THE TRIANGLE PUZZLE

project document

Mira Keränen, 772040
Tietotekniikka/Computer science, vuosikurssi/class 1
23.04.2020

General description

The project is a triangle puzzle. It is a puzzle where the pieces have the shape of a triangle and are placed on a hexagon game board. Each piece contains three symbols, one symbol for each side. In the game, these symbols are letters from a to d. Letters can either be lowercase, a, b, c, or d, or uppercase, A, B, C or D. In the project plan, the letters were planned to be depicted as colorful arrows with the colors red, green, yellow and blue. However, the letters were noticed to be a clearer way to present the different pieces since the tip of an arrow and the end of an arrow would not be as noticeable as uppercase and lowercase letters. Pieces are placed on the board, and the goal is to match the letters on the sides of the pieces. In a correct solution, a lowercase letter is always positioned next to the same letter in uppercase.

The game was completed as the hard version. It has a graphical user interface. First, the program generates a solution. The solution consists of 24 different pieces. Therefore, two pieces can not have the exact same letters in the same order, taking into consideration that a player can rotate the pieces. A piece has six different positions. After the solution is found, the pieces are placed in a pile, and the program shuffles this pile.

A player can flip through the pile, rotate the pieces, position pieces on the board, move them on the board, and return them to the pile. The pile can be flipped both to the left and to the right. The pile can continuously be flipped to both directions, and, when the end is reached, the flipping continues from the beginning of the pile. This was changed from the plan since there was no reason to stop the flipping at the end of the pile. During the game, the board can contain a contradictory solution, and the program automatically recognizes a correct solution. This solution does not exactly have to be the generated one as long as all of the symbols match. In other words, the generated solution ensures the existence of a solution and generates the pieces, but several correct solutions might be found during a game. The plan for the project was to only accept the generated solution as a correct solution. It would, however, be quite unfair for a human player if some solution was found but it was not the generated solution and therefore was not accepted. This would considerably increase the level of difficulty, and almost make the game impossible. The game ends, when the player finds a correct solution.

There is a help functionality that reveals some of the pieces in the generated solution to facilitate the playing of the game. It removes all of the false pieces from the board and rotates the correct pieces to a correct position. It also reveals three pieces at a time randomly around the board. However, if the size of the pile is less than three, it only reveals one piece. This functionality was not in the plan or in the project description since the idea of giving hints was invented while testing and creating the game. It facilitates the game for players.

An unfinished game can be saved in a file and continued later. The game saves the situation on the board, the pile of pieces and the generated solution in the file. The original plan was to only save the board, but the pile and the solution were necessary for continuing a game. If the

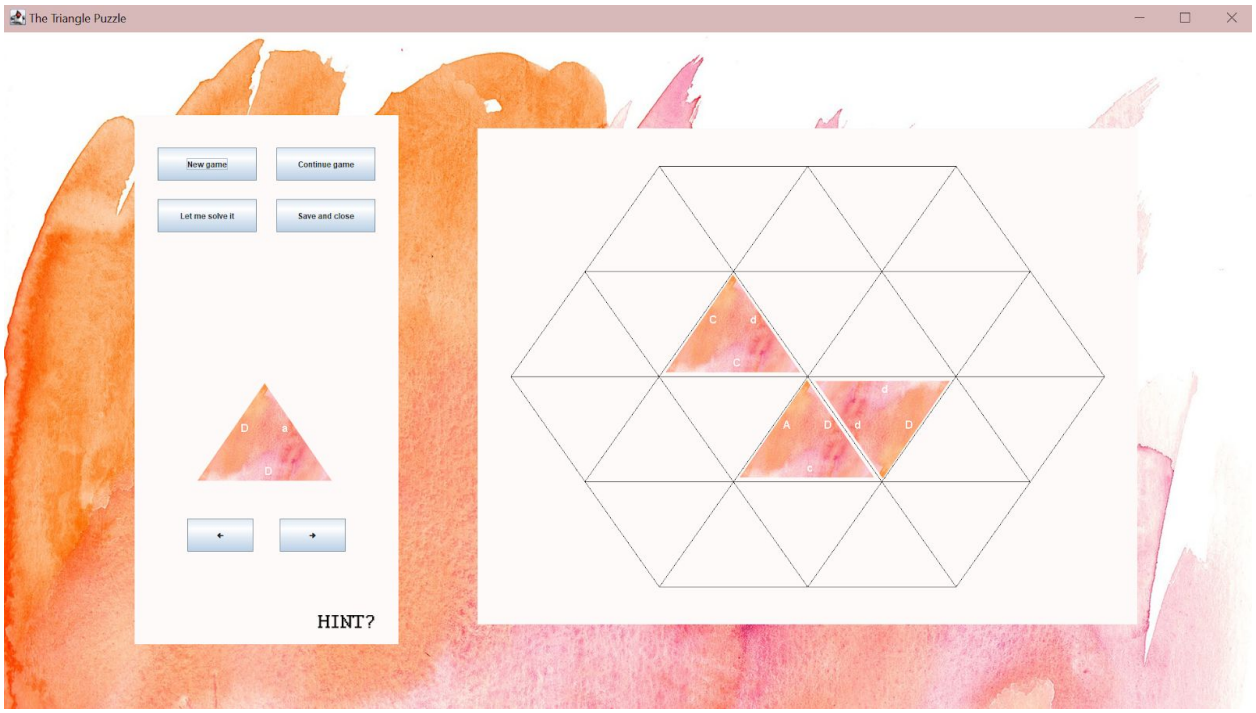
program has previously saved a game situation, the saved game can be loaded from the file and continued by a player.

The game has an intelligent algorithm. If the player can not solve the game by himself, he can let the computer solve the puzzle. In this case, the computer tries to find one correct solution. This solution is also one possible solution and not necessarily the generated solution as for human players.

User interface

The user interface is a graphical user interface. The UI contains one main frame. The main frame consists of two parts: the hexagon shaped board and the pile of puzzle pieces, including buttons.

When a player runs the program, a window opens up with an empty board frame, an empty pile and four buttons. These buttons are ‘New game’, ‘Continue game’, ‘Let me solve it’ and ‘Save and close’. In addition, the pile contains two buttons with arrows to the left and to the right. Underneath the pile there is a ‘HINT?’ button. The UI reacts to clicks and to the user’s mouse as well as to the buttons. Before any game is ongoing, the player can only press on the buttons ‘New game’ and ‘Continue game’.



Picture 1: The graphical user interface during a game.

If the player presses on the button ‘New game’, a new game starts. The other buttons can be pressed on except for the buttons ‘New game’ and ‘Continue game’ since a game can be started or continued only if a game is not ongoing.

The piece on top of the pile appears on the screen. By clicking on the piece with the mouse, the player can rotate the piece to all six positions of a piece. Pressing on the buttons under the pile flips through the pile to the right and to the left. By dragging a piece to the board, the player can add pieces on the board. A piece can only be added on the board in a correct position meaning

that if a place on the board is tip facing upwards the tip of the piece must as well be facing upwards. However, if a piece that is tried to be added on the board is not in the correct position the piece is automatically rotated to the closest suitable position when added on the board. This enables adding pieces to all places on the board. Pieces can be moved on the board or added back to the pile by dragging as well. They can not be dragged outside of the pile or the board. If a piece is dragged outside the board from the pile, it is added back to the end of the pile but added as the current piece, which is on top of the pile in the GUI. If a piece is dragged outside the board or pile from the board, it is added back on the board to the position it started from. The player can click on a piece on the board and rotate it. A piece always rotates two times at a time to ensure it is in a correct position.

The player can press on ‘HINT?’ to get a hint for the game. The hint always removes all false pieces from the board and rotates the correct pieces to a correct position. If the pile contains more than three pieces, it adds three new pieces correctly on the board and if the pile contains less than three pieces, only one piece is added. In case the pile is empty, no new pieces are added to the board. The hint always adds the correct pieces to the board according to the generated solution.

Pressing on the button ‘Let me solve it’ the computer tries to solve the game for the player and draws the answer on the board for the player. Instead, pressing on ‘Save and close’ saves the game situation in a file. Different execution situations of the program are visible in the appendix 1.

The button ‘Continue game’ starts the game as well but, instead of starting a new game, reads an unfinished game from the file. The pile and the solution remains the same and the pieces that were added on the board are also added on the board in the continued game. The current piece, the piece on top of the pile, may not be the same as when the game was saved. If a game is continued but the file is empty, a new game is started as when ‘New game’ is pressed.

During the game, the program does not print any messages for the player. It does not allow the player to drag a piece outside the board area or the pile. The game throws pop up windows, containing information for the player. When a game is started, information pops up about the game. Continuing the game pops up a window that tells the player that the game was continued. When the player solves the puzzle, the program pops up a window with the text “You solved the puzzle”. In case the program has solved the puzzle, it shows the correct solution and pops up a text “The puzzle was solved”. The pop-up windows are shown in the appendix 1.

Program structure

The game can be divided into four bigger parts: the game itself, the game board, the pile of pieces and the puzzle pieces. These are all represented as their own classes. Furthermore, the project includes an object FileOperations, for writing to and reading from the file, and another object PuzzleGUI, for the user interface.

The class Piece represents one piece in the game. To create a new instance of a piece, the piece takes four variables. The variables left, bottom and right are chars, denoting the three symbols of a piece. They are all vals and public variables. The variable pos, on the other hand, is an int and contains information of the position of a piece. The position is described with a letter from 1 to 6 which can change. Therefore, it is a var and a private variable. In addition, a piece has the variable coords to store the coordinates of the piece at each moment on the board. The coords contains the value None when the piece is not located on the board and therefore, does not have any coordinates. Coords is a private variable, as well, since the value needs to be changed. The private variables prevent from accidentally changing the values of them.

To get the position and coordinates of a piece, the class has methods position and getCoords that only return the values of the two variables. Method addCoords takes two ints, the row and the column of the piece location and adds these as the coordinates. The removeCoords, on the other hand, removes any coordinates a piece has. A piece has a method to rotate the piece. The method rotate changes the position of a piece to the right. For instance, if a piece has the position 1 and is rotated, it afterwards has the position 2. The method samePiece takes another piece and returns true if the two pieces are the same. In other words, it returns true if the left sides, right sides and bottom sides are the same.

In a puzzle the symbols of a piece must be A, a, B, b, C, c, D or d. However, the class piece does not prohibit creating pieces containing other symbols. Thus, the class could easily be reused. When first created, the symbols of a piece are immutable. The left, right and bottom represent the symbols on the left side, right side and bottom side respectively when the piece is in position 1. This enables ensuring that all of the pieces created are different, which would be more difficult if the three variables were mutable. Therefore, the piece has a method convertPos that returns the actual left side, right side and bottom side or top side depending on the position after the piece has been rotated. The method returns for chars and the fourth char represents the tip of the piece and is a char ‘O’.

The class PileOfPieces represents the pile in a game, where pieces are stored when not on the board. The pile is in a val variable and is a Buffer, containing Pieces. The buffer is in a private variable to not accidentally alter the pile. The amount of pieces in the pile varies between 0 and 24, but the pile does not prevent adding more pieces to be able to easily reuse the code.

To access the pile, it has a method piecePile that returns the pile as an immutable vector. The pile has methods isEmpty and size that return true if the pile is empty and the size of the pile respectively. The method contains takes a piece as a parameter and returns true if the pile already contains the piece. To access a piece on a certain index, the pile has the method pieceOnIndex that takes an int and returns the piece in the pile on that index in an option wrapper or None if the index does not exist. AddPiece, adds a piece as the last piece in the pile buffer, and takePiece removes a piece from the pile buffer. A piece can only be added to the pile if it does not already exist in it and removed if it exists in the pile. The pile can be entirely emptied with the method empty.

A pile always has a piece on top of it unless it is empty. The private variable current stores this piece. To access the piece, the pile has a method currentPiece that returns the current piece or None if the pile is empty. Piles have methods flipLeft and flipRight that flip through the pile and therefore change the current piece. The current piece can also be changed in case a piece is added to a pile and the piece becomes the current piece, or the current piece is removed from the pile and the next piece on the right becomes the new current piece.

The class Board represents the board in a game and keeps track of all of the pieces on the board. The board is an array, containing pieces. The board is a private variable to not accidentally change the values on it. The pieces on the board can be accessed by calling the method getBoard that returns a clone of the board. Pieces in the array are wrapped in an option to be able to have empty places on the board. In case a place is empty, it contains None.The amount of pieces varies between 0 to 24. In addition, the board has padding around it, containing pieces with symbols ‘x’. It has a method init to create the padding on the board. The padding facilitates adding pieces on the board since all of the pieces can be treated similarly. It also gives the board a shape of a square instead of an uneven shape. The board has a variable to store the places for game pieces and the position of pieces in an array. This array contains ints 0, 1 and 2. Integer 0 represents padding, 1 represents pieces with the tip facing upwards and 2 pieces with the tip facing downwards. The array containing the piece positions facilitates comparing whether a piece has the correct position for that piece or not.

A board has a method addPiece which adds a certain piece on the board array. The method takes a piece and two integers as parameters. The integers denote the place of the piece, the column and the row, on the board. The method automatically rotates a piece to the closest suitable position if the piece is already not in a suitable position. A method removePiece of a board removes the piece given as a parameter from the board array. A piece can only be removed if it already exists on the board. Both addPiece and removePiece return a boolean value. They return true if they have added or removed a piece successfully. The mtehod pieceOnCoords takes two integers as coordinates and returns the piece on the board in these coordinates. The returned piece is given in an option wrapper. If there is no piece or the coordinates are not on the board, it returns None.

The class board has methods isFull and isEmpty that return a boolean value. The value true is returned in isFull if the board is full and in isEmpty if the board is empty. To empty the board, remove all of the pieces from the board, the board has a method empty. A method rows returns the number of rows and columns number of columns. Together they return the size of the board.

The class game represents one ongoing game. A game contains one instance of the class board as the game board and one instance of pileOfPieces as the pile of pieces both stored in variables. It has also stored a solution in a private var sol and a boolean value of whether the game has been started in a var gameStarted. The default value of gameStarted is false. The solution can be accessed with a method solution. The class has a method generateSolution that generates the solution at the beginning of each game and stores it in this variable solution. It also shuffles and rotates all of the pieces and adds them to the pile of the game.

The class has a method startGame that starts the game by calling this method generateSolution and changes the value of gameStarted to true. The class game has a method continueGame for reading a saved game situation from the file. It takes a board array, a pile buffer and a solution array and updates the board, pile and solution of the game to match the given parameters. In case the file read is empty and the solution, board and pile are empty, this method as well calls generateSolution and starts a new game.

Game has a method solutionFound that returns true when a solution on the board is correct. The solution does not have to be the exact same solution as stored in the variable sol as long as the symbols on the pieces match correctly. Thus, the generated solution only ensures that a solution exists, but other solutions are accepted as well if they are found. The method gameOver calls the method solutionFound and returns true if a game has been started and a solution found. When the game is over, the game has a method endGame that ends the ongoing game by emptying the pile, board and solution. The method getHelp helps the player by removing all false pieces from the board, rotating the correct pieces to the correct positions and adding three pieces correctly on the board. In case the pile is empty, no pieces are added and, if the pile contains less than three pieces, only one piece is added. The class has a method solveGame that tries to solve the game. It first empties the board and then tries pieces on the board until it finds one solution. The method does not either have to find the generated solution as long as it finds some solution.

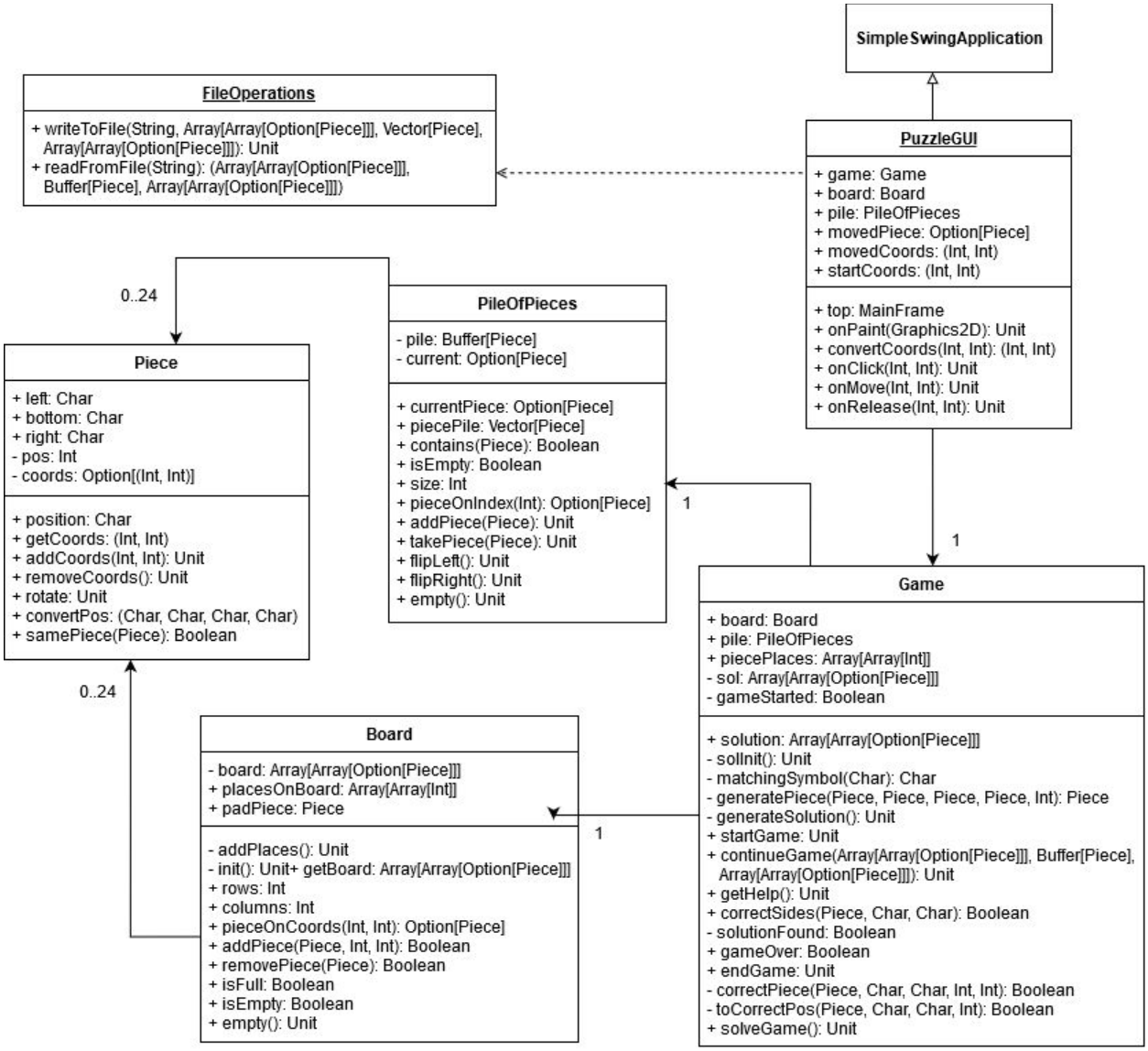
The user interface is a graphical user interface and Scalas Swing libraries are used in it. It is an object that extends the Scalas SimpleSwingApplication and uses its methods. The puzzleGUI, the user interface, includes a method top that contains the MainFrame of the game. The main frame is created as an instance of a class MainFrame. The class extends a boxpanel and contains all of the buttons and contents for the user interface. It also listens to the buttons and mouse moves and clicks and then reacts to them.

The puzzleGUI has a variable game where the game for the user interface is stored. A GUI can only contain one game. It has a method onPaint that paints the pictures for the interface. The

method paints the background, the pieces in the pile and on the board, and the frame for the board. It also has methods onClick, onMove and onRelease that are called when a mouse is clicked, moved or released respectively. OnClick rotates the pieces correctly in the pile or on the board and onMove and onRelease remove pieces from the pile and add to the correct places on the board. A method convertCoords converts the coordinates in pixels from the main frame to the ones on the board.

The project contains another object fileOperations for writing a game situation to a file and reading a saved game situation from the file. The method writeToFile writes and the method readFromFile reads. WriteToFile takes as parameters a board array, a vector of pieces and a solution array. It also takes a string containing the name of the file to write in. If a file with the given name does not exist, the method creates a new file. ReadFromFile only takes as a parameter the name of a file. If the file does not exist, it returns an exception. The method returns a tuple, containing an array for the board, a pile buffer and an array for the solution. The puzzleGUI utilises the object fileOperations to write to and read from file a game situation.

There are several other options when dividing the program into classes. For example, the pileOfPieces would not necessarily have to be its own class. The buffer for the pile could be in the class board or game. Having a separate class was chosen because the pile needs several methods of its own. In a separate class they are conveniently separate from other methods and have access to the pile buffer. The same applies to the class board. The board array could have been in the class Game, but, by creating a new class, it can have its own methods that have access to the array and are only used for the board. A class piece was created because pieces have many parameters. With a new class, the piece can be used as a type for the pile and the board and we have access to the parameters. If it was not a class, it would have been necessary to present the information some other way as in tuples, which would complicate handling the pieces. FileOperations was chosen to be its own object since the methods writeToFile and readFromFile are separate from other classes. They could have been added to the GUI but it would have made the GUI longer and the method would have been separate from the other methods in the GUI. The fileOperations is an object since no new instances need to be created from it.



Picture 2: An UML chart of the classes and objects in the program. The class MainFrame in the PuzzleGUI is not depicted.

Algorithms

The game utilises two essential algorithms. One algorithm creates a correct solution and shuffles pieces in a pile at the beginning of a game. Another algorithm tries to solve the puzzle without any given information and, thus, is apparently intelligent .

The algorithm for generating a solution tries different combinations of pieces until it finds a solution where all of the 24 pieces are different. The algorithm starts with a solution array full of padding pieces, pieces with symbol ‘x’ on their sides. For each piece, it matches the symbols on the sides of a new piece by taking the correct sides of the three surrounding pieces and returns the matching symbols. In other words, it matches A to a, B to b, C to c and D to d and vice versa. In case the side next to this piece has the symbol ‘x’ it randomly chooses one of the previous eight letters. It continues matching the sides until it finds a new piece that has not already been generated.

For the first piece, all of the surrounding sides contain the symbol ‘x’, meaning the program randomly generates this piece and adds it to the solution. For the other pieces in the first row, one side is already fixed. Thus, the program randomly chooses the two other sides of the pieces. Either one or two sides are fixed for pieces in the next row, and the program randomly chooses the rest of the sides. This action continues until a solution is found. This algorithm is efficient enough since each piece has three sides and each side can have eight different letters. There should exist 512 different pieces, and one game only needs 24 of these. For this program, the algorithm functions properly and is quite simple, which is why it was chosen.

After a solution is found, all of the pieces are shuffled and randomly rotated with a random generator. The pieces are stored in a buffer. The random generator shuffles the buffer, rotates all of the pieces, and removes the first piece from the buffer to add it to the pile of the game. This is continued until the end of the buffer is reached and all of the pieces are added to the pile. This could also have been done by generating two numbers with the random generator and switching the place of the pieces in the buffer on these indices. First, numbers from 1 to 24 could have been chosen, then from 2 to 24 and so on. Choosing two numbers would have been more complicated, since the random generator already contains a method for shuffling a collection.

In pseudo-code the algorithm would look like this:

```
for each row i and column j in the solution {
  do {
    if (the position solution(i)(j) does not contain padding) {
      generate a new piece with matching symbol
      update variable piece to be the generated piece
    }
```

```
  } while (this piece in variable piece already exists)

  if (the position solution(i)(j) does not contain padding) {
    add the piece to the solution
    add coordinates to the piece
    add the piece to a buffer containing all generated pieces
  }
}

for 24 pieces {
  shuffle the buffer of pieces
  rotate each piece
  remove the first piece from the buffer and add it to the pile
}
```

The intelligent algorithm functions by trying different pieces on the board. For each piece, it compares the left side and upper side of the piece in its converted position to the sides of the pieces on the left and above this piece. It matches symbols as the algorithm for generating the solution except for the case of the symbol ‘x’. If one of the symbols the piece is compared to is ‘x’, the algorithm accepts all eight letters. It also considers the position of the place on the board. Therefore, it rotates the piece first if it is not in a correct position. Then, if the symbols do not match when compared, it rotates the piece to the next suitable position and compares the piece again. It can rotate a piece a third time, and if the symbols still do not match, the piece is not suitable and the algorithm takes another piece.

The algorithm searches through the pile of pieces in order. It always starts from the index 0 and grows the index by one if the piece on this index is not suitable. It stores all of the used indices in a buffer and before every piece checks that the piece has not already been used. It also stores the last index not used from the pile. By this means, the algorithm can deduce when there are no more possible pieces left in the pile, which is when the piece on the last index is not suitable, and return one step back.

All of the pieces are stored in the pile of pieces. The program chooses the first piece from the pile. It compares it to the pieces next to it, and adds it on the board, since the symbols next to it are ‘x’:s and the first piece always fits. Then, it takes the next free piece and compares this piece. If it fits, the algorithm adds it on the board and continues to the next free piece. Together with the added piece itself, it adds the information of the pile index for the piece to not try this piece in the same place again. When it runs into a situation where none of the pieces in the pile fit in the next position, the algorithm returns one step backwards. In other words, it removes the last piece added and continues finding another suitable piece in its place. The algorithm continues this action until the board is full. Meaning, it continues until the program is at the 24th piece, has found all of the other pieces and the 24th piece matches the previous pieces. In this case, the correct solution is found.

The intelligent algorithm functions well but is not perfect. In the worst cases, the algorithm is quite slow and it may take several seconds to solve the puzzle. The algorithm could be further developed, if there was more time. An implementation using recursion could be one other option for the algorithm. The ideal situation would be an algorithm, that would not have to go through every single piece for every place in the worst case.

The algorithm would look like this in pseudo-code:

```
store the symbols on the left and above of this piece, the position, the index in pile
and the index in stack, the current piece, and the last index in the pile in
variables.

while (size of the piece stack is less than 24) {
  if (the pile index is not yet used) {
    try to match the piece in the next place
    if (the piece matches) {
      add the piece to the piece stack
      add the pile index to the used indices
      if (the pile index is the last index) {
        update the last index to match the last unused index
      }
      update the values of the variables
    } else {
      if (the pile index is the last index) {
        return one step back
        update the values of variables
      } else {
        grow the pile index by one
        update the current to be the new piece on the pile index
      }
    }
  } else {
    grow the pile index by one
    update the current to be the new piece on the pile index
  }
}

for 24 pieces {
  add the piece on the board
  take the piece from the pile
}
```

Data structures

This game uses Scalas data structures. The game board and the solution or the game are stored in an array because the board never changes its size. Array, which size is immutable, suits well as the board. The elements are still changeable, which is obligatory when pieces are added and removed from the board. A vector is completely immutable, which would have not enabled changing pieces. Another option would have been a buffer, but it is mutable, increasing the possibility of accidentally changing the size of the board. Since we do not have to change the size of the board, an array is perhaps the best option. In addition, the board needs to be two dimensional, and arrays are typically used as two dimensional tables, meaning the class contains methods for handling these tables.

The pile is stored in a buffer. The size of the pile changes and elements are added and removed from the data structure, which is why a mutable buffer suits well. For example, an immutable vector would not be convenient since it is immutable and adding elements is more complicated. A list could have functioned as this data structure, but, in a list new elements are added and removed on top of the list. It must be allowed to handle all of the pieces in the pile, not just the one on top since the game wants to access both the following element and the preceding element in the pile. With a buffer, indices can be used, which is not possible in a list. However, a vector is used when accessing the pile outside the class pileOfPieces to avoid accidental changes in the pile buffer.

For the intelligent algorithm, a buffer is also used to store the pieces on the board while solving the puzzle, since indices for the pieces are needed when adding pieces to the board. In a buffer it is possible to add pieces at the end of the collection and remove them from the end, which must be possible. For example, a Stack would have been another option that could have been used as the data structure for the pieces on the board since in a Stack it is easy to add and remove the pieces only on top of it. However, in a stack, elements do not have indices, which would have made handling the pieces more complicated.

The board array uses the data structure Option[Piece]. The board can be empty and there can be places with no piece in it. With an option structure, None can be stored in those places. Otherwise, there would have to be another piece to demonstrate an empty spot, making handling the board more complicated. The Option is also used to store the coordinates of a piece in its coords variable, since a piece only has coordinates when it is on the board. When the piece does not have any coordinates, None is updated as the value of the coords. If option was not used, we would have to come up with some other way to express a piece being in the pile, such as a pair (0, 0), which could be more complicated.

Files and Internet access

The program saves each situation of the game in a text file (.txt). The information for a game starts with a header #GAME. The game file has three subheaders: #BOARD that stores the information of the situation on the board, #PILE that stores information of the pieces in the pile, and #SOLUTION that stores the solution. In case the file contained some other headers, these headers and their information would be ignored.

The information is presented separately for each piece. For the situation on the board, the pile and the solution, the information for a piece is presented in the same way. Pieces in the game have three sides that can each have a different letter from A to D, and a piece is identified by these three sides. These letters can either be uppercase, A, B, C, D, or lowercase, a, b, c, d. Each uppercase letter pairs up with a lowercase letter, for example, A - a and C - c. In the file format the game is saved in, the symbols on the sides of pieces are depicted with these eight characters as in the game. The sides of a piece are saved in the file in order: left side, bottom side and right side. The sides saved in the file are the three variables left, bottom and right of a piece. For example, if a piece had the symbol A on its left side, D on its bottom side and a on its right side, the file would contain the string ADa. In addition, the file contains information of the position of a piece. This position is the value of the variable pos of a piece and is an integer from 1 to 6. For example, if the previous piece was in position 3, the file would contain the string ADa3

Each piece in a pile is represented by a string of the same form as the previous string and each piece is written on a separate row. However, in the situation on the board and in the solution, the pieces have coordinates represented in the string as well. Coordinates are described as a pair of numbers. The first number shows the row of the piece and ranges from 1 to 4. The second number shows the column of a piece and ranges from 1 to 7. For instance, if the previous piece was on the second row and in the first column, the entire string would be ADa321. If the row number has a value of 1 or 4, the number for the column only ranges from 1 to 5. One row in the file contains the information of pieces on the same row on the board or in the solution.

For an ongoing game situation where the solution is the same as in the example game in the assignment, the file would look like this:

```
#GAME
```

```
#BOARD
DCa111 AdB313 DBb414
aba226
bAa532 Dcb633 CDA235 ddd637
cAD441 CBD142

#PILE
aBd2
add3
AAA5
caB5
CAa4
cDD6
cdc6
bDb1
BDD3
CaA4
dAB5
dcA2
BCd1
baA6

#SOLUTION
DCa111 AAA412 aBd113 CBD414 cdc115
dAB121 Dcb422 dcA123 aba424 AdB125 bDb426 BDD127
CaA431 cDD132 BCd433 bAa134 CDA435 cAD136 add437
ddd441 DBb142 caB443 CAa144 baA445
```

The program uses some pictures that are saved as picture files (.png or .jpg) in the root directory. One file contains one picture. There are three different pictures used and saved in files: the images for a triangle with the tip facing upwards and tip facing downwards, and the background image for the game. The image for the hint-button in the GUI is also an image.

Testing

The project was approximately tested as planned, and the program passed all of the tests. There were no essential holes in the tests except that some methods were not yet included in the plan. All of the planned unit tests were executed. Some new methods, that were not in the plan, were implemented, and some new unit tests were made for them. For example the methods flipLeft and flipRight in the class pileOfPieces were added and tested. Unit tests were used for all of the methods and classes that could easily be tested with them. Other classes were tested in other ways.

The graphical user interface and the object were tested using other methods than unit tests. For the fileOperations, example data was created. The method writeToFile was given some array of the board, some pile and some solution. The method was called, and it was verified that the contents of a file are correct. Similarly, after some situation was written in a file, the readFromFile was called and the file was read. The returned arrays and the buffer were then compared and verified that the values are correct. For the GUI methods were tested by running the program and pressing on buttons or using the mouse. Methods were called this way and the results were verified that they are correct. For example the method onClick was tested by clicking on the piece in the pile and pieces on the board, and the methods onMove and onRelease by dragging a piece and releasing it either on the board or on the pile. From the GUI it is then easy to notice whether the piece has been added to the board or to the pile, or it has rotated correctly. When the GUI was finished, almost all of the methods were tested in the GUI in addition to unit tests to ensure correct functionality of the game.

Unit tests were used for the most crucial methods in classes piece, pileOfPieces, board and game. In the class piece, the most important method is the method rotate. The method should change the position of the piece to another position. It should return Unit but should have affected the state of the piece. For instance, if the position of a piece is originally 3 and the rotate method is called, the position should now be 4, and if the position is 6 and rotate is called, the position should be 1. The method was tested by creating an instance of the class piece and then calling the rotate method for it. After the method call, the actual value of the position can be compared to the expected value. If the previous example with position 3 was tested, the expected value after the method call would be 4. Similarly methods addCoords, removeCoords, convertPos and samePiece were tested by creating instances of pieces, calling methods for them and comparing the returned values or values of variables to the expected ones.

For all of the unit tests, the tested situations were tried to be chosen as diverse as possible. Most tests contain three cases, the basic case where the method is called as intended, the false case where there is something wrong with the method call, and the borderline case where the method is called with some borderline values. For example, a borderline case would be when the first piece is added to the board.

For class board methods addPiece and removePiece were tested with unit tests. For addPiece, three different cases were tested. In the basic case where a piece is added to an empty spot on the board, the method should add the piece correctly. Instances of the board and pieces were created and the value on the board was compared to the pieces. These values should be the same. Another case is when a piece is added outside the board. In this case, the piece should not be added on the board. In the third case a piece is tried to be added on the board in a place that already contains a piece, and the piece should not be added. Similarly, for removePiece cases where a piece on the board is removed and where a piece that is not on the board is removed. In the second case nothing should happen and the method should return false. For addPiece and removePiece, it was tested that the methods add and remove coordinates for the pieces correctly by comparing the value of the piece.getCoords to the expected one.

In pileOfPieces the method pieceOnIndex was tested by unit tests. An instance of the pile was created and some pieces were added to the pile. When given an integer the method should return the correct piece from the pile or None if the index does not exist. The returned value was compared to the expected value. Methods addPiece and takePiece were tested. When addPiece is called, the method should add new pieces as the last piece. The value on the last index of the pile buffer was tested and compared. In the other case for addPiece, the method should not add pieces to the pile that already exist in the pile. Take piece was tested similarly as addPiece, but it should remove pieces correctly from the pile and should not remove pieces that are not in the pile. Flipping through the pile was tested by comparing the value of the variable current. When flipLeft or flipRight are called, the current should be updated correctly. It should also be correct when a piece is added to an empty pile or a piece in the current is removed.

For the class game, the methods for generating a solution and solving the game were tested by unit tests. An instance of the game was created. The method startGame that calls the method generateSolution was called. The unit test compares each value in the solution array and ensures each place in the solution contains a piece. It compares the symbols on the sides of pieces, and this part of the test passes if all of these symbols match. The third case it tests is that each piece in the pile is different. The test for solveGame is similar to the previous test, but it only compares the symbols on the pieces and passes if these symbols match.

In addition to unit tests, the program was tested during the implementation. The methods were called, and the values stored in variables or the returned values were printed and ensured they are correct. Also the GUI was used for testing after it was finished to easily see the results of calling different methods.

Known bugs and missing features

The project has been tested and should not have any significant bugs. All of the features for the hard level have been implemented. Thus, the project does not contain missing features.

While testing and playing the game, some of the correctly positioned pieces sometimes rotate to another position when added on the board in the GUI. For instance, the place on the board requires a piece with the tip facing up, and the piece is in position 1, which is a correct position, but, when added on the board, the piece rotates to the next possible position, position 3 . This only happens seldom, and the code for adding pieces on the board has been tested and works correctly by only rotating pieces in the wrong position. Therefore, this bug probably is not caused by the code itself. The cause for it might be the mouse moves in the GUI. Mouse clicks, mouse moves and mouse releases have different reactions. When a mouse is released, it could accidentally be clicked. Clicking a mouse rotates a piece, which would explain the rotating of a piece. This bug does not disturb the game since the piece can be rotated back to the original position. However, if there was more time, the reason for this small problem would have been properly solved.

Another bug or a weakness is that the coordinates for adding a piece on the board in the GUI are only accepted from a rectangle inside each triangle piece. In other words, to add a piece on the board, the mouse must be inside this rectangle while released. At the moment, the rectangle size is only about half of the triangle size. The rectangle is in the middle and lower part of the triangle piece, and, if the mouse is not inside it, the piece is not added to the board but added back to the pile. This weakness does not either cause problems to the game but might be annoying. The coordinate rules for adding pieces on the board could easily be improved by instead of only one rectangle having several rectangles on top of each other. By this means, the size of the rectangles would approach the triangle. If there was more time, the accepted coordinates on the board would have been improved. However, adding more coordinates makes the code much more complicated. Even now, adding the pieces on the board functions properly as long as the mouse is aimed at the lower center of the triangle, which is where the mouse normally lands in most cases.

Adding pieces to the pile could also be improved. Currently, adding a piece to the pile adds the piece at the end of the pile but sets it as the current piece. Therefore, the piece on top of the pile in the GUI is the added piece as wanted. However, the methods for flipping the pile always flip the pile continuing from the index of the current piece. Thus, when a piece is added to the pile, the method flipRight sets the current to the first piece in the pile and flipLeft sets it to the second to last piece in the pile. In most cases, this is not a problem but, if the current was previously some piece in the middle of the pile and a piece is added, the player has to flip through the pile all the way to the middle again. Adding pieces to the pile could be implemented in a way that the piece always is added in the middle of the pile after the current piece. This, however, affects

other parts of the code as well, and can not be implemented in a too short amount of time. If there was more time, this would be implemented.

The intelligent algorithm could be improved if there was more time. The algorithm finds a correct solution, but the solution is only one possible solution, not necessarily the generated solution. The algorithm could be improved to always find the generated solution. In addition, the method might be quite slow. In most cases, the method finds the solution quickly, but, in the worst case, it could take a long time to find the solution. For now, the method has always been able to find a solution, but there is a risk that the method would be too slow. Finding the generated solution might not be possible due to this slowness. Slowness could be fixed by improving the algorithm. However, it might currently be hard, when I do not yet have enough knowledge on machine learning.

An improvement could be made to the reading of a file. The method currently assumes that the read data is in a certain form. It assumes that the file contains three letters for the sides of a piece, one number for the position and possibly two numbers for its coordinates. In the game, the files should always have this form and it should not be a problem since there is no user data that is written to the file. However, considering further development of the game, it might be wise to add a mechanism that the given data is in correct form.

3 best sides and 3 weaknesses

One of the best sides is the algorithm for generating the solution. The method generateSolution is quite simple. The method has always worked fast and been able to generate the solution. Generating the solution is also divided into separate tasks which different helper methods manage. The division is quite clear and the method simple.

In addition, the program is well divided into separate classes, piece, pileOfPieces, board and game as well as fileOperations and puzzleGUI. Classes piece, pileOfPiece and board are especially quite separate from each other and have their own methods. The class game is mostly the only one that calls methods from all of the previous classes, and the puzzleGUI calls the methods in the class game. Private variables are used for var variables and mutable collections to avoid accidentally changing their values.

Overall, the program and especially the GUI became quite good and simple for the user to use with such little experience. This was the first GUI I have created, and it listens to buttons and mouse moves, positions the buttons and other elements correctly, and draws everything on the board correctly. The code for the GUI is somewhat messy, but it functions as intended.

The program has some weaknesses that would have been improved if there was more time. The quality of the code was not the best in all classes of the game. Some classes, such as piece and pileOfPieces, turned out good and the quality was rather good. However, especially the code in the GUI and the solve game method in the class game would need some improvements. Several if else clauses were used in both, which made the code complicated and long. The method convertCoords that converts coords from pixels on the screen to coordinates on the game board especially had many if conditions and became extremely long and complex. In this case, the method will always contain many conditions, and it might not be possible to get it completely clean. However, the conditions could have been simplified. In addition, the methods onDraw and onRelease were long. In the method onDraw, all of the lines and pictures must be drawn but the conditions for different situations could be simplified.

The intelligent algorithm could be improved in the method solveGame. Currently, the algorithm finds a solution, but it may be slow. In the worst case, the algorithm might not be efficient enough. For example, recursion could be another option for the algorithm. The method solveGame also contains several if else clauses, which make the method long and complicated.

There are some illogicalities in the program. For example, pieces contain information of the left, right and bottom side in this order whereas the information saved in a file contains the sides in order left, bottom and right. This could easily be edited by changing the order in the file. However, at this point this might cause bugs and might not be worth the change. Similarly, the coordinates are given differently for adding pieces to the board and checking a certain piece on the board. This logic could be changed, but it could cause other problems.

Deviations from the plan, realized process and schedule

The first thing created were the classes used in the game. The classes board, pileOfPieces and piece were created. The most crucial methods were implemented and variables, such as the sides for a piece and the pile buffer and board array, were added. For example, the methods addPiece and takePiece in the class pileOfPieces, the methods addPiece and removePiece in the class Board and rotate in the class Piece. These classes and important methods were implemented. It took approximately 5 hours as was estimated in the plan. The class game was also started and the method generateSolution and its algorithm was added as well as the gameOver condition. This took another 5 hours. The order of implementing the classes was as planned, but at this point, I was slightly ahead from the schedule.

During the next two weeks the fileOperations and the methods writeToFile and readFromFile were implemented. It took approximately 3 hours. The previously written code was tested and this took a couple of hours. At this point the basic structure of the game was ready. The graphical user interface was started. The structure for the interface was created, including all of the panels and buttons in the GUI. Writing the code itself did not take much time but searching information about creating GUIs with Scalas swing libraries was time consuming and took probably over ten hours.

The graphical user interface was finished. The method onPaint for drawing the background and pieces was implemented as well as the methods onClick for reacting to mouse clicks, onMove for reacting to mouse moves and onRelease for reacting to releasing a mouse. Especially the methods onMove and onRelease were tricky since the pixel coordinates had to be converted to the coordinates on the board. Therefore, all of this took approximately 10 hours. At this point the project was well ahead of the schedule.

During the end of the third two weeks and at the beginning of the fourth two weeks the algorithm and the method for solving the game were implemented. The method solveGame took perhaps 10 hours to finish and test. At this point the program was completely finished. The game was tested and some unit tests were written. The code was improved by simplifying it, writing comments in it and creating some helper methods for commonly used functionalities. The finishing touches and testing took 5 hours.

During the fifth two weeks some finishing touches were still made but the focus was on writing the documentation. The documentation was written, which took approximately 10 hours. All in all, the project was ahead in the schedule and finished in time. The order of written code in the schedule was quite good, and was followed. The time estimates were about right except for the GUI and testing, which took more time than estimated, mostly because information had to be researched. The deviations from the plan were mostly due to having already completed everything in the schedule and, thus, proceeding to next tasks early.

Final evaluation

The final result of the program is not perfect but quite good. The game functions as intended and contains all of the features of a hard version of the game. The game is divided into classes and objects: piece, pileOfPieces, board, game, fileOperations and puzzleGUI. Each of them describes its own part of the game. Therefore, the classes are quite separate from each other and are easy to test and possibly modify.

However, the project contains some weaknesses that could have been improved. The code in the GUI is long and contains many if else clauses. The methods onDraw, convertCoords and onRelease are especially long and complicated. These methods could have been simplified but only to some extent since reactions to drawing and moving a mouse are somewhat complicated. The quality of these parts of the code could be improved in the future. The intelligent algorithm functions properly and finds a solution for the most of the time. In worst cases, it might be too slow. The algorithm could have been improved. Currently, it starts trying piece after piece until it has found a solution, which is slow if the first correct pieces happen to be at the end of the pile. An ideal solution for the algorithm would have been that it tries to find the generated solution. Currently, the algorithm only finds some solution that often is not the generated one. Finding the generated solution would probably be too slow with the current algorithm, which is why the algorithm only finds some solution of all the possible solutions. The intelligent algorithm could be improved in the future after gaining more knowledge on the subject.

The GUI would have been improved, if there was more time to use. A completely separate window would have been created with the buttons ‘New game’ and ‘Continue game’, which would open when the program was started. When one of the buttons were clicked, the window with the board and the pile would become visible. However, there was not enough time for searching information on how to execute this. For the intelligent algorithm, a text ‘Solving…’ could have been added on the screen every time the algorithm is solving the puzzle in case it takes long. An implementation for this functionality was tried, but the text did not appear on the screen, probably due to parallel execution. There was not enough time to search information on executing this properly.

Some functionalities in the game could have been thought more in depth. For example, flipping through the pile could have been executed differently. Currently, the player can flip the pile forever and does not know when the end is reached. Adding pieces to the pile also causes illogicalities since the piece is added at the end of the pile but becomes the current piece. When a piece is added, the pile jumps to the end of the pile and loses the spot where the player was scrolling before adding the piece. The hint functionality could be improved since currently it removes all false pieces from the board and adds three new pieces. However, the player might not want to lose all pieces. However, only adding three new pieces might also cause problems since if the board is empty the player might not see where the new pieces went. The hint functionality would need some more planning to function smoothly.

The quality of the game is as good as it can be for a first big programming project. Of course, there are many parts that could be improved, but the quality of the code is good in many classes. For example, in the class piece and pileOfPieces, private variables are used for parts that could accidentally be changed. The methods are short and compact. The method generateSolution for generating the solution is also structured well by dividing it into smaller parts, and the algorithm always finishes fast.

The coordinates on the board could have been chosen differently and planned more precisely. The coordinates are given as actual places on the board for addPiece whereas while returning a piece from a certain position on the board the coordinates are given as indexes in the array. The coordinates returned from different methods are in some cases given in the form (y, x) and in other cases in the form (x, y), which causes confusion. The same applies to the sides of pieces. In the file, they are saved in order left, bottom and right, but, in a piece, they are saved as left, right and bottom. All of them are small details that could have been avoided by planning the project more precisely and could be improved in the future. The chosen data structures for the game functioned as needed. However, the buffer used for the intelligent algorithm could have been thought twice and possibly changed for another data structure, such as a stack since a buffer does not seem the best option for adding and removing pieces at the top of the collection.

The game structure suits quite well for changes and extensions. The classes are quite separate from each other. For example, changing methods in the class board does not affect the classes pile or pileOfPieces. A change for accepting more letters for pieces, for example, could be easily made since the classes pile, pileOfPieces and board do not prohibit different letters in pieces. Changes must only be made to the methods in the game class.

If the project was started again from the beginning, the game would have been planned in more detail. Illogicalities as described previously would have been avoided. The quality of the code and writing the code as simple as possible would have been focused on instead of only concentrating on the correct functionality of the code. In addition, the structure of the GUI would have been planned in advance. Otherwise, the project was planned quite accurately and executed on schedule. The project was overall a success.

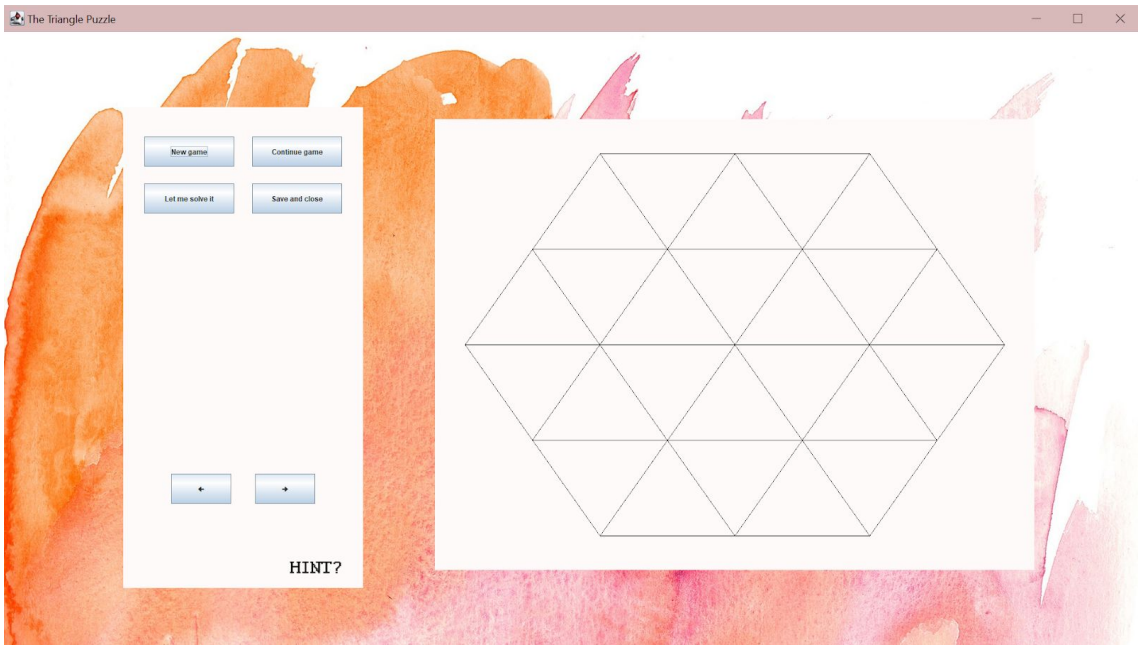
References

While doing this project, the course materials were used from courses Ohjelmointistudio 2 and Ohjelmointi 1. Also, the Scala standard library, Scala API , https://www.scala-lang.org/api/current/, and especially the Scala Swing library, https://www.scala-lang.org/api/2.9.1/scala/swing/package.html, was used. Especially for the graphical user interface, information was searched on the internet. Two websites were mainly used for the GUI: https://www.artima.com/pins1ed/gui-programming.html and http://otfried.org/scala/gui.html.

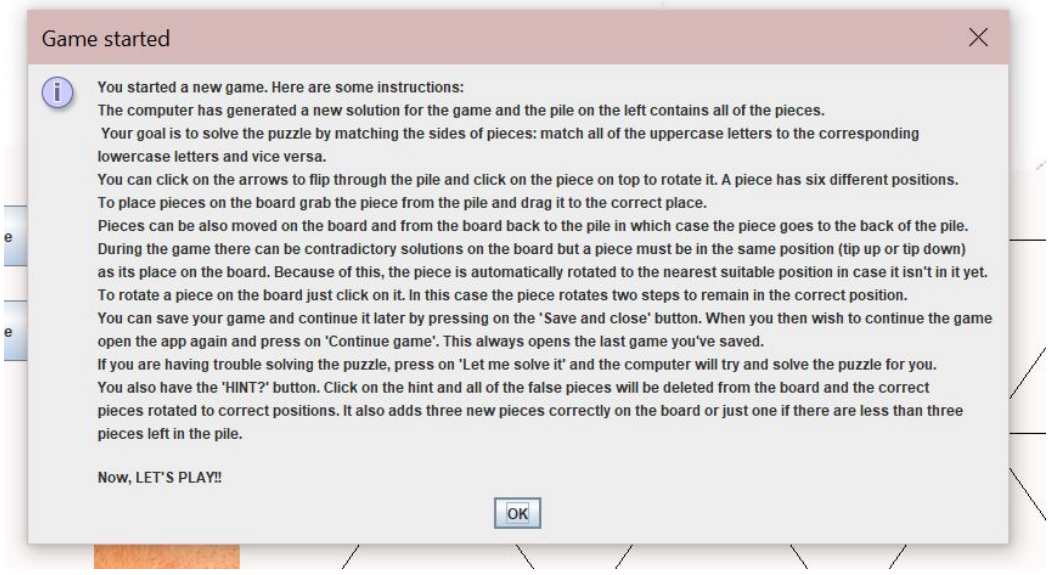
Appendixes

Appendix 1: Execution situations of the program

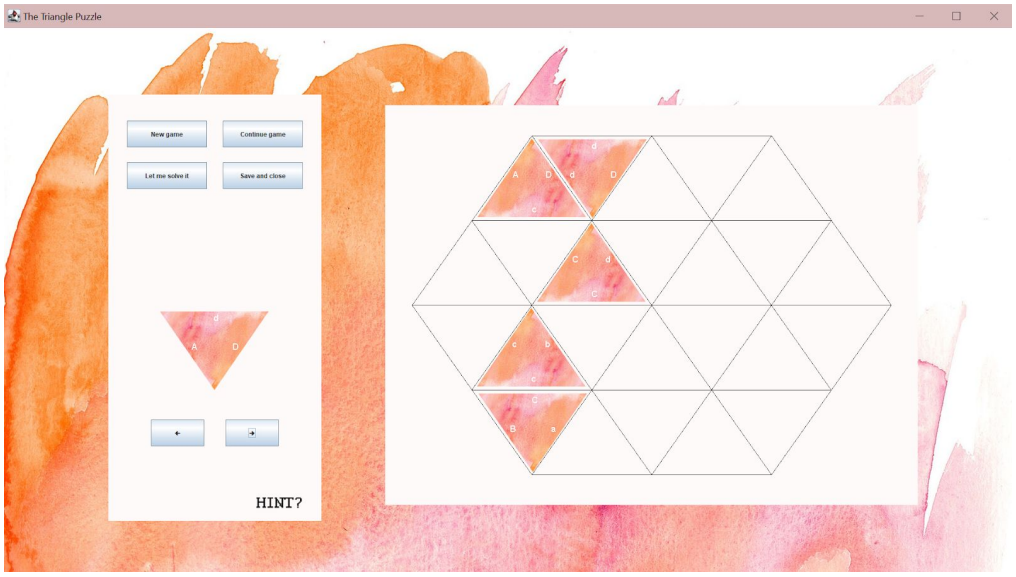
1. The program is started, and a window for the main frame opens.



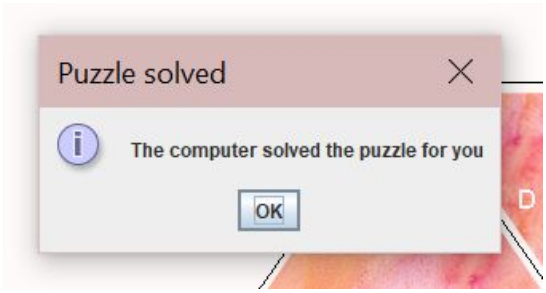
2. When a new game is started, a message pops up with information about the game.



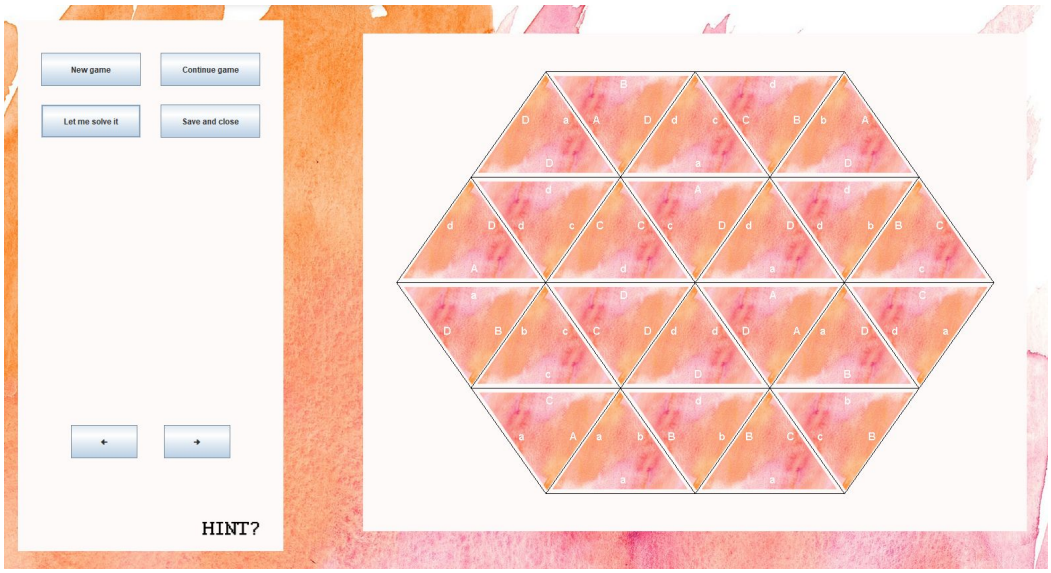
3. A situation in the middle of the game. Pieces are drawn on the board and in the pile.



4. When the button ‘Let me solve it’ has been pressed and the computer has solved the puzzle, a message pops up with a message to inform the player and the game ends.



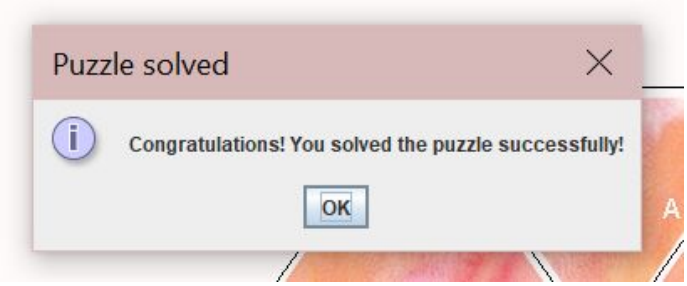
5. One complete solution on the board. The pile is empty.



6. When the button ‘Continue game’ has been pressed, a previously saved game is continued and a message pops up to inform the player.

7. When the button ‘Save and close’ has been pressed, the game is saved into a file and a message pops up to inform the player. If the player presses on ‘Yes’, the main frame closes.

8. When the player solves the puzzle, the game ends and a message pops up to congratulate the player.



Appendix 2: The source code of the project

The source code of the project is in the src-directory of the zip-file.