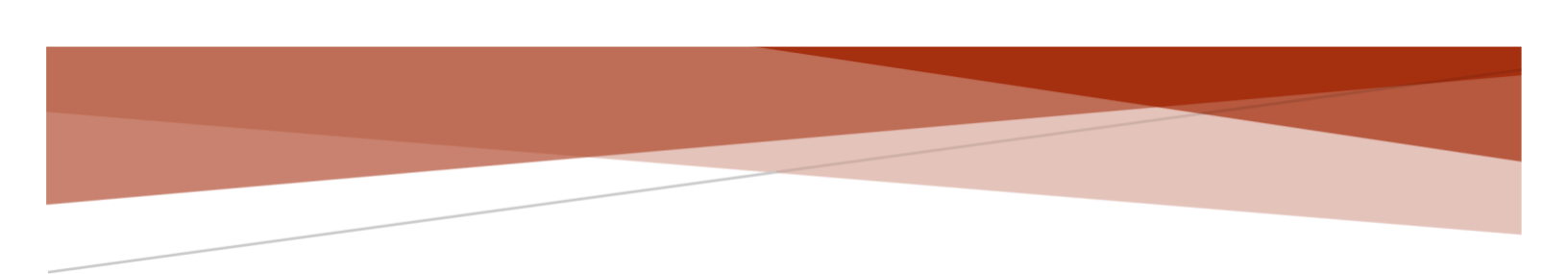
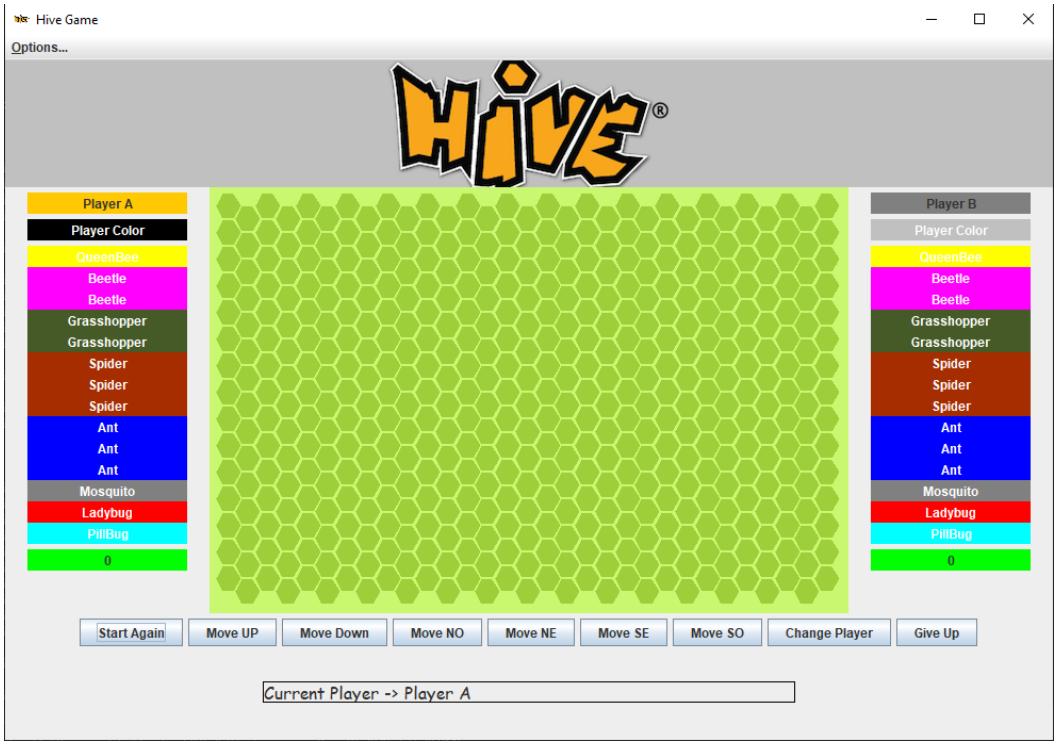


ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

TRABALHO PRÁTICO Nº4

Modelação e Programação | Semestre Verão 19/20



Docente: Engº António Teófilo
Alunos: 45746 Nuno Oliveira 45977 Eduardo Marques

Trabalho efetuado no âmbito da cadeira letiva de Modelação e Programação de Engenharia Informática e Multimédia.

Índice

Objetivos

Este trabalho tem por objetivo a familiarização com Swing, estruturas de dados dinâmicas e acesso a ficheiros.

Jogo da Colmeia – Hive Game

Objectivo do jogo

Pretende-se um programa em java que implemente o jogo Hive que é um jogo de tabuleiro para dois jogadores em que cada um tem 13 peças: 1 abelha rainha (QueenBee), 2 escaravelhos (Beetle), 2 gafanhotos (Grasshopper), 3 aranhas (Spider), 3 formigas (Ant), 1 mosquito, 1 joaninha (Ladybug) e 1 percevejo (Pillbug) e que tem como objetivo colocar a abelha rainha do adversário totalmente rodeada por peças (quaisquer peças). O jogador que começar deve selecionar com o rato uma das suas peças, ainda por colocar, e colocá-la no tabuleiro escolhendo a posição desejada com o rato. Cada jogador, na sua vez, ou faz a colocação de uma peça ou o movimento de uma das suas peças já colocadas.

O tabuleiro (classe Board) de células hexagonais deve ser suportado por um array 2D de elementos de uma classe de nome BoardPlace que deverá permitir conter várias peças sobrepostas. Para o seu posicionamento as colunas de x ímpar devem ser desfasadas em altura de ½ do lado das peças e todas as colunas devem ser sobrepostas com a coluna anterior em ¼ do lado das peças. A sua visualização será realizada com um hexágono cujos vértices se localizam a ¼ em x e ½ em y.

Assim, os vizinhos das células de uma coluna ímpar são de coordenadas relativas diferentes aos vizinhos das células de colunas par. Para resolver tal divergência e como num tabuleiro de células hexagonais o que se pretende é obter os vizinhos em função da direção, que só pode ser Norte (N), Nordeste (NE), Sudeste (SE), Sul (S), Sudoeste (SO) e Noroeste (NO), sugere-se a existência de uma função Point getNeighbourPoint(int x, int y, Direction d) que devolve o ponto vizinho de x,y na direção d ou null se ele não existir, caso já estivéssemos no rebordo do tabuleiro, onde Direction deve ser um enumerado com as coordenadas já descritas.

As peças devem ter uma peça base (classe Piece) da qual todas as peças devem derivar. Os BoardPlaces devem possuir um ArrayList de forma a poderem conter várias peças, pois pode haver sobreposições.

O jogo deverá mostrar: qual é o jogador corrente; para cada jogador o seu número de movimentos (em que cada colocação ou passar de vez equivale a um movimento) realizados e as suas peças por colocar; nos casos do um escaravelho estar por cima de outras peças tem de conseguir saber que peças são e a sua ordem.

Dever-se-á permitir: reiniciar o jogo; dar a vez ao outro jogador; mover o Hive (a colmeia) nas várias direções (N, NO, NE, …), se possível; e mostrar as mensagens numa label. Deverá existir também um menu com as opções de: reiniciar o jogo; ver scores; About que mostre o nome do jogo, do curso, da UC, dos autores (nº e nome) e com uma foto dos mesmos; e Help que mostre as regras do jogo.

O jogo deverá manter o registo, em ficheiro, dos melhores 10 resultados em relação ao menor nº de movimentos necessários para ganhar. Para tal, assim que um jogador termina dentro desses 10 deve-se pedir o seu nome e registar esse facto. Esses 10 melhores resultados podem ser vistos ordenados por nº de movimentos (por omissão) ou por nome, mas mantendo a indicação

do lugar na ordenação por omissão para cada nome. Essas visualizações devem utilizar comparadores para fazer a ordenação. O ficheiro de scores deve ser um ficheiro de texto com linhas contendo: nome - nº de movimentos.

Regras

Regras gerais

Das regras gerais salienta-se:

R1: Quando se coloca uma peça no tabuleiro ela não pode ter vizinhos inimigos, à exceção da primeira peça do segundo jogador a jogar.

R2: A abelha rainha (QueenBee) tem de ser colocada obrigatoriamente até à 4ª jogada. Só se pode fazer movimentos de peças de um jogador depois de ele ter colocador a sua abelha rainha.

R4: O Hive deve ser sempre um. O movimento das peças também não deve gerar dois Hives, mesmo que momentaneamente, ou seja, os lugares de origem e de destino de qualquer movimento de uma posição, deve ser conectado por outra(s) peça(s), que não aquela que se quer mover. Um movimento de várias posições (exceto no caso do gafanhoto) é uma composição de vários movimentos de uma posição. Todas as peças têm de respeitar esta regra.

R5: Regra de fim de jogo: o jogo termina quando uma abelha rainha (QueenBee), ou as duas (empate), estiver(em) totalmente rodeada(s) por (quaisquer) peças.

R6: Um jogador pode passar a sua vez, contando como um seu movimento.

R7: As peças Abelha Rainha (QueenBee), Aranha (Spider) e formiga (Ant) só se podem deslocar se o movimento for fisicamente realizável em cada passo. Cada peça para poder colocada no tabuleiro tem de poder deslizar fisicamente do rebordo para o lugar de destino.

Regras sobre a caracterização do movimento

Das regras sobre a caracterização do movimento de cada peça salienta-se:

RQ: A abelha rainha (QueenBee) só se pode deslocar uma posição.

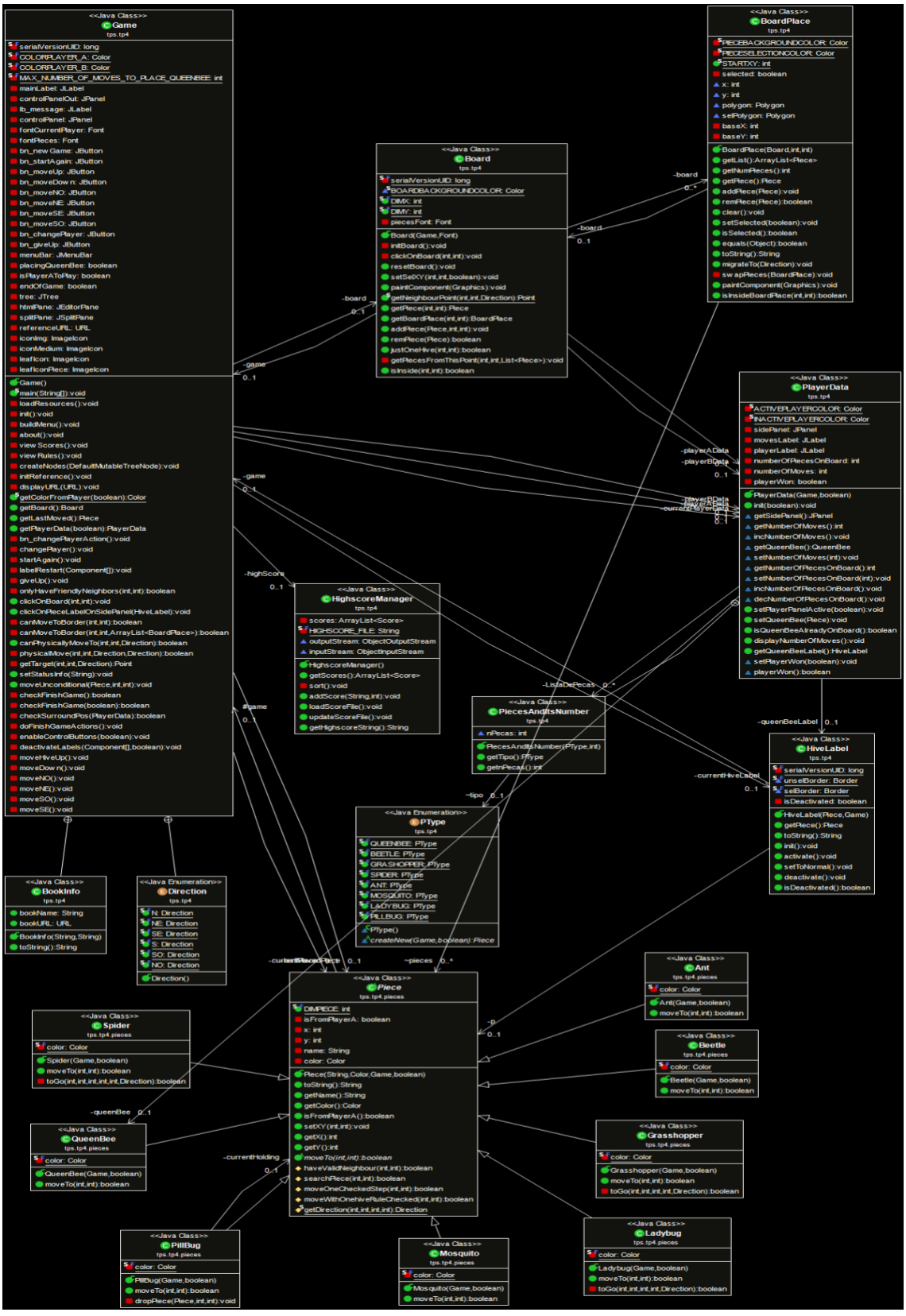
RB: Os escaravelhos (Beetle) só se deslocam 1 posição, mas podem subir para cima de outras peças colocadas, mesmo ficando uns por cima dos outros. É a única peça que pode subir para cima de outras. A peça que estiver por cima é que pode ser movimentada e que conta como a peça daquela posição. O escravelho não pode ser colocado inicialmente em cima de outra peça.

RG: Os gafanhotos (Grasshopper) têm de saltar, mas sempre em linha reta e por cima de pelo menos uma posição ocupada. Mas não podem saltar por cima de posições não ocupadas.

RS: As aranhas (Spider) devem deslocar-se sempre em movimentos de 3 posições diferentes.

RA: As formigas (Ant) podem deslocar-se um qualquer nº de posições.

Diagrama UML



O diagrama UML tem o propósito de fornecer ao programador uma melhor visualização do programa, visto que todas as ligações entre classes e todas as classes estão dispostas graficamente.

Anexos (Código)

Class Game

```
package tps.tp4;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.io.IOException;
import java.net.URL;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;

import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JEditorPane;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTree;
import javax.swing.KeyStroke;
import javax.swing.WindowConstants;
import javax.swing.event.TreeSelectionEvent;
import javax.swing.event.TreeSelectionListener;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeCellRenderer;
import javax.swing.tree.TreeSelectionModel;
```

```
import tps.layouts.CenterLayout;
import tps.tp4.pieces.Piece;

/**
 * HIVE GAME: one Queen, two Beetles, two
Grasshoppers, three Spiders and three
 * Ants
 *
 * http://en.wikipedia.org/wiki/Hive_(game)
 *
http://www.gen42.com/downloads/rules/Hive_Rules.pdf
 */

/**
 * the main class - that supports the game
 */
public class Game extends JFrame {

    // enumerate that supports directions
    public enum Direction {
        N, NE, SE, S, SO, NO
    }

    private static final long serialVersionUID = 1L;
    private static final Color COLORPLAYER_A =
Color.black;
    private static final Color COLORPLAYER_B =
Color.lightGray;
    private static final int
MAX_NUMBER_OF_MOVES_TO_PLACE_QUEENBEE = 4;

    private JLabel mainLabel;
    // private JLabel subMainLabel;
    private HiveLabel currentHiveLabel = null;
    private Piece currentPiece = null;
    private JPanel controlPanelOut;
    private JLabel lb_message;
    private JPanel controlPanel;

    private Board board;

    private Font fontCurrentPlayer;
    private Font fontPieces;

    // buttons to move the Hive, if possible
```

```
    private JButton bn_newGame; // Adicionado por
Nuno
    private JButton bn_startAgain; // Adicionado por
Nuno
    private JButton bn_moveUp;
    private JButton bn_moveDown;
    private JButton bn_moveNO;
    private JButton bn_moveNE;
    private JButton bn_moveSE;
    private JButton bn_moveSO;
    private JButton bn_changePlayer;
    private JButton bn_giveUp;

    private JMenuBar menuBar;

    private boolean placingQueenBee = false;
    private boolean isPlayerAToPlay;
    private PlayerData currentPlayerData;
    private boolean endOfGame = false;

    private PlayerData playerAData;
    private PlayerData playerBData;

    private JTree tree;
    private JEditorPane htmlPane;
    private JSplitPane splitPane;
    private URL referenceURL;

    private HighscoreManager highScore;

    private ImageIcon iconImg, iconMedium, leafIcon,
leafIconPiece;

    private Piece lastMovedPiece;

    /**
     * methods
==========================================
     */

    /**
     * main
     */
    public static void main(String[] args) {
```

```
        javax.swing.SwingUtilities.invokeLater(new
Runnable() {
            public void run() {
                // create object Game
                Game g = new Game();
                // launch the frame, but will be
activated with some delay
                g.init();
            }
        });
        System.out.println("End of main...");

    }

    /**
     * load resources: fonts, images, sounds
     */
    private void loadResources() {

        String fontType = "Comic Sans MS";
        int size = 16;
        this.fontCurrentPlayer = new Font(fontType,
Font.PLAIN, size);
        this.fontPieces = new Font(fontType,
Font.BOLD, size);
        this.iconImg = new
ImageIcon(this.getClass().getResource("images/logo/hi
veLogo2.png"));
        this.iconMedium = new
ImageIcon(this.getClass().getResource("images/icon/hi
veIconMedium.png"));
        this.leafIcon = new
ImageIcon(this.getClass().getResource("images/icon/hi
veIconSmall.png"));
        this.leafIconPiece = new
ImageIcon(this.getClass().getResource("images/icon/qu
eenbeenIcon.png"));
    }

    /**
     * the JFrame initialization method
     */
    private void init() {
        highScore = new HighscoreManager();
        setTitle("Hive Game");
```

```
        setSize(1000, 700);
        loadResources();
        this.isPlayerAToPlay = true;
        setDefaultCloseOperation(WindowConstants.DISP
OSE_ON_CLOSE);
        setLayout(new CenterLayout());
        setLocationRelativeTo(null);
        setIconImage(iconImg.getImage());

        lastMovedPiece = null;

        playerAData = new PlayerData(this, true);
        JPanel panelA = new JPanel();
        panelA = playerAData.getSidePanel();
        add(panelA, BorderLayout.WEST);

        playerBData = new PlayerData(this, false);
        JPanel panelB = new JPanel();
        panelB = playerBData.getSidePanel();
        add(panelB, BorderLayout.EAST);

        board = new Board(this, fontPieces);
        add(board, BorderLayout.CENTER);

        controlPanel = new JPanel(new GridLayout(2,
9, 0, 0));

        Game that = this;
        ActionListener al = new ActionListener() {

            public void actionPerformed(ActionEvent
e) {
                switch (e.getActionCommand()) {
                case "new_game":
                    enableControlButtons(true);
                    startAgain();
                    break;
                case "start_again":
                    int n =
JOptionPane.showConfirmDialog(that, "Are you sure
about your decision?",
                            "Restart Game
Confirmation", JOptionPane.YES_NO_OPTION,
JOptionPane.QUESTION_MESSAGE,
                            iconMedium);
```

```
                    if (n == JOptionPane.YES_OPTION)
{
                        enableControlButtons(true);
                        startAgain();
                    } else {
                        return;
                    }
                    break;
                case "move_up":
                    moveHiveUp();
                    break;
                case "move_down":
                    moveDown();
                    break;
                case "move_no":
                    moveNO();
                    break;
                case "move_ne":
                    moveNE();
                    break;
                case "move_se":
                    moveSE();
                    break;
                case "move_so":
                    moveSO();
                    break;
                case "change_player":
                    bn_changePlayerAction();
                    break;
                case "give_up":
                    giveUp();
                    break;
                }
            }
        };
        JPanel buttons = new JPanel();

        bn_newGame = new JButton("New Game");
        bn_newGame.setActionCommand("new_game");
        bn_newGame.addActionListener(al);
        bn_newGame.setVisible(false);
        bn_newGame.setEnabled(false);
        buttons.add(bn_newGame);

        bn_startAgain = new JButton("Start Again");
```

```
        bn_startAgain.setActionCommand("start_again")
;
        bn_startAgain.addActionListener(al);
        buttons.add(bn_startAgain);

        bn_moveUp = new JButton("Move UP");
        bn_moveUp.setActionCommand("move_up");
        bn_moveUp.addActionListener(al);
        buttons.add(bn_moveUp);

        bn_moveDown = new JButton("Move Down");
        bn_moveDown.setActionCommand("move_down");
        bn_moveDown.addActionListener(al);
        buttons.add(bn_moveDown);

        bn_moveNO = new JButton("Move NO");
        bn_moveNO.setActionCommand("move_no");
        bn_moveNO.addActionListener(al);
        buttons.add(bn_moveNO);

        bn_moveNE = new JButton("Move NE");
        bn_moveNE.setActionCommand("move_ne");
        bn_moveNE.addActionListener(al);
        buttons.add(bn_moveNE);

        bn_moveSE = new JButton("Move SE");
        bn_moveSE.setActionCommand("move_se");
        bn_moveSE.addActionListener(al);
        buttons.add(bn_moveSE);

        bn_moveSO = new JButton("Move SO");
        bn_moveSO.setActionCommand("move_so");
        bn_moveSO.addActionListener(al);
        buttons.add(bn_moveSO);

        bn_changePlayer = new JButton("Change
Player");
        bn_changePlayer.setActionCommand("change_play
er");
        bn_changePlayer.addActionListener(al);
        buttons.add(bn_changePlayer);

        bn_giveUp = new JButton("Give Up");
        bn_giveUp.setActionCommand("give_up");
        bn_giveUp.addActionListener(al);
```

```
        buttons.add(bn_giveUp);

        controlPanelOut = new JPanel();
        lb_message = new JLabel("Log do Jogo");
        lb_message.setFont(fontCurrentPlayer);
        lb_message.setPreferredSize(new
Dimension(500, 20));
        lb_message.setBorder(BorderFactory.createLine
Border(Color.BLACK, 1));
        controlPanelOut.add(lb_message);

        controlPanel.add(buttons,
BorderLayout.CENTER);
        controlPanel.add(controlPanelOut,
BorderLayout.SOUTH);
        add(controlPanel, BorderLayout.SOUTH);

        buildMenu();
        currentPlayerData = playerAData;

        // Main Label
        mainLabel = new JLabel(iconImg);
        lb_message.setText("Current Player -> " +
"Player A");
        mainLabel.setOpaque(true);
        mainLabel.setBackground(Color.LIGHT_GRAY);
        add(mainLabel, BorderLayout.NORTH);

        setVisible(true);
    }

    /**
     * build menu
     */
    private void buildMenu() {
        JMenu menu;
        JMenuItem restartMenuItem;
        JMenuItem viewScoresItem;
        JMenuItem aboutItem;
        ActionListener al = null;

        // Menu Action Listener
        al = new ActionListener() {
            public void actionPerformed(ActionEvent
e) {
```

```
                JMenuItem mi = (JMenuItem)
(e.getSource());
                String menuItemText = mi.getText();
                switch (menuItemText) {
                case "Restart Game":
                    enableControlButtons(true);
                    startAgain();
                    break;
                case "View Scores":
                    viewScores();
                    break;
                case "About":
                    about();
                    break;
                case "Rules":
                    viewRules();
                    break;
                }
            }
        };

        // Create the menu bar.
        menuBar = new JMenuBar();

        // Build the menu.
        menu = new JMenu("Options...");
        menu.setMnemonic(KeyEvent.VK_O);
        menu.addSeparator();
        menuBar.add(menu);

        restartMenuItem = new JMenuItem("Restart
Game", KeyEvent.VK_S);
        restartMenuItem.setAccelerator(KeyStroke.getK
eyStroke(KeyEvent.VK_S, ActionEvent.CTRL_MASK));
        restartMenuItem.addActionListener(al);
        menu.add(restartMenuItem);

        menu.addSeparator();

        viewScoresItem = new JMenuItem("View Scores",
KeyEvent.VK_P);
        viewScoresItem.setAccelerator(KeyStroke.getKe
yStroke(KeyEvent.VK_P, ActionEvent.CTRL_MASK));
        viewScoresItem.addActionListener(al);
        menu.add(viewScoresItem);
```

```
        menu.addSeparator();

        aboutItem = new JMenuItem("About",
KeyEvent.VK_A);
        aboutItem.setAccelerator(KeyStroke.getKeyStro
ke(KeyEvent.VK_A, ActionEvent.CTRL_MASK));
        aboutItem.addActionListener(al);
        menu.add(aboutItem);

        JMenuItem helpItem = new JMenuItem("Rules",
KeyEvent.VK_R);
        helpItem.setAccelerator(KeyStroke.getKeyStrok
e(KeyEvent.VK_R, ActionEvent.CTRL_MASK));
        helpItem.addActionListener(al);
        menu.add(helpItem);

        // set Menu Bar on JFrame
        setJMenuBar(menuBar);
    }

    /**
     * activate About window
     */
    private void about() {
        String content = null;
        try {
            content = new
String(Files.readAllBytes(Paths.get("src/tps/tp4/abou
t.txt")));
        } catch (IOException e) {
            e.printStackTrace();
        }
        JOptionPane.showMessageDialog(this, content,
"About information.",
JOptionPane.INFORMATION_MESSAGE, iconImg);
    }

    /**
     * activate View scores window
     */
    private void viewScores() {
        JOptionPane.showMessageDialog(this,
highScore.getHighscoreString(), "Top Scores",
```

```
                JOptionPane.INFORMATION_MESSAGE,
iconMedium);
    }

    /**
     * activate View Rules Window
     */
    private void viewRules() {

        // Create the nodes.
        DefaultMutableTreeNode top = new
DefaultMutableTreeNode("The Hive Game");
        createNodes(top);

        // Create a tree that allows one selection at
a time.
        tree = new JTree(top);
        tree.getSelectionModel().setSelectionMode(Tre
eSelectionModel.SINGLE_TREE_SELECTION);

        DefaultTreeCellRenderer renderer = new
DefaultTreeCellRenderer();
        renderer.setLeafIcon(leafIconPiece);
        renderer.setOpenIcon(leafIcon);
        renderer.setClosedIcon(leafIcon);
        tree.setCellRenderer(renderer);

        TreeSelectionListener tsl = new
TreeSelectionListener() {
            @Override
            public void
valueChanged(TreeSelectionEvent e) {
                DefaultMutableTreeNode node =
(DefaultMutableTreeNode)
tree.getLastSelectedPathComponent();

                if (node == null)
                    return;

                Object nodeInfo =
node.getUserObject();
                if (node.isLeaf() && (nodeInfo
instanceof BookInfo)) {
                    BookInfo book = (BookInfo)
nodeInfo;
```

```
                    displayURL(book.bookURL);

                } else {
                    displayURL(referenceURL);
                }

            }
        };
        // Listen for when the selection changes.
        tree.addTreeSelectionListener(tsl);

        // Create the scroll pane and add the tree to
it.
        JScrollPane treeView = new JScrollPane(tree);

        // Create the HTML viewing pane.
        htmlPane = new JEditorPane();
        htmlPane.setEditable(false);
        initReference();
        JScrollPane htmlView = new
JScrollPane(htmlPane);

        // Add the scroll panes to a split pane.
        splitPane = new
JSplitPane(JSplitPane.VERTICAL_SPLIT);
        splitPane.setTopComponent(treeView);
        splitPane.setBottomComponent(htmlView);

        Dimension minimumSize = new Dimension(100,
50);
        htmlView.setMinimumSize(minimumSize);
        treeView.setMinimumSize(minimumSize);
        splitPane.setDividerLocation(100);

        splitPane.setPreferredSize(new Dimension(500,
300));
        JOptionPane.showMessageDialog(this,
splitPane, "Rules", JOptionPane.INFORMATION_MESSAGE,
iconMedium);
    }

    private void createNodes(DefaultMutableTreeNode
top) {
        DefaultMutableTreeNode category = null;
        DefaultMutableTreeNode book = null;
```

```
        category = new
DefaultMutableTreeNode("General");
        top.add(category);

        // Components
        book = new DefaultMutableTreeNode(new
BookInfo("Components", "rules/component.html"));
        category.add(book);

        // Setup of the game
        book = new DefaultMutableTreeNode(new
BookInfo("Setup", "rules/setup.html"));
        category.add(book);

        // Objective of the game
        book = new DefaultMutableTreeNode(new
BookInfo("Objective of the Game",
"rules/objective.html"));
        category.add(book);

        // Game play
        book = new DefaultMutableTreeNode(new
BookInfo("Game Play", "rules/gameplay.html"));
        category.add(book);

        // Placing
        book = new DefaultMutableTreeNode(new
BookInfo("Placing", "rules/placing.html"));
        category.add(book);

        // Moving
        book = new DefaultMutableTreeNode(new
BookInfo("Moving", "rules/moving.html"));
        category.add(book);

        // Pieces
        category = new
DefaultMutableTreeNode("Pieces");
        top.add(category);

        // Queen Bee
        book = new DefaultMutableTreeNode(new
BookInfo("Queen Bee", "rules/pieces/queenbee.html"));
        category.add(book);
```

```
        // Beetle
        book = new DefaultMutableTreeNode(new
BookInfo("Beetle", "rules/pieces/beetle.html"));
        category.add(book);

        // Grasshopper
        book = new DefaultMutableTreeNode(new
BookInfo("Grasshopper",
"rules/pieces/grasshopper.html"));
        category.add(book);

        // Spider
        book = new DefaultMutableTreeNode(new
BookInfo("Spider", "rules/pieces/spider.html"));
        category.add(book);

        // Ant
        book = new DefaultMutableTreeNode(new
BookInfo("Ant", "rules/pieces/ant.html"));
        category.add(book);

        // Ladybug
        book = new DefaultMutableTreeNode(new
BookInfo("Ladybug", "rules/pieces/ladybug.html"));
        category.add(book);

        // Mosquito
        book = new DefaultMutableTreeNode(new
BookInfo("Mosquito", "rules/pieces/mosquito.html"));
        category.add(book);

        // Pillbug
        book = new DefaultMutableTreeNode(new
BookInfo("Pillbug", "rules/pieces/pillbug.html"));
        category.add(book);

        // Restrictions
        category = new
DefaultMutableTreeNode("Restrictions");
        top.add(category);

        // One Hive Move
```

```
        book = new DefaultMutableTreeNode(new
BookInfo("One Hive Move",
"rules/restrictions/onehivemove.html"));
        category.add(book);

        // Freedom to Move
        book = new DefaultMutableTreeNode(new
BookInfo("Freedom to Move",
"rules/restrictions/freedomtomove.html"));
        category.add(book);

        // Unable to Move or to Place
        book = new DefaultMutableTreeNode(
                new BookInfo("Unable to Move or to
Place", "rules/restrictions/unabletomove.html"));
        category.add(book);

    }

    private class BookInfo {
        public String bookName;
        public URL bookURL;

        public BookInfo(String book, String filename)
{
            bookName = book;
            bookURL =
this.getClass().getResource("docs/" + filename);
            if (bookURL == null) {
                System.err.println("Couldn't find
file: " + filename);
            }
        }

        public String toString() {
            return bookName;
        }
    }

    private void initReference() {
        String s = "docs/rules/" + "reference.html";
        referenceURL =
this.getClass().getResource(s);
        if (referenceURL == null) {
```

```
            System.err.println("Couldn't open help
file: " + s);
        }
        displayURL(referenceURL);
    }

    private void displayURL(URL url) {
        try {
            if (url != null) {
                htmlPane.setPage(url);
            } else { // null url
                htmlPane.setText("File Not Found");
            }
        } catch (IOException e) {
            System.err.println("Attempted to read a
bad URL: " + url);
        }
    }

    /**
     * get color from player
     */
    static public Color getColorFromPlayer(boolean
isPlayerA) {
        return isPlayerA ? COLORPLAYER_A :
COLORPLAYER_B;
    }

    /**
     * get board
     */
    public Board getBoard() {
        return board;
    }

    public Piece getLastMoved() {
        return lastMovedPiece;
    }

    /**
     * get player data
     */
    public PlayerData getPlayerData(boolean
fromPlayerA) {
```

```
        return fromPlayerA ? playerAData :
playerBData;
    }

    /**
     * change player actions - to be called from the
menu or from the button
     * changePlayer
     */
    private void bn_changePlayerAction() {
        this.changePlayer();
    }

    /**
     * change player actions
     */
    private void changePlayer() {
        if (currentPlayerData.getNumberOfMoves() ==
MAX_NUMBER_OF_MOVES_TO_PLACE_QUEENBEE - 1
                &&
!currentPlayerData.isQueenBeeAlreadyOnBoard()) {
            lb_message.setText("You need to place the
QueenBee.");
            if (!placingQueenBee)
                return;
        }
        currentPlayerData.incNumberOfMoves();
        if (currentPlayerData.equals(playerAData)) {
            currentPlayerData = playerBData;
            isPlayerAToPlay = false;
            playerAData.setPlayerPanelActive(isPlayer
AToPlay);
            playerBData.setPlayerPanelActive(true);
            lb_message.setText("Current Player ->
Player B");
        } else {
            currentPlayerData = playerAData;
            isPlayerAToPlay = true;
            playerBData.setPlayerPanelActive(!isPlaye
rAToPlay);
            playerAData.setPlayerPanelActive(isPlayer
AToPlay);
            lb_message.setText("Current Player ->
Player A");
        }
```

```
        if
(!currentPlayerData.isQueenBeeAlreadyOnBoard())
            placingQueenBee = false;
        if (currentHiveLabel != null) {
            currentHiveLabel.setToNormal();
            currentHiveLabel = null;
        }
        if (currentPiece != null) {
            board.getBoardPlace(currentPiece.getX(),
currentPiece.getY()).setSelected(false);
            currentPiece = null;
        }

    }

    /**
     * start again actions
     */
    private void startAgain() {

        Component[] playAComponents =
playerAData.getSidePanel().getComponents();
        Component[] playBComponents =
playerBData.getSidePanel().getComponents();
        this.labelRestart(playAComponents);
        this.labelRestart(playBComponents);
        playerAData.setNumberOfMoves(0);
        playerAData.setNumberOfPiecesOnBoard(0);
        playerBData.setNumberOfMoves(0);
        playerBData.setNumberOfPiecesOnBoard(0);
        currentPlayerData = playerAData;
        currentPlayerData.setPlayerPanelActive(true);
        playerBData.setPlayerPanelActive(false);
        isPlayerAToPlay = true;
        lb_message.setText("Current Player -> Player
A");
        if (currentPiece != null)
            board.getBoardPlace(currentPiece.getX(),
currentPiece.getY()).setSelected(false);
        board.resetBoard();
        board.repaint();
        currentHiveLabel = null;
        currentPiece = null;
        bn_newGame.setVisible(false);
        bn_newGame.setEnabled(false);
```

```
    }

    /**
     * Restart the Pieces label to normal state
     *
     * @param component -> components from one player
     */
    private void labelRestart(Component[] component)
{
        for (Component c : component) {
            if (c instanceof JPanel) {
                JPanel jp = (JPanel) c;
                Component[] piecesLabel =
jp.getComponents();
                for (Component x : piecesLabel) {
                    HiveLabel z = (HiveLabel) x;
                    z.setToNormal();
                }
            }
            c.setEnabled(true);
        }
    }

    private void giveUp() {

        int n = JOptionPane.showConfirmDialog(this,
"Are you that chicken?", "Give Up Confirmation",
                JOptionPane.YES_NO_OPTION,
JOptionPane.QUESTION_MESSAGE, iconMedium);
        if (n == JOptionPane.YES_OPTION) {
            this.endOfGame = true;

            if (this.checkFinishGame()) {
                if (currentPiece != null &&
currentHiveLabel != null) {
                    board.getBoardPlace(currentPiece.
getX(), currentPiece.getY()).setSelected(false);
                    currentHiveLabel = null;
                    currentPiece = null;
                }
                this.doFinishGameActions();
            }

        } else if (n == JOptionPane.NO_OPTION) {
```

```
            return;
        }
    }

    /**
     * check if coordinate x,y only have friendly
neighbor of current player
     */
    private boolean onlyHaveFriendlyNeighbors(int x,
int y) {
        boolean enemyNeib = false;
        for (Direction d : Direction.values()) {
            Point p = Board.getNeighbourPoint(x, y,
d);
            if (p == null)
                continue;
            int k = (int) p.getX();
            int j = (int) p.getY();
            Piece piece = board.getPiece(k, j);
            if (piece == null)
                continue;
            if (board.getPiece(k, j).isFromPlayerA()
&& currentPlayerData.equals(playerAData)) {
                enemyNeib = true;
            } else if (!board.getPiece(k,
j).isFromPlayerA() &&
currentPlayerData.equals(playerBData)) {
                enemyNeib = true;
            } else if (board.getPiece(k,
j).isFromPlayerA() &&
currentPlayerData.equals(playerBData)) {
                enemyNeib = false;
                break;
            } else if (!board.getPiece(k,
j).isFromPlayerA() &&
currentPlayerData.equals(playerAData)) {
                enemyNeib = false;
                break;
            }
        }
        return enemyNeib;
    }

    /**
     * a click on the board
```

```
     */
    public void clickOnBoard(int x, int y) {

        if (currentPlayerData.getNumberOfMoves() ==
MAX_NUMBER_OF_MOVES_TO_PLACE_QUEENBEE - 1
                &&
!currentPlayerData.isQueenBeeAlreadyOnBoard()) {
            lb_message.setText("You need to place the
QueenBee.");
            if (!placingQueenBee)
                return;
        }
        if (currentHiveLabel == null) {
            Piece p = board.getPiece(x, y);
            if (p == null && currentPiece == null) {
                return;
            }
            if (currentPiece != null) {
                if (currentPiece.equals(p)) {
                    board.getBoardPlace(x,
y).setSelected(false);
                    currentPiece = null;
                    return;
                }
                int pastX = currentPiece.getX(),
pastY = currentPiece.getY();
                if
(currentPlayerData.isQueenBeeAlreadyOnBoard() &&
currentPiece.moveTo(x, y)) {
                    lastMovedPiece = currentPiece;
                    board.getBoardPlace(pastX,
pastY).setSelected(false);
                    board.repaint();
                    if (checkFinishGame()) {
                        doFinishGameActions();
                    }
                    changePlayer();
                    return;
                }
                lb_message.setText("Invalid
movement!");
                return;
            } else {
```

```
                if (!isPlayerAToPlay &&
p.isFromPlayerA() || isPlayerAToPlay &&
!p.isFromPlayerA()) {
                    return;
                }
                currentPiece = p;
                board.getBoardPlace(x,
y).setSelected(true);
                return;
            }
        }
        if (isPlayerAToPlay) {
            if (currentPlayerData.getNumberOfMoves()
!= 0 && !this.onlyHaveFriendlyNeighbors(x, y)) {
                lb_message.setText("Invalid position!
Can't play here!");
                return;
            } else {
                Piece p = board.getPiece(x, y);
                if (p != null) {
                    lb_message.setText("All pieces
must be played on the Board");
                    return;
                }
            }
        } else {
            if (currentPlayerData.getNumberOfMoves()
== 0) {
                boolean validNeib = false;
                for (Direction d :
Direction.values()) {
                    Point p =
Board.getNeighbourPoint(x, y, d);
                    if (p == null)
                        continue;
                    int k = (int) p.getX();
                    int j = (int) p.getY();
                    if (board.getPiece(k, j) != null)
                        validNeib = true;
                }
                if (!validNeib) {
                    lb_message.setText("The Piece has
must be played adjacent to another piece");
                    return;
                }
```

```
            } else if
(!this.onlyHaveFriendlyNeighbors(x, y)) {
                lb_message.setText("Invalid position!
Can't play here!");
                return;
            } else {
                Piece p = board.getPiece(x, y);
                if (p != null) {
                    lb_message.setText("All pieces
must be played on the Board");
                    return;
                }
            }
        }
        if (placingQueenBee) {
            currentPlayerData.setQueenBee(currentHive
Label.getPiece());
            placingQueenBee = false;
        }

        board.addPiece(currentHiveLabel.getPiece(),
x, y);
        board.repaint();
        currentHiveLabel.deactivate();
        currentHiveLabel = null;
        if (checkFinishGame()) {
            doFinishGameActions();
        }
        changePlayer();
    }

    /**
     * a click on a label on side panel
     */
    public void
clickOnPieceLabelOnSidePanel(HiveLabel hl) {
        if (hl.isDeactivated())
            return;
        if (hl.getPiece().isFromPlayerA() !=
isPlayerAToPlay)
            return;
        if (currentHiveLabel != null) {
            currentHiveLabel.setToNormal();
            if (placingQueenBee)
                placingQueenBee = false;
```

```
            if (currentHiveLabel.equals(hl)) {
                currentHiveLabel = null;
                return;
            }
        }
        if (currentPiece != null) {
            board.getBoardPlace(currentPiece.getX(),
currentPiece.getY()).setSelected(false);
            currentPiece = null;
        }
        currentHiveLabel = hl;
        currentHiveLabel.activate();
        if
(currentHiveLabel.getPiece().getName().equalsIgnoreCa
se("queenbee"))
            placingQueenBee = true;
    }

    /**
     * Can move to border, used to check if piece can
be placed on hive physically
     * sliding from the border. We can use a
ArrayList to keep the boardPlaces and
     * try to find a way to the border. The ArrayList
is used to avoid loops. If a
     * new boardPlace is already in the ArrayList so
it will start a loop, so
     * abandon that boardPlace as not valid move.
This method only call the
     * auxiliary method.
     */
    private boolean canMoveToBorder(int x, int y) {
        return canMoveToBorder(x, y, new
ArrayList<BoardPlace>());
    }

    /**
     * can move to border - auxiliary method
     */
    private boolean canMoveToBorder(int x, int y,
ArrayList<BoardPlace> path) {
        if (board.getBoardPlace(x + 1, y) != null ||
board.getBoardPlace(x, y + 1) != null) {

            return true;
```

```
        }

        return false;
    }

    /**
     * check if can move physically from x,y in to
the direction received
     *
     * By physical we mean, that the piece has
physical space to move. A piece, with
     * the NE and NO places occupied, cannot move to
N.
     */
    public boolean canPhysicallyMoveTo(int x, int y,
Direction d) {
        switch (d) {
        case N:
            return this.physicalMove(x, y,
Direction.NE, Direction.NO);
        case NE:
            return this.physicalMove(x, y,
Direction.N, Direction.SE);
        case NO:
            return this.physicalMove(x, y,
Direction.N, Direction.SO);
        case S:
            return this.physicalMove(x, y,
Direction.SE, Direction.SO);
        case SE:
            return this.physicalMove(x, y,
Direction.NE, Direction.S);
        case SO:
            return this.physicalMove(x, y,
Direction.S, Direction.NO);
        default:
            break;
        }
        return false;
    }

    private boolean physicalMove(int x, int y,
Direction d1, Direction d2) {
```

```
        Point pt1 = getTarget(currentPiece.getX(),
currentPiece.getY(), d1);
        Point pt2 = getTarget(currentPiece.getX(),
currentPiece.getY(), d2);
        if (pt1 == null || pt2 == null)
            return false;
        Piece p1 = board.getPiece((int) pt1.getX(),
(int) pt1.getY());
        Piece p2 = board.getPiece((int) pt2.getX(),
(int) pt2.getY());
        if (p1 != null && p2 != null) {
            return false;
        }

        return true;
    }

    private Point getTarget(int x, int y, Direction
d) {
        int auxX = 0, auxY = 0;
        Point p = null;
        switch (d) {
        case N:
            if (y == 0)
                return p;
            auxX = x;
            auxY = y - 1;
            break;

        case NE:
            if (x % 2 == 0) {
                if (x == Board.DIMX || y == 0)
                    return p;
                auxX = x + 1;
                auxY = y - 1;
            } else {
                if (x == Board.DIMX)
                    return p;
                auxX = x + 1;
                auxY = y;
            }
            break;
        case NO:
            if (x % 2 == 0) {
                if (x == 0 || y == 0)
```

```
                    return p;
                auxX = x - 1;
                auxY = y - 1;
            } else {
                if (x == 0)
                    return p;
                auxX = x - 1;
                auxY = y;
            }
            break;
        case S:
            if (y == Board.DIMY)
                return p;
            auxX = x;
            auxY = y + 1;
            break;
        case SE:
            if (x % 2 == 0) {
                if (x == Board.DIMX)
                    return p;
                auxX = x + 1;
                auxY = y;
            } else {
                if (x == Board.DIMX || y ==
Board.DIMY)
                    return p;
                auxX = x + 1;
                auxY = y + 1;
            }
            break;
        case SO:
            if (x % 2 == 0) {
                if (x == 0)
                    return p;
                auxX = x - 1;
                auxY = y;
            } else {
                if (x == 0 || y == 0)
                    return p;
                auxX = x - 1;
                auxY = y + 1;
            }
            break;
        default:
            break;
```

```
        }
        p = new Point(auxX, auxY);
        return p;
    }

    /**
     * set status info in label status
     */
    public void setStatusInfo(String str) {
        lb_message.setText(str);
    }

    /**
     * Move the received piece unconditionally from
its position to target position
     * with: remPiece and addPiece.
     */
    public void moveUnconditional(Piece p, int x, int
y) {
        board.remPiece(p);
        board.addPiece(p, x, y);
        p.setXY(x, y);
    }

    /**
     * check if end of game - update playerAWon
and/or playerBWon states
     */
    private boolean checkFinishGame() {
        if (endOfGame) {
            if (isPlayerAToPlay)
                playerBData.setPlayerWon(true);
            else
                playerAData.setPlayerWon(true);
            return true;

        } else {
            if (checkFinishGame(true) &&
checkFinishGame(false)) {
                lb_message.setText("Oh my God!! It's
a Draw!!!");
                JOptionPane.showMessageDialog(this,
"Oh my God!! It's a Draw!!", "DRAW!",
                        JOptionPane.INFORMATION_MESSA
GE, iconMedium);
```

```
                return true;
            } else if (checkFinishGame(true)) {
                playerBData.setPlayerWon(true);
                return true;
            } else if (checkFinishGame(false)) {
                playerAData.setPlayerWon(true);
                return true;
            }
        }
        return false;
    }

    /**
     * check if queen of received player is
surrounded
     */
    private boolean checkFinishGame(boolean playerA)
{
        if (playerA) {
            return
this.checkSurroundPos(playerAData);
        } else {
            return
this.checkSurroundPos(playerBData);
        }
    }

    private boolean checkSurroundPos(PlayerData
player) {
        Piece pt = player.getQueenBee();
        if (pt == null)
            return false;
        int x = pt.getX();
        int y = pt.getY();
        boolean emptyNeib = false;
        for (Direction d : Direction.values()) {
            Point p = Board.getNeighbourPoint(x, y,
d);
            if (p == null)
                continue;
            int k = (int) p.getX();
            int j = (int) p.getY();
            if (board.getPiece(k, j) == null) {
                emptyNeib = true;
                break;
```

```
            }
        }
        return !emptyNeib;
    }

    /**
     * do end of game actions
     */
    private void doFinishGameActions() {
        boolean checkWinner =
playerAData.playerWon();
        String winningPlayer = checkWinner ? "A" :
"B";
        int n = JOptionPane.showConfirmDialog(this,
                "Congratulations Player " +
winningPlayer + "\n Do you want to enter your name?",
"Winner Panel",
                JOptionPane.YES_NO_OPTION,
JOptionPane.QUESTION_MESSAGE, iconMedium);

        if (n == JOptionPane.YES_OPTION) {
            String s = (String)
JOptionPane.showInputDialog(this, "Please enter your
name", "Player Name",
                    JOptionPane.PLAIN_MESSAGE,
iconMedium, null, null);
            if (s != null)
                highScore.addScore(s, (checkWinner ?
playerAData.getNumberOfMoves() :
playerBData.getNumberOfMoves()));
            else
                highScore.addScore("Unknown",
                        (checkWinner ?
playerAData.getNumberOfMoves() :
playerBData.getNumberOfMoves()));
        }

        viewScores();
        lb_message.setText("Winner: Player " +
winningPlayer);
        enableControlButtons(false);
        bn_newGame.setVisible(true);
        bn_newGame.setEnabled(true);
    }
```

```
    /**
     * change state of general buttons
     */
    private void enableControlButtons(boolean enable)
{
        Component[] container =
controlPanel.getComponents();
        Component[] playerAContainer =
playerAData.getSidePanel().getComponents();
        Component[] playerBContainer =
playerBData.getSidePanel().getComponents();

        for (Component c : container) {
            if (c instanceof JPanel) {
                JPanel jp = (JPanel) c;
                Component[] buttons =
jp.getComponents();
                for (Component x : buttons) {
                    if (x instanceof JLabel)
                        continue;
                    x.setEnabled(enable);
                }
            }
        }
        this.deactivateLabels(playerBContainer,
enable);
        this.deactivateLabels(playerAContainer,
enable);
        board.resetBoard();
        board.repaint();
    }

    private void deactivateLabels(Component[]
component, boolean enable) {
        for (Component c : component) {
            if (c instanceof JPanel) {
                JPanel jp = (JPanel) c;
                Component[] piecesLabel =
jp.getComponents();
                for (Component x : piecesLabel) {
                    x.setEnabled(enable);
                    HiveLabel z = (HiveLabel) x;
                    z.deactivate();
                }
            }
```

```
            c.setEnabled(enable);
        }
    }

    /**
     * move hive UP, if it can be moved
     */
    private void moveHiveUp() {
        for (int i = 0; i < Board.DIMY; i++) {
            for (int j = 0; j < Board.DIMX; j++) {
                BoardPlace bp =
board.getBoardPlace(j, i);
                if (i == 0 && bp.getPiece() != null)
{
                    lb_message.setText("Can't move
up! Upper limit of map reached");
                    return;
                }
                board.getBoardPlace(j,
i).migrateTo(Direction.N);
            }
        }
    }

    /**
     * move hive DOWN, if it can
     */
    private void moveDown() {
        for (int i = (Board.DIMY - 1); i >= 0; i--) {
            for (int j = 0; j < Board.DIMX; j++) {
                BoardPlace bp =
board.getBoardPlace(j, i);
                if (i == Board.DIMY - 1 &&
bp.getPiece() != null) {
                    lb_message.setText("Can't move
down! Lower limit of map reached");
                    return;
                }
                board.getBoardPlace(j,
i).migrateTo(Direction.S);
            }
        }
    }

    /**
```

```
     * move hive NO, if it can
     */
    private void moveNO() {
        boolean canMove = true;
        for (int i = 0; i < Board.DIMX; i = i + 2) {
            BoardPlace bp = board.getBoardPlace(i,
0);
            if (bp.getPiece() != null) {
                canMove = false;
                break;
            }
        }
        for (int j = 0; j < Board.DIMX; j++) {
            for (int i = 0; i < Board.DIMY; i++) {
                BoardPlace bp =
board.getBoardPlace(j, i);
                if ((j == 0 && bp.getPiece() != null)
|| !canMove) {
                    lb_message.setText("Can't move
NO! Northwestern limit of map reached");
                    return;
                }
                board.getBoardPlace(j,
i).migrateTo(Direction.NO);
            }
        }
    }

    /**
     * move hive NE, if it can
     */
    private void moveNE() {
        boolean canMove = true;
        for (int i = (Board.DIMX - 1); i >= 0; i = i
- 2) {
            BoardPlace bp = board.getBoardPlace(i,
0);
            if (bp.getPiece() != null) {
                canMove = false;
            }
        }
        for (int j = (Board.DIMX - 1); j >= 0; j--) {
            for (int i = 0; i < Board.DIMY; i++) {
                BoardPlace bp =
board.getBoardPlace(j, i);
```

```
                if ((j == (Board.DIMX - 1) &&
bp.getPiece() != null) | !canMove) {
                    lb_message.setText("Can't move
NE! Northeastern limit of map reached");
                    return;
                }
                board.getBoardPlace(j,
i).migrateTo(Direction.NE);
            }
        }
    }

    /**
     * move hive SO, if it can
     */
    private void moveSO() {
        boolean canMove = true;
        for (int i = 1; i < Board.DIMX; i = i + 2) {
            BoardPlace bp = board.getBoardPlace(i,
Board.DIMY - 1);
            if (bp.getPiece() != null) {
                canMove = false;
            }
        }
        for (int i = (Board.DIMY - 1); i >= 0; i--) {
            for (int j = 0; j < Board.DIMX; j++) {
                BoardPlace bp =
board.getBoardPlace(j, i);
                if ((j == 0 && bp.getPiece() != null)
|| !canMove) {
                    lb_message.setText("Can't move
SO! Southwestern limit of map reached");
                    return;
                }
                board.getBoardPlace(j,
i).migrateTo(Direction.SO);
            }
        }
    }

    /**
     * move hive SE, if it can
     */
    private void moveSE() {
        boolean canMove = true;
```

```
        for (int i = (Board.DIMX - 2); i >= 0; i = i
- 2) {
            BoardPlace bp = board.getBoardPlace(i,
Board.DIMY - 1);
            if (bp.getPiece() != null) {
                canMove = false;
            }
        }
        for (int i = (Board.DIMY - 1); i >= 0; i--) {
            for (int j = (Board.DIMX - 1); j >= 0; j-
-) {
                BoardPlace bp =
board.getBoardPlace(j, i);
                if ((j == Board.DIMX - 1 &&
bp.getPiece() != null) || !canMove) {
                    lb_message.setText("Can't move
SE! Southeastern limit of map reached");
                    return;
                }
                board.getBoardPlace(j,
i).migrateTo(Direction.SE);
            }
        }
    }
}
```

Class Board

```
package tps.tp4;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Point;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.util.ArrayList;
import java.util.List;

import javax.swing.JPanel;

import tps.tp4.Game.Direction;
import tps.tp4.pieces.Piece;

public class Board extends JPanel {
    private static final long serialVersionUID =
1L;

    // background color for the board
    static Color BOARDBACKGROUNDCOLOR = new
Color(0xC9F76F);

    // number of cells in the board
    public static final int DIMX = 31;
    public static final int DIMY = 15;

    // the board data - array 2D of BoardPlaces
    private BoardPlace[][] board;

    // game reference
    private Game game;

    // references for the two PlayerData
    private PlayerData playerAData, playerBData;

    // Font for the pieces
    private Font piecesFont;

    // methods
================================================
```

```
    /**
     * constructor
     *
     * @param fontPieces
     */
    public Board(Game game, Font piecesFont) {
        this.game = game;
        this.piecesFont = piecesFont;
        playerAData = game.getPlayerData(true);
        playerBData = game.getPlayerData(false);
        this.board = new BoardPlace[DIMX][DIMY];
        setPreferredSize(new Dimension(600, 400));
        initBoard();
    }

    /**
     * Create the board places for pieces
     */
    private void initBoard() {
        setBackground(BOARDBACKGROUNDCOLOR);
        setFont(piecesFont);
        for (int y = 0; y < DIMY; y++) {
            for (int x = 0; x < DIMX; x++) {
                board[x][y] = new BoardPlace(this,
x, y);

            }
        }
        MouseListener ml = new MouseListener() {

            @Override
            public void mousePressed(MouseEvent e)
{

            }

            @Override
            public void mouseReleased(MouseEvent e)
{

            }

            @Override
            public void mouseEntered(MouseEvent e)
{
```

```
            }

            @Override
            public void mouseExited(MouseEvent e) {

            }

            @Override
            public void mouseClicked(MouseEvent e)
{
                int b = e.getButton();
                switch (b) {
                case MouseEvent.BUTTON1:
                    int posX = e.getX();
                    int posY = e.getY();
                    clickOnBoard(posX, posY);
                    break;
                case MouseEvent.BUTTON2:
                    break;
                case MouseEvent.BUTTON3:
                    break;
                default:
                    break;
                }
            }

        };
        addMouseListener(ml);

    }

    /**
     * method called by the mouseListener of the
board. Should check if the x, y
     * received is inside any of the polygons. In
affirmative case should call
     * game.clickOnBoard with the (x, y) of the
BoardPlace clicked
     */
    private void clickOnBoard(int xPix, int yPix) {
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length;
j++) {
```

```
                if
(board[i][j].isInsideBoardPlace(xPix, yPix)) {
                    game.clickOnBoard(i, j);
                    return;
                }
            }
        }

    }

    /**
     * clears the board data - clear all the pieces
on board
     */
    public void resetBoard() {
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length;
j++) {
                board[i][j].clear();
            }
        }
        playerAData.setNumberOfPiecesOnBoard(0);
        playerBData.setNumberOfPiecesOnBoard(0);
    }

    /**
     * sets one boardPlace selected state
     */
    public void setSelXY(int x, int y, boolean
selectedState) {
        board[x][y].setSelected(selectedState);
    }

    /**
     * draw the board - call the paintComponent for
the super and for each one of
     * the BoardPlaces
     */
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        for (int y = 0; y < DIMY; y++) {
            for (int x = 0; x < DIMX; x++) {
                board[x][y].paintComponent(g);
            }
        }
```

```
    }

    /**
     * get the neighbor point starting from x,y and
going in the d direction. If the
     * point doesn't exist the method returns null.
     */
    public static Point getNeighbourPoint(int x,
int y, Direction d) {
        Point p = new Point(x, y);

        // IMPORTANT NOTE: as the move depends on
X, we must work on Y first
        switch (d) {
        case N:
            if (y == 0)
                return null;
            p.y--;
            break;
        case NE:
            // y first
            if (p.x % 2 == 0) {
                if (y == 0)
                    return null;
                p.y--;
            }
            // then x
            if (x == Board.DIMX - 1)
                return null;
            p.x++;
            break;
        case SE:
            if (p.x % 2 == 1) {
                if (y == Board.DIMY - 1)
                    return null;
                p.y++;
            }
            if (x == Board.DIMX - 1)
                return null;
            p.x++;
            break;
        case S:
            if (y == Board.DIMY - 1)
                return null;
            p.y++;
```

```
            break;
        case SO:
            if (p.x % 2 == 1) {
                if (y == Board.DIMY - 1)
                    return null;
                p.y++;
            }
            if (x == 0)
                return null;
            p.x--;
            break;
        case NO:
            if (p.x % 2 == 0) {
                if (y == 0)
                    return null;
                p.y--;
            }
            if (x == 0)
                return null;
            p.x--;
            break;
        }
        return p;
    }

    /**
     * returns the (tail) piece on board[x][y]
     */
    public Piece getPiece(int x, int y) {
        if (isInside(x, y))
            return board[x][y].getPiece();
        else
            return null;
    }

    /**
     * returns the BoardPlace at board[x][y]
     */
    public BoardPlace getBoardPlace(int x, int y) {
        return board[x][y];
    }

    /**
     * add a piece (on tail) on the BoardPlace x,y.
Should increase the
```

```
     * numberOfPiecesOnBoard of the player that own
the piece. Any change to the
     * board should call the repaint() method.
Every piece on board should keep its
     * BoardPlace coordinates on board.
     */
    public void addPiece(Piece p, int x, int y) {
        board[x][y].addPiece(p);
        if (p.isFromPlayerA())
            playerAData.incNumberOfPiecesOnBoard();
        else
            playerBData.incNumberOfPiecesOnBoard();
        p.setXY(x, y);
    }

    /**
     * Removes the piece if this piece is on tail
on its BoardPlace. Should adjust
     * numberOfPiecesOnBoard from its owner
     */
    public boolean remPiece(Piece p) {
        int x = p.getX(), y = p.getY();
        if (p.isFromPlayerA())
            playerAData.decNumberOfPiecesOnBoard();
        else
            playerBData.decNumberOfPiecesOnBoard();
        return board[x][y].remPiece(p);
    }

    /**
     * check if staring from x,y is just one hive.
The number of adjacent pieces
     * should be all the pieces on board. Can be
used an ArrayList to collect the
     * pieces.
     */
    public boolean justOneHive(int x, int y) {
        int nPieces =
playerAData.getNumberOfPiecesOnBoard() +
playerBData.getNumberOfPiecesOnBoard();
        List<Piece> l = new ArrayList<Piece>();
        this.getPiecesFromThisPoint(x, y, l);
        return l.size() == nPieces;
    }
```

```
    /**
     * Get all the pieces that are connected with
the x, y received, and put them on
     * the List received.
     */
    private void getPiecesFromThisPoint(int x, int
y, List<Piece> pieces) {
        ArrayList<Piece> pieceList =
board[x][y].getList();
        for (Piece p : pieceList) {
            pieces.add(p);
        }
        for (Direction d : Direction.values()) {
            Point point = getNeighbourPoint(x, y,
d);
            if(point == null) continue;
            Piece p2 =
board[point.x][point.y].getPiece();
            if (p2 != null && !pieces.contains(p2))
{
                getPiecesFromThisPoint(point.x,
point.y, pieces);
            }
        }
    }

    /**
     *
     * @param x
     * @param y
     * @return
     */
    public boolean isInside(int x, int y) {
        if (x >= 0 && x < DIMX && y >= 0 && y <
DIMY)
            return true;
        return false;
    }

};
```

Class BoardPlace

```
package tps.tp4;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Polygon;
import java.util.ArrayList;
import java.util.Deque;
import java.util.LinkedList;

import tps.tp4.Game.Direction;
import tps.tp4.pieces.Piece;

public class BoardPlace {

    private static Color PIECEBACKGROUNDCOLOR = new
Color(0x9CCF3A);
    private static Color PIECESELECTIONCOLOR =
Color.RED;

    public static int STARTXY = 5;

    // the place to have the pieces on this board
place
    // pieces must be added at the tail, and the only
accessible piece must be
    // the tail piece
    ArrayList<Piece> pieces = new ArrayList<Piece>();

    // is selected or not
    private boolean selected = false;

    // board reference
    private Board board;

    // the board place coordinates
    int x, y;

    // the polygon for this board place
    Polygon polygon = new Polygon();

    // the selection polygon for this board place
    Polygon selPolygon = new Polygon();

    // the base xy from the board for this place
```

```
    private int baseX;
    private int baseY;

    // Methods
=========================================

    /**
     * constructor
     */
    public BoardPlace(Board board, int x, int y) {
        this.board = board;
        this.x = x;
        this.y = y;

        // base data for polygons
        baseX = STARTXY + (int) (x * Piece.DIMPIECE *
0.75);
        baseY = STARTXY + (int) (y * Piece.DIMPIECE);

        if (x % 2 == 1) {
            baseY += Piece.DIMPIECE / 2;
        }

        // build polygon for this board place
        polygon.addPoint(baseX + Piece.DIMPIECE / 4 +
1, baseY + 1);
        polygon.addPoint(baseX + (Piece.DIMPIECE * 3)
/ 4 - 1, baseY + 1);

        polygon.addPoint(baseX + Piece.DIMPIECE - 1,
baseY + Piece.DIMPIECE / 2);
        polygon.addPoint(baseX + (Piece.DIMPIECE * 3)
/ 4 - 1, baseY
                + Piece.DIMPIECE - 1);
        polygon.addPoint(baseX + Piece.DIMPIECE / 4 +
1, baseY + Piece.DIMPIECE
                - 1);
        polygon.addPoint(baseX + 1, baseY +
Piece.DIMPIECE / 2);

        // build selected polygon
        selPolygon.addPoint(baseX + Piece.DIMPIECE /
4, baseY - 1);
        selPolygon.addPoint(baseX + (Piece.DIMPIECE *
3) / 4, baseY - 1);
```

```
        selPolygon.addPoint(baseX + Piece.DIMPIECE,
baseY + Piece.DIMPIECE / 2);
        selPolygon.addPoint(baseX + (Piece.DIMPIECE *
3) / 4, baseY
                + Piece.DIMPIECE);
        selPolygon.addPoint(baseX + Piece.DIMPIECE /
4, baseY + Piece.DIMPIECE);
        selPolygon.addPoint(baseX, baseY +
Piece.DIMPIECE / 2);

    }

    public ArrayList<Piece> getList(){
        return pieces;
    }

    public int getNumPieces() {
        return pieces.size();
    }

    /**
     * get the tail piece - the others are not
accessible
     */
    public Piece getPiece() {
        if (pieces.size() == 0)
            return null;

        return pieces.get(pieces.size() - 1);
    }

    /**
     * Add piece to tail
     */
    public void addPiece(Piece p) {
        pieces.add(p);
    }

    /**
     * remove piece P if it is on tail
     */
    public boolean remPiece(Piece p) {
        if(pieces.get(pieces.size() - 1).equals(p)) {
            pieces.remove(pieces.size() - 1);
```

```
            return true;
        }
        return false;
    }

    /**
     * clear all the pieces on this boardPlace
     */
    public void clear() {
        for (int i = 0; i < pieces.size();i++) {
            pieces.remove(i);
        }
    }

    /**
     * set selected state
     */
    public void setSelected(boolean selected) {
        this.selected = selected;
        board.repaint();
    }

    /**
     * get selected state
     */
    public boolean isSelected() {
        return selected;
    }

    /**
     * equals, two BoardPlaces are equal if they have
the same x and y
     */
    public boolean equals(Object o) {
        return this.x == ((BoardPlace)o).x && this.y
== ((BoardPlace)o).y;
    }

    /**
     * to be viewed in debug watch
     */
    public String toString() {
        return "(" + x + "," + y + ")";
    }
```

```
    /**
     * Migrate the state of this board place 1
position to the neighbor in the
     * received direction. To be used is move HIVE
up, down, NO, ....
     */
    public void migrateTo(Direction d) {
        if(pieces.size() ==  0)
            return;
        switch(d) {
            case N:
                BoardPlace bpn =
board.getBoardPlace(x, y-1);
                bpn.setSelected(this.isSelected());
                this.swapPieces(bpn);
                break;
            case NO:
                BoardPlace bpno;
                if(x % 2 == 0)
                    bpno = board.getBoardPlace(x-1,
y-1);
                else
                    bpno = board.getBoardPlace(x-1,
y);
                bpno.setSelected(this.isSelected());
                this.swapPieces(bpno);
                break;
            case NE:
                BoardPlace bpne;
                if(x % 2 == 0)
                    bpne = board.getBoardPlace(x + 1,
y - 1);
                else
                    bpne = board.getBoardPlace(x + 1,
y);
                bpne.setSelected(this.isSelected());
                this.swapPieces(bpne);
                break;
            case S:
                BoardPlace bps =
board.getBoardPlace(x, y + 1);
                bps.setSelected(this.isSelected());
                this.swapPieces(bps);
                break;
            case SO:
```

```
                BoardPlace bpso;
                if(x % 2 == 0)
                    bpso = board.getBoardPlace(x-1,
y);
                else
                    bpso = board.getBoardPlace(x-1, y
+ 1);
                bpso.setSelected(this.isSelected());
                this.swapPieces(bpso);
                break;
            case SE:
                BoardPlace bpse;
                if(x % 2 == 0)
                    bpse = board.getBoardPlace(x+1,
y);
                else
                    bpse = board.getBoardPlace(x+1,
y+1);
                bpse.setSelected(this.isSelected());
                this.swapPieces(bpse);
                break;
            default:
                break;
        }
        board.repaint();

    }

    private void swapPieces(BoardPlace bp) {
        if(pieces.size() == 1) {
            Piece p = this.getPiece();
            this.remPiece(p);
            bp.addPiece(p);
            p.setXY(bp.x, bp.y);
        } else {
            Deque<Piece> stack = new
LinkedList<Piece>();
            while(pieces.size() > 0) {
                Piece p = this.getPiece();
                this.remPiece(p);
                p.setXY(bp.x, bp.y);
                stack.add(p);
            }
            while(!stack.isEmpty()) {
                bp.addPiece(stack.pollLast());
```

```
            }
        }
    }

    /**
     * Paint this boardPiece - if it doesn't have any
piece we should the draw
     * polygon in background color
     */
    public void paintComponent(Graphics g) {

        if (getPiece() == null) {
            // draw empty board place
            g.setColor(PIECEBACKGROUNDCOLOR);
            g.fillPolygon(polygon);
        } else {
            g.setColor(getPiece().getColor());
            g.fillPolygon(polygon);
            if(getPiece().isFromPlayerA())
g.setColor(Color.BLACK);
            else g.setColor(Color.LIGHT_GRAY);
            g.drawString(String.valueOf(getPiece().ge
tClass().getSimpleName().charAt(0)), baseX+7,
baseY+17);

        }
        // if selected, draw selection
        if (isSelected()) {
            g.setColor(PIECESELECTIONCOLOR);
            g.drawPolygon(polygon);
        }
    }

    /**
     * check if x,y received is inside the polygon of
this boardPlace - uses the
     * contains method from polygon
     */
    public boolean isInsideBoardPlace(int x, int y) {
        return polygon.contains(x, y);
    }

}
```

Class PlayerData

```
package tps.tp4;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.BorderFactory;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingConstants;
import javax.swing.border.Border;

import tps.tp4.pieces.Ant;
import tps.tp4.pieces.Beetle;
import tps.tp4.pieces.Grasshopper;
import tps.tp4.pieces.Ladybug;
import tps.tp4.pieces.Mosquito;
import tps.tp4.pieces.Piece;
import tps.tp4.pieces.PillBug;
import tps.tp4.pieces.QueenBee;
import tps.tp4.pieces.Spider;

/**
 * class that keep and control the data from one
player
 */
public class PlayerData {

    private static Color ACTIVEPLAYERCOLOR =
Color.orange;
    private static Color INACTIVEPLAYERCOLOR =
Color.gray;

    /**
     * one Queen, two Beetles, two Grasshoppers,
three Spiders, three Ants, one Mosquito, one Pillbug
and one Ladybug
     *
     * Don't change this
     */
    private final PiecesAndItsNumber[] ListaDePecas =
new PiecesAndItsNumber[] {
```

```
            new PiecesAndItsNumber(PType.QUEENBEE,
1), new PiecesAndItsNumber(PType.BEETLE, 2),
            new PiecesAndItsNumber(PType.GRASHOPPER,
2), new PiecesAndItsNumber(PType.SPIDER, 3),
            new PiecesAndItsNumber(PType.ANT, 3),
            new PiecesAndItsNumber(PType.MOSQUITO,
1), new PiecesAndItsNumber(PType.LADYBUG, 1), new
PiecesAndItsNumber(PType.PILLBUG, 1) };

    private JPanel sidePanel;
    private JLabel movesLabel;
    private JLabel playerLabel;
    private HiveLabel queenBeeLabel;
    private QueenBee queenBee;

    private int numberOfPiecesOnBoard;
    private int numberOfMoves;

    private boolean playerWon;

    /**
     * auxiliary class
     */
    private class PiecesAndItsNumber {
        PType tipo;
        int nPecas;

        public PiecesAndItsNumber(PType tipo, int
nPecas) {
            this.tipo = tipo;
            this.nPecas = nPecas;
        }

        public PType getTipo() {
            return tipo;
        }

        public int getnPecas() {
            return nPecas;
        }
    }

    /**
     * Constructor - should build the side panel for
the player
```

```
     */
    public PlayerData(Game game, boolean isPlayerA) {
        this.init(isPlayerA);

        JPanel piecesPanel = new JPanel(new
GridLayout(14, 1, 0, 0));
        for (PiecesAndItsNumber p : ListaDePecas) {
            Piece addedPiece =
p.getTipo().createNew(game, isPlayerA);
            for (int i = 0; i < p.getnPecas(); i++) {
                HiveLabel pieceLabel = new
HiveLabel(addedPiece, game);
                pieceLabel.setText(pieceLabel.getPiec
e().getName());
                pieceLabel.setHorizontalAlignment(Swi
ngConstants.CENTER);
                pieceLabel.setPreferredSize(new
Dimension(150, 20));
                pieceLabel.setForeground(Color.WHITE)
;
                pieceLabel.setBackground(pieceLabel.g
etPiece().getColor());
                pieceLabel.setOpaque(true);
                pieceLabel.addMouseListener(new
MouseAdapter() {
                    @Override
                    public void
mouseClicked(MouseEvent e) {
                        game.clickOnPieceLabelOnSideP
anel(pieceLabel);
                    }
                });
                if(pieceLabel.getPiece().getName().eq
ualsIgnoreCase("QueenBee")) {
                    queenBeeLabel = pieceLabel;
                }
                piecesPanel.add(pieceLabel);
            }
        }
        sidePanel.add(piecesPanel);

        String bjoras =
String.valueOf(numberOfMoves);
        movesLabel = new JLabel(bjoras,
SwingConstants.CENTER);
```

```
        movesLabel.setOpaque(true);
        movesLabel.setBackground(Color.GREEN);
        movesLabel.setPreferredSize(new
Dimension(150, 20));
        sidePanel.add(movesLabel);

    }

    /**
     * Initializes the counters and the labels
     */
    public void init(boolean playerIsActive) {
        numberOfMoves = 0;
        numberOfPiecesOnBoard = 0;
        queenBee = null;
        queenBeeLabel = null;
        sidePanel = new JPanel();
        playerWon = false;

        Dimension dim = new Dimension(150, 20);
        String player = playerIsActive == true ? "A"
: "B";
        playerLabel = new JLabel("Player " + player,
SwingConstants.CENTER);
        playerLabel.setPreferredSize(dim);
        playerLabel.setOpaque(true);
        if(playerIsActive)playerLabel.setBackground(A
CTIVEPLAYERCOLOR);
        else
playerLabel.setBackground(INACTIVEPLAYERCOLOR);
        sidePanel.add(playerLabel);

        JLabel playerColor = new JLabel("Player
Color", SwingConstants.CENTER);
        playerColor.setPreferredSize(dim);
        playerColor.setForeground(Color.WHITE);
        playerColor.setBackground(Game.getColorFromPl
ayer(playerIsActive));
        playerColor.setOpaque(true);
        sidePanel.add(playerColor);

    }

    /**
     * get side panel
```

```
     */
    JPanel getSidePanel() {
        return sidePanel;
    }

    /**
     * get number of moves of this player
     */
    int getNumberOfMoves() {
        return numberOfMoves;
    }

    /**
     * increment number of moves of this player
     */
    void incNumberOfMoves() {
        numberOfMoves++;
        displayNumberOfMoves();
    }

    /**
     * get Queen Bee reference of this player
     */
    QueenBee getQueenBee() {
        return queenBee;
    }

    /**
     * sets the number of moves ...
     */
    void setNumberOfMoves(int n) {
        this.numberOfMoves = n;
        this.displayNumberOfMoves();
    }

    /**
     * get the number of pieces on board ...
     */
    int getNumberOfPiecesOnBoard() {
        return numberOfPiecesOnBoard;
    }

    /**
     * set the number of pieces on board ...
     */
```

```
    void setNumberOfPiecesOnBoard(int np) {
        this.numberOfPiecesOnBoard = np;
    }

    /**
     * increases the number of pieces on board ...
     */
    void incNumberOfPiecesOnBoard() {
        this.numberOfPiecesOnBoard++;
    }

    /**
     * decreases the number of pieces on board ..
     */
    void decNumberOfPiecesOnBoard() {
        if (this.numberOfPiecesOnBoard > 0)
            this.numberOfPiecesOnBoard--;
    }

    /**
     * set this player background as current player
or not
     */
    public void setPlayerPanelActive(boolean active)
{
        if
(active)playerLabel.setBackground(ACTIVEPLAYERCOLOR);
        else
playerLabel.setBackground(INACTIVEPLAYERCOLOR);
    }

    public void setQueenBee(Piece queen) {
        queenBee = (QueenBee)queen;
    }

    /**
     * check if queen bee of this player is already
on board
     */
    public boolean isQueenBeeAlreadyOnBoard() {
        return queenBee != null;
    }

    /**
```

```
     * display the current number of moves in the
last label
     */
    public void displayNumberOfMoves() {
        movesLabel.setText("" + numberOfMoves);
    }

    /**
     * get the reference for the queen bee of this
player
     */
    public HiveLabel getQueenBeeLabel() {
        return queenBeeLabel;
    }

    /**
     * sets if player won
     */
    void setPlayerWon(boolean won) {
        playerWon = won;
    }

    /**
     * return true if player won
     */
    boolean playerWon() {
        return playerWon;
    }

}

/**
 * classe que suporta as labels das peï¿½as iniciais
de cada jogador
 */
class HiveLabel extends JLabel {
    private static final long serialVersionUID = 1L;
    final static Border unselBorder =
BorderFactory.createLineBorder(Color.darkGray);
    final static Border selBorder =
BorderFactory.createLineBorder(Color.white, 3);

    private Piece p;
    private Game game;
    private boolean isDeactivated = false;
```

```
    /**
     *
     */
    public HiveLabel(Piece p, Game game) {
        this.p = p;
        this.game = game;
        init();

    }

    /**
     *
     */
    public Piece getPiece() {
        return p;
    }

    /**
     *
     */
    public String toString() {
        return p.toString();
    }

    /**
     *
     */
    public void init() {
        this.setToNormal();

    }

    /**
     *
     */
    public void activate() {
        setBorder(selBorder);
    }

    /**
     *
     */
    public void setToNormal() {
```

```
        setBorder(BorderFactory.createLineBorder(p.ge
tColor()));
        this.isDeactivated = false;
    }

    /**
     *
     */
    public void deactivate() {
        setBorder(unselBorder);
        this.isDeactivated = true;
    }

    /**
     *
     */
    public boolean isDeactivated() {
        return isDeactivated;
    }

}

/**
 * enum with the several pieces and create methods
 */
enum PType {
    QUEENBEE {
        Piece createNew(Game game, boolean
isFromPlayerA) {
            return new QueenBee(game, isFromPlayerA);
        };
    },
    BEETLE {
        Piece createNew(Game game, boolean
isFromPlayerA) {
            return new Beetle(game, isFromPlayerA);
        };
    },
    GRASHOPPER {
        Piece createNew(Game game, boolean
isFromPlayerA) {
            return new Grasshopper(game,
isFromPlayerA);
        };
    },
```

```
    SPIDER {
        Piece createNew(Game game, boolean
isFromPlayerA) {
            return new Spider(game, isFromPlayerA);
        };
    },
    ANT {
        Piece createNew(Game game, boolean
isFromPlayerA) {
            return new Ant(game, isFromPlayerA);
        };
    },
    MOSQUITO {
        Piece createNew(Game game, boolean
isFromPlayerA) {
            return new Mosquito(game, isFromPlayerA);
        };
    },
    LADYBUG {
        Piece createNew(Game game, boolean
isFromPlayerA) {
            return new Ladybug(game, isFromPlayerA);
        };
    },
    PILLBUG {
        Piece createNew(Game game, boolean
isFromPlayerA) {
            return new PillBug(game, isFromPlayerA);
        };
    };

    abstract Piece createNew(Game game, boolean
isFromPlayerA);
};
```

Class HighscoreManager

```
package tps.tp4;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.Collections;

import tps.tp4.highscoreManagement.Score;
import tps.tp4.highscoreManagement.ScoreComparator;

public class HighscoreManager {
    // An arraylist of the type "score" we will use
to work with the scores inside the class
    private ArrayList<Score> scores;

    // The name of the file where the highscores
will be saved
    private static final String HIGHSCORE_FILE =
"hiveHighScore.dat";

    //Initialising an in and outputStream for
working with the file
    ObjectOutputStream outputStream = null;
    ObjectInputStream inputStream = null;


    public HighscoreManager() {
        //initialising the scores-arraylist
        scores = new ArrayList<Score>();
    }

    public ArrayList<Score> getScores() {
        loadScoreFile();
        sort();
        return scores;
    }

    private void sort() {
        ScoreComparator comparator = new
ScoreComparator();
```

```
        Collections.sort(scores, comparator);
    }

    public void addScore(String name, int score) {
        loadScoreFile();
        scores.add(new Score(name, score));
        updateScoreFile();
    }

    @SuppressWarnings("unchecked")
    public void loadScoreFile() {
        try {
            inputStream = new ObjectInputStream(new
FileInputStream(HIGHSCORE_FILE));
            scores = (ArrayList<Score>)
inputStream.readObject();
        } catch (FileNotFoundException e) {
            System.out.println("Read FNF Error: " +
e.getMessage());
        } catch (IOException e) {
            System.out.println("Read IO Error: " +
e.getMessage());
        } catch (ClassNotFoundException e) {
            System.out.println("Read CNF Error: " +
e.getMessage());
        } finally {
            try {
                if (outputStream != null) {
                    outputStream.flush();
                    outputStream.close();
                }
            } catch (IOException e) {
                System.out.println("Read IO Error:
" + e.getMessage());
            }
        }
    }

    public void updateScoreFile() {
        try {
            outputStream = new
ObjectOutputStream(new
FileOutputStream(HIGHSCORE_FILE));
            outputStream.writeObject(scores);
        } catch (FileNotFoundException e) {
```

```
            System.out.println("Update FNF Error: "
+ e.getMessage() + ", the file will be created");
        } catch (IOException e) {
            System.out.println("Update IO Error: "
+ e.getMessage());
        } finally {
            try {
                if (outputStream != null) {
                    outputStream.flush();
                    outputStream.close();
                }
            } catch (IOException e) {
                System.out.println("[Update] Error:
" + e.getMessage());
            }
        }
    }

    public String getHighscoreString() {
        String highscoreString = "";
        final int max = 10;

        ArrayList<Score> scores;
        scores = getScores();

        int i = 0;
        int x = scores.size();
        if (x > max) {
            x = max;
        }
        highscoreString = "HiveGame | Top
players\n";
        while (i < x) {
            highscoreString += (i + 1) + " - \t" +
scores.get(i).getName() + "\t\t" +
scores.get(i).getScore() + "\n";
            i++;
        }
        return highscoreString;
    }
}
```

Pieces (Código)

Piece

```
package tps.tp4.pieces;

import java.awt.Color;
import java.awt.Point;

import tps.tp4.Board;
import tps.tp4.Game;
import tps.tp4.Game.Direction;

public abstract class Piece {

    final public static int DIMPIECE = 25;

    private boolean isFromPlayerA;
    private int x = -1, y = -1;
    protected Game game;

    private String name;
    private Color color;

    /**
     * constructor
     */
    public Piece(String name, Color color, Game game,
boolean isFromPlayerA) {
        this.name = name;
        this.game = game;
        this.isFromPlayerA = isFromPlayerA;
        this.color = color;
    }

    /**
     * toString
     */
    public String toString() {
        return "The piece " + getName() + " with
color " + getColor() + " is from player A? -> " +
isFromPlayerA();
    }

    public String getName() {
```

```
        return this.name;
    }

    /**
     * get color
     */
    public Color getColor() {
        return this.color;
    }

    /**
     * get if piece if from player A or not
     */
    public boolean isFromPlayerA() {
        return this.isFromPlayerA;
    }

    /**
     * set xy
     */
    public void setXY(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /**
     * get x
     */
    public int getX() {
        return this.x;
    }

    /**
     * set y
     */
    public int getY() {
        return this.y;
    }

    /**
     * move this piece to x,y if doesn't violate the
rules
     */
    public abstract boolean moveTo(int x, int y);
```

```
    /**
     * checks if the x, y received position have one
neighbor that is not me
     */
    protected boolean haveValidNeighbour(int x, int
y) {
        for (int i = x - 1; i <= x + 1; i++) {
            for (int j = y - 1; j <= j + 1; j++) {
                if (!game.getBoard().isInside(i, j))
                    continue;
                if (x == i && y == j)
                    continue;
                if (searchPiece(i, j))
                    return true;
            }
        }
        return false;
    }

    protected boolean searchPiece(int x, int y) {
        if (game.getBoard().getPiece(x, y) != null)
            return true;
        return false;
    }

    /**
     * move one step if it is verify the rules
     */
    protected boolean moveOneCheckedStep(int x, int
y) {
        int currentX = -1, currentY = -1;
        Direction d = null;
        if(currentX < 0) {
            d = getDirection(getX(), getY(), x, y);
            Point p = Board.getNeighbourPoint(getX(),
getY(), d);
            currentX = (int)p.getX();
            currentY = (int)p.getY();
        }
        while(currentX != x && currentY != y) {
            System.out.println(game.getBoard().justOn
eHive(x, y));
            if(!game.canPhysicallyMoveTo(currentX,
currentY, d)||!game.getBoard().justOneHive(x, y)) {
                return false;
```

```
            }
            d = getDirection(currentX, currentY, x,
y);
            Point p =
Board.getNeighbourPoint(currentX, currentY, d);
            currentX = (int)p.getX();
            currentY = (int)p.getY();

        }
        return true;
    }

    /**
     * move to the destination if the move from the
current position to the destiny
     * doesn't violate the one hive rule. It can move
several steps.
     */
    protected boolean moveWithOnehiveRuleChecked(int
x, int y) {
        int originalX = getX(), originalY = getY();
        game.moveUnconditional(this, x, y);
        boolean oneHive =
game.getBoard().justOneHive(x, y);
        if(!oneHive) game.moveUnconditional(this,
originalX, originalY);
        else game.getBoard().repaint();
        return oneHive;
    }

    /**
     * get the direction from start coordinates to
destiny coordinates
     */
    protected static Direction getDirection(int
fromX, int fromY, int toX, int toY) {

        Direction d1 = null;
        if( fromX != toX || fromY != toY) {
            if(fromX % 2 == 0) {
                if(toX > fromX && toY < fromY)
                    d1 = Direction.NE;
                else if(toX > fromX)
                    d1 = Direction.SE;
                else if(toX == fromX && toY < fromY)
```

```
                    d1 = Direction.N;
                else if(toX == fromX && toY > fromY)
                    d1 = Direction.S;
                else if(toX < fromX && toY < fromY)
                    d1 = Direction.NO;
                else
                    d1 = Direction.SO;
            } else {
                if(toX > fromX && toY > fromY)
                    d1 = Direction.SE;
                else if(toX > fromX)
                    d1 = Direction.NE;
                else if(toX == fromX && toY < fromY)
                    d1 = Direction.N;
                else if(toX == fromX && toY > fromY)
                    d1 = Direction.S;
                else if(toX < fromX && toY > fromY)
                    d1 = Direction.SO;
                else
                    d1 = Direction.NO;
            }
        }
        return d1;
    }

}
```

QueenBee

```
package tps.tp4.pieces;

import java.awt.Color;
import java.awt.Point;

import tps.tp4.Board;
import tps.tp4.Game;
import tps.tp4.Game.Direction;

/**
 * QueenBee class
 */
public class QueenBee extends Piece {
    final static private Color color =
Color.yellow;

    /**
     * constructor
     */
    public QueenBee(Game game, boolean
isFromPlayerA) {
        super("QueenBee", color, game,
isFromPlayerA);
    }

    /**
     * Move this piece to x,y if doesn't violate
the rules.
     *
     * The QueenBee can move only one step. Should
not violate the one hive rule and
     * the physical possible move rule.
     *
     */
    public boolean moveTo(int x, int y) {
        if (game.getBoard().getPiece(x, y) != null)
{
            return false;
        }
        boolean reachable = false;
        // execute search for all the coordinates
        for (Direction direc : Direction.values())
{
```

```
            Point p =
Board.getNeighbourPoint(getX(), getY(), direc);
            if (p == null)
                continue;
            if (p.getX() == x && p.getY() == y) {
                reachable = true;
                break;
            }
        }
        if (!reachable)
            return false;
        Direction direc =
Piece.getDirection(getX(), getY(), x, y);

        boolean canMove =
game.canPhysicallyMoveTo(x, y, direc);
        // move if one hive rule checked
        boolean moved = false;
        if (canMove) {
            moved = moveWithOnehiveRuleChecked(x,
y);
        }

        if (moved) {
            game.moveUnconditional(this, x, y);
        }
        return moved;
    }

}
```

Beetle

```
package tps.tp4.pieces;

import java.awt.Color;
import java.awt.Point;

import tps.tp4.Board;
import tps.tp4.Game;
import tps.tp4.Game.Direction;

/**
 * Beetle class
 */
public class Beetle extends Piece {
    final static private Color color =
Color.magenta;


    /**
     * constructor
     */
    public Beetle(Game game, boolean isFromPlayerA)
{
        super("Beetle", color, game,
isFromPlayerA);
    }

    /**
     * Move this piece to x,y if doesn't violate
the rules.
     *
     * The Beetle can move only one step and be
placed on top on another piece(s).
     * Should not violate the one hive rule.
     */
    public boolean moveTo(int x, int y) {
        boolean reachable = false;
        // execute search for all the coordinates
        for (Direction direc : Direction.values())
{
            Point p =
Board.getNeighbourPoint(getX(), getY(), direc);
            if (p == null)
                continue;
            if (p.getX() == x && p.getY() == y) {
```

```
                reachable = true;
                break;
            }
        }
        if (!reachable)
            return false;

        Direction direc =
Piece.getDirection(getX(), getY(), x, y);

        boolean canMove =
game.canPhysicallyMoveTo(x, y, direc);
        // move if one hive rule checked
        boolean moved = false;
        if(canMove) {
            moved = moveWithOnehiveRuleChecked(x,
y);
        }
        if (moved) {
            game.moveUnconditional(this, x, y);
        }
        return moved;
    }
}
```

Grasshopper

```
package tps.tp4.pieces;

import java.awt.Color;
import java.awt.Point;

import tps.tp4.Board;
import tps.tp4.Game;
import tps.tp4.Game.Direction;

/**
 * Grasshopper class
 */
public class Grasshopper extends Piece {
    final static private Color color = new
Color(70, 90, 40);

    /**
     * constructor
     */
    public Grasshopper(Game game, boolean
isFromPlayerA) {
        super("Grasshopper", color, game,
isFromPlayerA);
    }

    /**
     * Move this piece to x,y if doesn't violate
the rules.
     *
     * The Grasshopper must move in strait jumps
over at least one piece (but not
     * empty places). Should not violate the one
hive rule.
     */
    public boolean moveTo(int x, int y) {
        if (game.getBoard().getPiece(x, y) != null)
{
            game.setStatusInfo("Invalid move - the
destiny must be empty");
            return false;
        }
```

```
        boolean reachable = false;
        // execute search for all the coordinates
        for (Direction direc : Direction.values())
{
            Point p =
Board.getNeighbourPoint(getX(), getY(), direc);
            if (p == null)
                continue;
            if (game.getBoard().getPiece(p.x, p.y)
!= null) {
                if (toGo(getX(), getY(), x, y,
direc)) {
                    reachable = true;
                    break;
                }

            }
        }
        if (!reachable) {
            game.setStatusInfo("OPS! Invalid move -
The "+this.getName()+" can't move to that
position");
            return false;
        }

        boolean moved =
moveWithOnehiveRuleChecked(x, y);

        if (moved) {
            game.setStatusInfo("The Piece "+
this.getName()+" moved!");
            game.moveUnconditional(this, x, y);
            return true;
        }
        return false;
    }


    private boolean toGo(int x, int y, int endX,
int endY, Direction direc){

        if(Board.getNeighbourPoint(x, y, direc) ==
null) return false;
```

```
        if(game.getBoard().getPiece(x, y) == null)
{
            if(x == endX && y == endY) return true;

            else return false;

        }
        if(toGo((int)Board.getNeighbourPoint(x, y,
direc).x, Board.getNeighbourPoint(x, y, direc).y,
endX, endY, direc)) return true;

        return false;
    }
}
```

Spider

```
package tps.tp4.pieces;

import java.awt.Color;
import java.awt.Point;

import tps.tp4.Board;
import tps.tp4.Game;
import tps.tp4.Game.Direction;

/**
 * Spider class
 */
public class Spider extends Piece {
    final static private Color color = new
Color(0xA62D00);

    /**
     * constructor
     */
    public Spider(Game game, boolean isFromPlayerA)
{
        super("Spider", color, game,
isFromPlayerA);
    }

    /**
     * Move this piece to x,y if doesn't violate
the rules.
     *
     * The Spider must move exactly 3 different
steps. Should not violate the one
     * hive rule and the physical possible move
rule in each step.
     */
    public boolean moveTo(int x, int y) {

        // execute search for all the coordinates,
with limit of 3 steps
        boolean reachable = false;
        for (Direction direc : Direction.values())
{
```

```
            Point p =
Board.getNeighbourPoint(getX(), getY(), direc);
            if (p == null)
                continue;
            if (toGo(p.x, p.y, x, y, 3, direc)) {
                reachable = true;
                break;
            }
        }

        if (!reachable) {
            return false;
        }

        // move if one hive rule checked
        boolean moved =
moveWithOnehiveRuleChecked(x, y);

        if (moved) {
            game.moveUnconditional(this, x, y);
        }
        return moved;
    }

    /**
     * Find if current Spider can move in 3 steps
to the final position. For each
     * step it decreases the value toMove. If it is
zero that means and is not the
     * destiny, that means that the Spider doesn't
arrived at the destination by
     * this path, We must try all the paths.
     */
    private boolean toGo(int x, int y, int endX,
int endY, int move, Direction lastDirec) {
        if (move > 3)
            return false;
        int auxX = x, auxY = y;
        while (move != 0) {

            boolean canMove =
game.canPhysicallyMoveTo(auxX, auxY, lastDirec);
            // move if one hive rule checked
            boolean moved = false;
            if (canMove) {
```

```
                moved =
game.getBoard().justOneHive(auxX, auxY);
            }
            if (!moved)
                return false;
            move--;

            Point p = Board.getNeighbourPoint(auxX,
auxY, lastDirec);
            if (p == null)
                continue;
            if (auxX == endX && auxY == endY)
                move = 0;
            else {
                auxX = p.x;
                auxY = p.y;
            }

        }
        if (move == 0 && auxX == endX && auxY ==
endY)
            return true;
        return false;
    }
}
```

Ant

```
package tps.tp4.pieces;

import java.awt.Color;

import tps.tp4.Game;
import tps.tp4.Game.Direction;

/**
 * Ant class
 */
public class Ant extends Piece {
    final static private Color color = Color.blue;

    /**
     * constructor
     */
    public Ant(Game game, boolean isFromPlayerA) {
        super("Ant", color, game, isFromPlayerA);
    }

    /**
     * move this piece to x,y if doesn't violate
the rules
     *
     * The Ant must move any numbers of steps.
Should not violate the one hive rule
     * and the physical possible move rule in each
step.
     */
    public boolean moveTo(int x, int y) {
        if(game.getBoard().getPiece(x, y) != null)
return false;
        boolean canMove = false;
        Direction direc =
Piece.getDirection(getX(), getY(), x, y);
        if(game.canPhysicallyMoveTo(x, y, direc))
canMove = true;
        boolean moved = false;
        if(canMove) moved =
moveWithOnehiveRuleChecked(x, y);
```

```
        if (moved) {
            game.moveUnconditional(this, x, y);
        }
        return moved;

    }

}
```

Mosquito

```
package tps.tp4.pieces;

import java.awt.Color;
import java.awt.Point;

import tps.tp4.Board;
import tps.tp4.Game;
import tps.tp4.Game.Direction;

/**
 * Beetle class
 */
public class Mosquito extends Piece {
    final static private Color color = Color.gray;

    /**
     * constructor
     */
    public Mosquito(Game game, boolean
isFromPlayerA) {
        super("Mosquito", color, game,
isFromPlayerA);
    }

    /**
     * Move this piece to x,y if doesn't violate
the rules.
     *
     * The Beetle can move only one step and be
placed on top on another
     * piece(s). Should not violate the one hive
rule.
     */
    public boolean moveTo(int x, int y) {
        if (game.getBoard().getPiece(x, y) != null)
{
            return false;
        }
        boolean reachable = false;
        // execute search for all the coordinates
        for (Direction direc : Direction.values())
{
            Point p =
Board.getNeighbourPoint(getX(), getY(), direc);
```

```
            if (p == null)
                continue;
            if (p.getX() == x && p.getY() == y) {
                reachable = true;
                break;
            }
        }
        if (!reachable)
            return false;
        Direction direc =
Piece.getDirection(getX(), getY(), x, y);

        boolean canMove =
game.canPhysicallyMoveTo(x, y, direc);
        // move if one hive rule checked
        boolean moved = false;
        if (canMove) {
            moved = moveWithOnehiveRuleChecked(x,
y);
        }

        if (moved) {
            game.moveUnconditional(this, x, y);
        }
        return moved;
    }
}
```

Ladybug

```
package tps.tp4.pieces;

import java.awt.Color;
import java.awt.Point;

import tps.tp4.Board;
import tps.tp4.Game;
import tps.tp4.Game.Direction;

/**
 * Beetle class
 */
public class Ladybug extends Piece {
    final static private Color color = Color.red;

    /**
     * constructor
     */
    public Ladybug(Game game, boolean
isFromPlayerA) {
        super("Ladybug", color, game,
isFromPlayerA);
    }


    public boolean moveTo(int x, int y) {

        boolean reachable = false;
        // execute search for all the coordinates
        for (Direction direc : Direction.values())
{
            Point p =
Board.getNeighbourPoint(getX(), getY(), direc);
            if (p == null)
                continue;
            if (game.getBoard().getPiece(p.x, p.y)
!= null) {
                if (toGo(getX(), getY(), x, y,
direc)) {
                    reachable = true;
                    break;
```

```
                }
            }
        }
        if (!reachable) {
            game.setStatusInfo("OPS! Invalid move -
The "+this.getName()+" can't move to that
position");
            return false;
        }


        // move if one hive rule checked
        boolean moved =
moveWithOnehiveRuleChecked(x, y);
        if (moved) {
            game.setStatusInfo("The Piece "+
this.getName()+" moved!");
            System.out.println("Piece " + this + "
with (x,y) of (" + getX() + ", " + getY() + ")
moved to (" + x + ", " + y + ")");
            game.moveUnconditional(this, x, y);
            return true;
        }
        return false;
    }


    private boolean toGo(int x, int y, int endX,
int endY, Direction direc){

        if(Board.getNeighbourPoint(x, y, direc) ==
null) return false;


        if(game.getBoard().getPiece(x, y) != null)
{
            if(x % 2 != 0) {
                if((endX == x + 1 || endX == x - 1)
&& endY == y - 2) return true;

                else return false;
            }
            else {
                if((endX == x + 1 || endX == x - 1)
&& endY == y - 3) return true;
```

```
                else return false;
            }
        }
        if(toGo((int)Board.getNeighbourPoint(x, y,
direc).x, Board.getNeighbourPoint(x, y, direc).y,
endX, endY, direc)) return true;

        return false;
    }
}
```

Pillbug

```
package tps.tp4.pieces;

import java.awt.Color;
import java.awt.Point;

import tps.tp4.Board;
import tps.tp4.Game;
import tps.tp4.Game.Direction;

/**
 * PillBug class
 */
public class PillBug extends Piece {
    final static private Color color = Color.cyan;
    private Piece currentHolding = null;

    /**
     * constructor
     */
    public PillBug(Game game, boolean
isFromPlayerA) {
        super("PillBug", color, game,
isFromPlayerA);
    }

    /**
     * Move this piece to x,y if doesn't violate
the rules.
     *
     * The PillBug can move only one step. Should
not violate the one hive rule.
     */
    public boolean moveTo(int x, int y) {
        boolean reachable = false;
        int posX = getX(), posY = getY();
        if (currentHolding == null) {
            Piece piece =
game.getBoard().getPiece(x, y);
            if (piece == null) {
```

```
                // execute search for all the
coordinates
                for (Direction direc :
Direction.values()) {
                    Point p =
Board.getNeighbourPoint(getX(), getY(), direc);
                    if (p == null)
                        continue;
                    if (p.getX() == x && p.getY()
== y) {
                        reachable = true;
                        break;
                    }
                }
                if (!reachable)
                    return false;

                Direction direc =
Piece.getDirection(getX(), getY(), x, y);

                boolean canMove =
game.canPhysicallyMoveTo(x, y, direc);
                // move if one hive rule checked
                boolean moved = false;
                if(canMove) {
                    moved =
moveWithOnehiveRuleChecked(x, y);
                }
                if (moved) {
                    game.moveUnconditional(this, x,
y);
                }

                return moved;
            } else {
                for (Direction direc :
Direction.values()) {
                    Point p =
Board.getNeighbourPoint(posX, posY, direc);
                    if (p == null)
                        continue;

                    int originalX = piece.getX(),
originalY = piece.getY();
```

```
                    game.moveUnconditional(piece,
getX(), getY());
                    boolean moved =
game.getBoard().justOneHive(getX(), getY());
                    if(!moved)
game.moveUnconditional(piece, originalX,
originalY);

                    if (x == p.x && y == p.y
&&(game.getLastMoved() == null ||
!game.getLastMoved().equals(piece)) && moved) {
                        if
(game.getBoard().getBoardPlace(x, y).getNumPieces()
> 2)
                            return false;
                        currentHolding = piece;
                        game.getBoard().repaint();
                        return false;
                    }
                }
            }
        }else {
            Piece piece =
game.getBoard().getPiece(x, y);
            if(piece != null) return false;
            for (Direction direc :
Direction.values()) {
                Point p =
Board.getNeighbourPoint(getX(), getY(), direc);
                if (p == null)
                    continue;
                if (p.getX() == x && p.getY() == y)
{
                    reachable = true;
                    break;
                }
            }
            if (!reachable)
                return false;
            dropPiece(currentHolding, x, y);
            currentHolding = null;
            return true;
        }
        return false;
    }
```

```
    private void dropPiece(Piece holdingPiece,int
x,int y) {
        game.getBoard().remPiece(holdingPiece);
        game.getBoard().addPiece(holdingPiece, x,
y);
        game.getBoard().repaint();

    }
}
```