Abstract:

This report outlines a 3 week period summarising the thoughts and consequences of the choices made during this time. It describes the creation of a simple version of the game checkers, how we structured it and how we later on added advanced features to the game. Some of the advanced features include music, themes, full set of pieces, and a timer, just to name a few.
The report is based on a simple structure and begins with describing the delimitation of the program. From here the more advanced features is outlined and described. Hereafter the use of the MVC model is presented and is followed up by a thorough discussion of the implementation of the game.
In the end a conclusion is given based on our reflections of our success and/or lack of.

1. Delimitation:

The specifications of the basic part of the program as seen in the project description.

- The game is played on a quadratic board of size n x n.
- The n variable is supplied as a command line argument and n > 3 & n < 100.
- All the fields on the board is either gray or white.
- The top player has a red piece and the bottom player has a black piece.
- The red player begins the game.
- It is played by two human players - no AI is supplied.
- The pieces can only be moved either as player one - red player - one down and to the left or one down and to the right - and as player two - one up and to the left or one up and to the right.
- The simple version makes use of a simple GUI, which follows the MVC principles for seperation of responsibility in the code.

2. **Advanced program:**

2.1 Delimitation of the advanced program:

- The game is played on a quadratic board of size n x n.
- The n variable is supplied as a command line argument and n > 3 & n < 100.
- All the fields on the board is coloured according the the chosen theme.
- The top player has a piece with colour according to the chosen theme.
- The bottom player has a piece with colour according to the chosen theme.
- It is played by two human players - no AI is supplied.
- The pieces can be moved either as player one - red player - one down and to the left or one down and to the right - and as player two - one up and to the left or one up and to the right.
- The pieces can also be moved in a position to 'slay' each other. This is a jump over the opponents piece and cause this piece being jumped to disappear.
- The advanced version makes use of a multilayered GUI, which follows the MVC principles for seperation of responsibility in the code.
- The advanced game incorporates music on and off functionality.
- A timer is added and forces players to win inside of the given time frame. Each player has its own timer and the timer only counts down when its that players turn. If it reaches 0 that player loses the current game. The timer can be set in settings.
- Slain pieces are added to a third panel to show who is 'winning the game'. The amount of slain pieces for each player is added as a lying stack on each side of the board. Should the supplied cmdLineArgument > 15 then a count is added to better support big boards.
- Game can now be saved and resumed after exiting and reopening the game.
- 'King pieces' (dam) is added.
- An undo last move button is also added.
- A highscore that keeps track of previous games are added.

3. Design:

3.1 Model View Controller

The codebase is designed with the model view controller (MVC) structure in mind. It is divided into three packages - one for model, one for view and one for controller. All three packages has a main class that governs the communication from one to the other. Besides the main classes each packet holds other classes that helps with dividing the code into 'manageable' pieces and supports each of the 'functions' of the model, view or controller. The controller handles user input, sends it to the model, which in return, updates the view and lets the user see the result.

// En større beskrivelse af hvad der sker, når man trykker på fx en brik eller en knap.
// En tegning af vores "fortolkning" af MVC, som beskriver hvad der sker, når man trykker på en knap.

The MVC structure were chosen, because it allows for easy implementation and addition of new features. After finishing the simple version of the game each of the group members would then be allowed to work on it almost independently adding all the advanced features. We saw this as a great strength that helped push the decision, to use the MVC structure.

3.2 Design of the GUI

3.2.1 Basic Program

For the GUI part of the basic game we decided to keep it very simple, since it was a simple game. The main menu has only two buttons. You either click new game or exit. If you click the exit button the game ends but clicking the new game button takes you to a new screen that, on the left side, holds the board with its pieces but, on the right side, holds a text in the top and a single button in the bottom. Pressing this one button leads you to the main menu and allows you to restart the game or exit from there.

3.2.2 Advanced Program

4. **Implementation:**

4.1 Basic Program

4.1.1 General Layout

The GUI is created using the Java Swing class from the standard library of java. To help assist the initial creation of the GUI we set out using the WindowBuilder plugin for the Eclipse IDE. It was very helpful as an initial design tool, because it allowed for quick visualisation of the GUI we tried to create. After playing around with it for a while we settled on an initial popup game menu that has two buttons - a new game button and an exit button. We chose this structure due to the fact that we would later be building upon the basic game adding advanced features. One of these features would be a settings button. Doing a menu before the real game popped up allowed us to easily manipulate the settings of the game, before it began. Press the new game button and you are send off to where the magic happens - the game window.

4.1.2 The Board

The game is played on a quadratic board of size n x n. The used n is the supplied command line argument. To make sure the n is inside the bounds of the specification we make a check before utilising it. The n has to be in between 3 and 100. A simple check is made with (n < 3 && n > 100) - if so an error message is printed to the standard output, otherwise the variable is passed to the view as an argument for the constructor.

We tried different things when designing the board. Especially we asked ourselves what kinda underlying data structure would suit this kind of problem. Firstly we settled on an array structure utilising two 2D arrays - one for holding the information about which index in the array that would hold the different kind of pieces. A '1' would denote the red piece and a '-1' would denote the black. If a '0' was found in the array no pieces would be placed on that field on the board. The other array consisted of JPanels each the size of the board divided by the command line argument. To fill this array with pieces we would do a double for loop looping each element - if a '1' was found a red piece was added to the JPanel array but if a '-1' was found a black piece would then be added. If it was a '0' nothing would be added. We ended up abandoning this idea, because it was kind of slow when it came to lifting the pieces. The piece pressed would kind of 'hang' for a second before it got lifted and it gave the game a slow feeling. It was not satisfactory.

The other solution that we ended up settling with was a more mathematically based solution. Instead of using JPanels to make up the board we drew everything by extending our draw class with JComponent and utilising its graphics method. The board is drawn on a JPanel as if it were a coordinate system beginning in (0,0) and ending in (size of board, size of the board). Four squares are drawn at a time in two rows using a double for loop. We could have drawn one square at a time in one row but it was easier to do four at a time, because we didn't have to worry about changing colours or anything. The first square is drawn in (0,0) in a grey colour. The next square is drawn to the right of it, in white, by adding the square size to the x-value. The same thing happens for the one square right under the first square but here the square size is added to the y-value. This structure of drawing continues until the whole JPanel is a game board.

4.1.3 The pieces

To create and move the pieces we instantiate all our pieces with a colour and a point. This allows us to distinguish between the different kind of pieces and move the pieces with coordinates using the point class. All the pieces is held in an ArrayList allowing for use of the 'for each' syntax of the for loop - for (Pieces piece: pieceList) - looping over each element of the list checking to see if the piece is trying a legal move or not. The checks being made for each piece is;

1. If the piece is inside the board.
2. I the move is legal inside of the board - only slanted moves are allowed.

The moves are divided into two main checks where a jump boolean is checked - if the boolean is set to true then the opponents piece is ‘standing in front’ of the piece about to jump - and a double jump is initiated.

Advanced feature - pieces slain:

The pieces slain feature is a feature that shows the pieces that has been slain in a sort of 'lying pile', on the right side of the board. Each slain piece is moved to the side after they have been slain. If the amount of slain pieces exceeds the number twelve then a string is drawn to the right of the pile, displaying any piece slain 'above' twelve. This was done due to the fact that if the program would be initiated with a large command line argument then there would be a lack of space for slain pieces in the GUI.

Starting out making this feature it got clear that a new class was needed for drawing on the score panel. Before this feature was implemented all the methods related to drawing on the board was in a class called Draw. This did not allow for drawing on the scoring panel, because the Draw object, that was used, was only related to the panel that the board was drawn on. This panel was named panelOne and the score panel was named panelTwo.
A new draw object relating to the score side of the GUI was needed. Therefore the panelOne and panelTwo was renamed into drawOnBoardPanel and drawOnScorePanel. Afterwards the Draw class was split into two classes with the same name as the panels it serves - drawOnBoardPanel and drawOnScorePanel. To prevent too much overlap of the classes an interface was constructed. Both the classes implements it and are forced into implementing the drawPieces() and drawListOfPieces(). This was done to create a clearer structure of the classes responsible of drawing the board and score board.

To count how many pieces has been slain the MovePieces class had to variables added. One for each of the players counting how many time one piece had jumped over the other, respecting their colours of course. For each time a player jumps over the other player this variable - called slainPiecesPlayerOne or two - is incremented by one.

All the drawing of the slain pieces is done in the drawOnScorePanel and is handled by the drawSlainPieces(), the drawPieces(), the drawListOfPieces and the drawPlusPiecesString() method.

Advanced feature - music:

The music feature adds music to the game and also allows the players to start and stop the music in-game.

Initially we tried to make it work using .mp3 files but the javax.sound.sampled does not support .mp3 files (supported types can be seen here: **http://docs.oracle.com/javase/7/docs/api/javax/sound/sampled/AudioFileFormat.Type.html**).

When making this feature it quickly became evident that it was difficult to make the music start and stop if it was all made in the same method. For this reason the method was split into three instead of one. One to activateMusic(), one to stopMusic() and one to toggleMusic(). The last method checks to see if the music is playing, if so it calls the stopMusic() method.
Making the three methods allowed for easier installation of buttons and the use of just one button to turn it on and off. Also the music kept playing when pressing the back to menu button. This was fixed with a simple if statement that checks if the music is playing when the button is clicked and calls the toggleMusic() method.

Advanced feature - multiple pieces:

To enable the possibility of multiple pieces on each side of the board there had to be made certain changes to the game. Firstly, adding more pieces. When two payers begin playing the game pieces will be added on each side on the first three rows on every dark coloured square.

Adding the pieces to the screen was not the hard part but to adapt the rules and logic of movement proved to be quite difficult. It had to be made sure that the pieces could not be placed on top of another piece, that the code, before and after each move, searches for opportunities of slaying another piece - in both directions, but also that the right piece would be removed from the board when a piece is jumped over by the opponents piece, and that the player would only be able to move the piece ready to slay his opponent, because when you can jump over an opponents piece you must jump it, according to the standard rules of the game.
To support these operations there also had to be checks made for colours of the different pieces facing off each other and there had to be check if the two pieces facing off did not have any pieces behind them, because then the jump over the opponent would not be possible.

When trying to implement some of this into the basic game, problems occurred. Since the basic game only have to be aware of two pieces, and the advanced game has three lines of pieces it needs to keep track of, the logic behind moving gets very, very complicated and grew quite out of hand. It was managed by implementing it all into one big class - not the most ideal solution but it was the only solution at the time.
To fix some of the problems changes had to be made to the searchBoard() method, basically it was rewritten to fit the new and more requiring conditions of many pieces. It now contains two arraylists of possible pieces to jump. The reason for this is that before it was only making a search for pieces to jump on two pieces and now it does it search for every piece on the board.

Initially we had two different versions of the basic game. The current basic version is a merge of the two but initially the other basic version, that is no longer in play, used a 2D array to keep track of where there were pieces and where there were none. It did so by having the pieces set to the values 1 and -1, and every other entry of the array (i.e. square on the board) set to 0. When filling the board and drawing the pieces on the screen a double for-loop was constructed that looped over the 2D array, setting the backgrounds of JPanels that would then be shown on the screen as both the individual squares on the board and with appropriate pieces on top of them.
If we had to redo the whole thing we would have probably gone with the array version instead, because it would have been easier, simpler and cleaner, code wise, to implement the logic of the movement of all the pieces. We also mentioned some of this describing the initial phase in the description of the basic game.

Advanced feature - themes:

Themes' been added to try and overcome the dilemma between hardcoding and customisation. It was relatively easy to build the structure of the themes and adding them to the GUI. It proved to be rather difficult to make the actual themes appear when they were chosen from the menu and a new game was begun.
The problems that arose had to do with the way the methods were painting the pieces on the board. All methods assumed that the pieces, which used to be only two and only red and black, were red and black. This caused issues when moving to a new theme with new colours denoting the pieces, because it made it impossible to draw the pieces. Therefore the method drawPieces() in DrawBoardOnPanel had to be changed so that it compared Color objects instead of the previously used int (1 for red and 2 for black). And also - the pieces themselves had to be changed to use Color objects to denote its colours instead of the previously mentioned int. A getColor() method was also added.
Both in the MovePieces class and in DrawOnBoardPanel there were a lot of dependencies regarding the old int comparing system, when pieces were drawn, and pieces were moved.
These were all changed from the old structure of comparing values: 'piece.color == 1' - to the new and more general structure: 'piece.getColor().equals(pOneColor)'.

Advanced feature - timer:

Timers were added as a fun little supplement to a more fast paced game. One timer is added for each of the players and counts down from a previously set amount of time. When the turn switches the count starts counting down for the opposite player. The first player to reach 0 before the game has ended, has lost.

A new class containing the StopWatch was created. It hosts the methods needed for counting, setting the time, stopping and resetting it.

Firstly, as a temporary solution, we chose to implement the whole thing in CheckersModel. This was pretty straight forward, because we had access to all the variables we needed and had both the ActionListener that we needed for the timer, and the timer itself, available. When we afterwards wanted to move it all to a separate class it proved difficult to find the proper structure that would allow for using the setText() method of the Label used to show the timer.
We ended up settling on a structure that had the ActionListener and timer in the CheckersModel class yet keeping the real functionality of the StopWatch kept in the StopWatch class, it self. The setText() of the JLabel is now called by passing the menu object to the StopWatch class, when instantiating it. That did the trick and allowed for manipulating the JLabel from the StopWatch class using its builtin functionality, but at the same time having the ActionListener and timer doing its job in the CheckersModel class.