert for checking filled lines.

Noteworthy design decisions

- Although Tetris creates creates the gameCallBack class as part of its role in initialising the utilities, Level has the necessary information that gameCallBack requires and communicates with gameCallBack during the operation of the game, gameCallBack is associated with Level rather than Tetris.
- The UIController holds the TetrisPiece blockPreview since that piece does not execute any game logic and is only for displaying on the preview UI screen. Level holds the TetrisPiece currentBlock since that piece is part of the gameplay. UIController passes any keyboard inputs to that piece via Level.

P3: design of Tetris Madness

Feature 1: difficulties

Tetris Madness involves 3 level difficulties each with slightly different behaviours. The polymorphism GRASP principle is relevant here, as we expect Level to have different behaviours depending on its type.
We make Level an abstract superclass, which is extended by 3 subclasses: Easy, Medium and Madness. Shared functionality such as removeFilledLine() and setCurrentTetrisBlock() remain in the superclass while level specific functionality will be overridden/ implemented by the subclass.

1. generateRandomBlockId(): a helper to createRandomTetrisBlock(), returns a set of valid pieces. For easy, it would only be the base pieces (ID 1 to 6); for medium and madness, this would include the 5 tetroBlock pieces as well (ID 7 to 9)

2. canRotate(): this returns a boolean for whether a piece can rotate. The default return value is True. In Madness, the method is overridden and returns False.

3. getFallSpeedMultiplier(): a function that returns the multiplier that increases the appropriate speed increase.

The fall speed increases were made possible by increasing the simulation period. Slowdown is only capable of increasing speed by 0.1 second increments (due to the library requiring integer parameters) and is not able to provide fine speed increases required by the specs.

The speed of a piece is calculated by:
1 move every (0.1 sec * slowDown * simulationPeriod/100) seconds
Or 100/(slowDown * simulationPeriod) moves per second
In order to increase the speed by 20%, simulation time is divided by 1.20
as 100/ (slowdown * simulationPeriod/1.20) = **1.20** * 100/(slowDown * simulationPeriod)
Verifying with System.nanoTime() finds that the player sees blocks falling roughly 20% faster with around ±3 milliseconds of error

For Easy, getFallSpeedMultiplier is set to 1
For Medium, it is set to 1/1.2 which causes the game to run 20% faster
For Madness, it chooses a random number between 1 and 1/2, which is up to twice as fast.

Feature 2: New pieces

The new pieces can extend our newly created tetrisPiece class as they are instances of tetris pieces and share all of their behaviours.
These new pieces contain 5 tetroBlocks instead of 4. Their location matrix (r[5][4]) contains the relative locations of 5 tetroBlocks instead of 4. The functionality is otherwise the same.

Feature 3: statistics logging

Since no class was available to perform the statistics logging responsibilities without compromising cohesion, we have used the Pure Fabrication pattern to create a new class StatisticsLogger, which will update statistics.txt.
Its responsibilities include reading and writing to the statistics file, updating the file every time a new piece is dropped or a new line is cleared and calculating the average score.
While we could have placed the file operation utilities in a separate class, we decided it was unnecessary at this stage as: we thought writing to other file types was not a probable extension in the near future and that it would be cheaper to rework the current design if it ever becomes necessary. For these reasons we decided that the costs of speculatively future proofing statistics logging would result in unnecessary complexity.

System Sequence Diagram

- The condition written is [can keep playing] rather than [nb!=0] because it would be easier to understand.