Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**КУРСОВОЙ ПРОЕКТ**

**Дисциплина:** Алгоритмы и структуры данных

**Тема:** Разработка решателя игры Terra Incognita

Выполнил

студент гр. 3530903/80003 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Хвацкин Л.И.

(подпись)

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Глухих М.И.

(подпись)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г.

Санкт-Петербург   
2019

**Оглавление**

[**ТЕХНИЧЕСКОЕ ЗАДАНИЕ** 3](#_Toc25151727)

[**МЕТОД РЕШЕНИЯ** 4](#_Toc25151728)

[**ЛИСТИНГ ПРОГРАММЫ** 7](#_Toc25151729)

[**Main.java** 7](#_Toc25151730)

[**MainMenuController.java** 8](#_Toc25151731)

[**ActionController.java** 13](#_Toc25151732)

[**HelpAboutController.java** 18](#_Toc25151733)

[**ExceptionController.java** 20](#_Toc25151734)

[**Parsing.java** 21](#_Toc25151735)

[**NodeCircle.java** 23](#_Toc25151736)

[**AlgorithmFindRoute.java** 24](#_Toc25151737)

[**ОШИБКИ И ПРЕДУПРЕЖДЕНИЯ** 29](#_Toc25151738)

[**ТЕСТЫ** 30](#_Toc25151739)

[**СКРИНШОТЫ ПРОГРАММЫ** 32](#_Toc25151740)

# **ТЕХНИЧЕСКОЕ ЗАДАНИЕ**

Задано поле для игры Terra Incognita, содержащее пустые клетки, стены, порталы, вход, выход и сокровище. Если существует путь от входа к выходу такой, что к выходу игрок придёт, собрав сокровище, решатель должен вывести такой путь, причём он должен по возможности быть кратчайшим. В противном случае решатель должен вывести по возможности кратчайший из путей, позволяющих определить, что введённое поле нерешаемо.

GitHub репозиторий проекта: https://github.com/TheGreenBeaver/TerraIncognita/tree/noFirstStep

# **МЕТОД РЕШЕНИЯ**

В проекте использована концепция MVC (Model-View-Controller) для отделения бизнес-логики от визуализации. Соответственно, классы, отвечающие за логику, размещены в директории model, контроллеры – в controllers, а отвечающие за визуализацию fxml-файлы – в resources.fxml.

Содержимое resources.fxml:

* MainScreen.fxml – графическое представление основной неизменной части экрана: верхней панели и геометрического фона. На верхней панели расположены кнопки Edit Maze и Help, позволяющие перейти к редактированию поля или к справке соответственно.
* MazeEditorView.fxml – графическое представление окна редактирования поля. Содержит на левой панели поля ввода параметров поля, индикаторы корректности введённых значений и кнопку подтверждения ввода. После подтверждения ввода также появляется редактируемый шаблон поля и кнопка Solve, запускающая решение. Правая часть сцены отведена под отображение результатов работы.
* HelpView.fxml – графическое представление окна справки.
* DangerousInputAlarm.fxml – графическое представление диалогового окна, появляющегося при вводе значений, больших 20, в поля параметров лабиринта. Содержит кнопки согласия и отказа продолжения работы с введёнными значениями.
* Hint.fxml – графическое представление подсказки с ошибками или предупреждениями.
* PortalSettings.fxml – графическое представление диалогового окна выбора номера портала. Содержит выпадающий список с возможными номерами и кнопку подтверждения выбора.

Cодержимое controllers:

* MainController – контроллер, отвечающий за обработку событий, происходящих на главном экране (нажатие кнопок Help и Edit Maze).
* MazeEditorController – контроллер, отвечающий за обработку событий, происходящих в окне редактирования поля. Также в классе реализованы методы отображения результата.
* DangerousInputAlarmController – контроллер, отвечающий за обработку событий в диалоговом окне подтверждения ввода больших чисел как параметров лабиринта.
* PortalSettingsControler – контроллер, отвечающий за обработку событий в диалоговом окне выбора номера портала.

Содержимое model:

* MainEngine – класс, в котором производятся основные вычисления.
* MazeGrid – класс, объектом которого является любой Лабиринт.
* Pair – класс, реализующий методы Пары.
* RadialCheck – класс, позволяющий искать ближайшую к текущей неизученную клетку, обходя лабиринт радиальными кольцами.
* UIHandler – класс, хранящий значения, введённые пользователем при редактировании поля.
* Util – класс, в котором заранее инициализируются необходимые картинки.
* Cell – класс, объектом которого является любая Клетка Лабиринта.
* CellType – перечисление возможных типов клетки: Пол, Стена, Портал, Сокровище, Вход, Выход.
* Coordinate – класс, реализующий свойства Координаты.
* Direction – перечисление возможных направлений движения.
* MoveResult – перечисление возможных результатов перемещения: Удача, Граница поля, Недостижимая клетка, Посещённая клетка, Портал.

Подробности работы приложения:

По умолчанию при запуске программы открывается окно редактирования поля. При желании пользователь может нажать на верхней панели кнопку Help и перейти к справке. За это отвечает метод MainController().loadHelp(), вызывающий внутри себя приватный метод loadView(Node view), выводящий на экран переданное в него окно (HelpView либо MazeEditorView).

Прежде всего пользователь должен ввести значения Высоты и Ширины Поля. Корректность ввода (исключительно числа, притом числа положительные, не большие максимального значения 35) проверяется каждый раз при изменении содержимого соответствующих текстовых полей путём вызова универсального метода checkInput. В работе этого метода, в свою очередь, участвуют методы universal, handleScull и clearScull, подсвечивающие нужным цветом глазницы черепов-индикаторов ошибок и предупреждений и устанавливающие для этих черепов появление определённой подсказки по нажатию.

Далее следует нажать на кнопку ОК. По её нажатию создаётся и визуализируется новый лабиринт введённых размеров. Если, однако, хотя бы одно из значений превышает рекомендуемый порог (20), сперва появляется диалоговое окно, предлагающее пользователю ещё раз подумать, хочет ли он продолжать работу именно с такими значениями. За все эти действия отвечает метод saveProperties, привязанный к событию onAction кнопки ОК.

После того, как поле отображено, пользователь может редактировать его, кликая на клетки. По умолчанию клетка является Пустой, после первого клика её тип меняется на «Стену», потом на «Портал», «Сокровище», «Вход» и «Выход». При нажатии на Портал правой кнопкой мыши также появляется диалоговое окно выбора номера для этого портала. Эту функциональность реализует метод onClick, привязанный к событию onMouseClick класса Cell.

По достижении желаемого вида поля предлагается нажать на кнопку Solve, чтобы запустить алгоритм собственно решения игры. При этом все введённые значения будут проверены с помощью методов CellType.fieldFilled, возвращающего значения

* GOOD, если все поля заполнены корректно
* ONE\_PORTAL, если на поле ровно один портал, что является невозможной ситуацией, т.к. у каждого портала должен быть следующий, в который он бы переправлял игрока;
* UNUSED\_ESSENTIALS, если игрок не добавил на поле Вход, Сокровище или Выход, являющиеся обязательными элементами игры;

… и метода UIHandler.portalNumsOK, возвращающего true только в том случае, если у всех добавленных порталов различные номера. Если не достигнуто сочетание CellType.fieldFilled = GOOD && UIHandler.portalNumsOK, решение не начнётся, а с помощью всё того же метода universal будет обработано поведение черепа, отвечающего за кнопки ОК и Solve.

В решении как таковом используется случайный выбор начальных направлений, чтобы по возможности избежать зацикливаний при многократном попадании в одни и те же порталы. Поэтому решение запускается четыре раза от разных промежутков выбора случайных значений, после чего из них выбирается наиболее удачное.

Процесс решения цикличен, на каждом шаге программа выбирает направление, в котором удобно двигаться в данный момент, и проверяет, что получится, если сделать такое движение. За движение отвечает метод Cell.move, возвращающий одно из перечисленных выше состояний после совершения перемещения. Теоретически можно было бы на каждом шаге вызывать алгоритм BFS и искать ближайшую неисследованную клетку, прилегающую к заведомо достижимой, однако этот алгоритм весьма трудоёмок, и такой метод решения сильно замедлил бы работу программы. Поэтому направление выбирается по возможности механически в методе calculateDirection, использующем для таких целей последовательность условных операторов и операторов множественного выбора. Выбранное таким образом направление дополнительно проверяется методом finalCheck, чтобы сразу отбросить направление, которое заведомо приведёт к неудовлетворительному результату, но по параметрам прошло проверку calculateDirection.

Алгоритм BFS используется тогда, когда решатель после очередного хода «утыкается» в уже посещённую клетку; тогда с помощью объекта класса RadialCheck ищется ближайшая достижимая непосещённая клетка. При этом используется также алгоритм DFS, что увеличивает время работы программы, но, увы, необходимо ввиду односторонней работы порталов. Впрочем, благодаря ленивому алгоритму проверки условий компилятором Java DFS вызывается только в том случае, если клетка достижима хотя бы потенциально.

В классе MainEngine также содержится важный субкласс LocalsTree, обеспечивающий работу с порталами. Каждый раз, когда решатель попадает в неизвестный портал, создаётся копия поля, на которой все клетки не изучены, т.к. решатель в таких условиях по правилам игры не знает своей текущей координаты. Такое состояние отмечается boolean переменной blindMode. Соответственно, решатель может в какой-то момент либо выйти из этого состояния, встретив на пути нечто, что позволит вновь определить свою координату, либо перейти в очередной портал. В любом случае все данные, собранные в процессе слепого путешествия, сохранятся как уровень дерева LocalsTree, что позволит использовать их при повторном попадании в те же порталы.

Каждый шаг алгоритма в итоге сохраняется в списке пар steps, причём первый элемент пары – координата посещённой клетки, а второй – boolean переменная, показывающая, возможно ли было бы посетить эту клетку, если бы в игру играл настоящий игрок по всем правилам. Дело в том, что при попадании в тупик программа возвращается к предыдущему уровню LocalsTree с помощью метода previousState; по правилам игры это запрещено, однако нам необходимо найти путь к выходу, если только он существует, и попадание в тупик до полного изучения лабиринта не всегда однозначно означает, что текущий лабиринт нерешаем.

По завершении работы алгоритма координаты из steps выводятся на экран в виде списка, при наведении мыши на текстовое отображение координаты в котором клетка с такой координатой подсвечивается в лабиринте. За отображение этого списка, а также прочих результатов работы программы отвечает часть метода MazeEditorController.solve, следующая за запуском MainEngine.solve.

# 

# **ЛИСТИНГ ПРОГРАММЫ**

**Main.java**

package greenbeaver.terraincognita;  
  
import javafx.application.Application;  
import javafx.fxml.FXMLLoader;  
import javafx.geometry.Rectangle2D;  
import javafx.scene.Parent;  
import javafx.scene.Scene;  
import javafx.scene.layout.VBox;  
import javafx.stage.Screen;  
import javafx.stage.Stage;  
import javafx.stage.StageStyle;  
  
public class Main extends Application {  
  
 @Override  
 public void start(Stage primaryStage) throws Exception{  
 Parent root = FXMLLoader.*load*(getClass().getResource("/fxml/MainScreen.fxml"));  
  
 Rectangle2D screenRect = Screen.*getPrimary*().getVisualBounds();  
 ((VBox) root).setPrefWidth(screenRect.getWidth());  
 ((VBox) root).setPrefHeight(screenRect.getHeight());  
  
 Scene rootScene = new Scene(root);  
 primaryStage.setFullScreen(true);  
 primaryStage.setResizable(false);  
 primaryStage.setScene(rootScene);  
 primaryStage.initStyle(StageStyle.*TRANSPARENT*);  
 primaryStage.show();  
 }  
  
 public static void main(String[] args) {  
 *launch*(args);  
 }  
}

**MainController.java**

package greenbeaver.terraincognita.controllers;  
  
import javafx.collections.ObservableList;  
import javafx.event.ActionEvent;  
import javafx.fxml.FXML;  
import javafx.fxml.FXMLLoader;  
import javafx.fxml.Initializable;  
import javafx.scene.Node;  
import javafx.scene.control.Button;  
import javafx.scene.layout.VBox;  
import javafx.stage.Stage;  
  
import java.io.IOException;  
import java.net.URL;  
import java.util.ResourceBundle;  
  
public class MainController implements Initializable {  
  
 @FXML  
 private VBox mainContainer;  
  
 private Node mazeEditorView;  
 {  
 try {  
 mazeEditorView = FXMLLoader.*load*(getClass().getResource("/fxml/MazeEditorView.fxml"));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 private Node helpView;  
 {  
 try {  
 helpView = FXMLLoader.*load*(getClass().getResource("/fxml/HelpView.fxml"));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Override  
 public void initialize(URL location, ResourceBundle resources) {  
 loadView(mazeEditorView);  
 }  
  
 private void loadView(Node view) {  
 ObservableList<Node> children = mainContainer.getChildren();  
 if (children.size() != 1) {  
 children.remove(1);  
 }  
 children.add(view);  
 }  
  
 @FXML  
 void loadHelp() {  
 loadView(helpView);  
 }  
  
 @FXML  
 void loadMazeEditor() {  
 loadView(mazeEditorView);  
 }  
  
 @FXML  
 void fold(ActionEvent actionEvent) {  
 ((Stage)((Node)actionEvent.getSource()).getScene().getWindow()).setIconified(true);  
 }  
  
 @FXML  
 void closeApp(ActionEvent actionEvent) {  
 ((Stage)((Node)actionEvent.getSource()).getScene().getWindow()).close();  
 }  
}

**MazeEditorController.java**

package greenbeaver.terraincognita.controllers;  
  
import greenbeaver.terraincognita.model.MainEngine;  
import greenbeaver.terraincognita.model.MazeGrid;  
import greenbeaver.terraincognita.model.Pair;  
import greenbeaver.terraincognita.model.UIHandler;  
import greenbeaver.terraincognita.model.cellConstruction.Cell;  
import greenbeaver.terraincognita.model.cellConstruction.CellType;  
import greenbeaver.terraincognita.model.cellConstruction.Coordinate;  
import javafx.collections.ObservableList;  
import javafx.fxml.FXML;  
import javafx.fxml.FXMLLoader;  
import javafx.fxml.Initializable;  
import javafx.geometry.Rectangle2D;  
import javafx.scene.Node;  
import javafx.scene.Parent;  
import javafx.scene.Scene;  
import javafx.scene.control.Button;  
import javafx.scene.control.Label;  
import javafx.scene.control.ListView;  
import javafx.scene.control.TextField;  
import javafx.scene.image.Image;  
import javafx.scene.image.ImageView;  
import javafx.scene.layout.AnchorPane;  
import javafx.scene.layout.VBox;  
import javafx.scene.paint.Color;  
import javafx.stage.Modality;  
import javafx.stage.Screen;  
import javafx.stage.Stage;  
import javafx.stage.StageStyle;  
import org.jetbrains.annotations.Nullable;  
  
import java.io.IOException;  
import java.net.URL;  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.Map;  
import java.util.ResourceBundle;  
  
public class MazeEditorController implements Initializable {  
  
 private enum InputState {  
 *CORRECT*,  
 *EMPTY*,  
 *WRONG*,  
 *DANGEROUS* }  
 private HashMap<TextField, InputState> inputs = new HashMap<>();  
 private MazeGrid currentMaze;  
  
 private final static int *MAX\_INPUT* = 35;  
 private final static int *DANGEROUS\_RANGE* = 20;  
 private final static String *NON\_NUMERICAL\_INPUT\_MESSAGE* = "ERROR: You may only use numbers";  
 private final static String *ZERO\_INPUT\_MESSAGE* = "ERROR: Enter a value that's above zero";  
 private final static String *MAX\_INPUT\_MESSAGE* = "ERROR: Enter a value less than " + *MAX\_INPUT*;  
 private final static String *DANGEROUS\_INPUT\_MESSAGE* = "WARNING: Inputs more than "  
 + *DANGEROUS\_RANGE* + " might cause visualising or processing problems";  
 private static final String *NOT\_ALL\_INPUTS\_MESSAGE* = "ERROR: You must fill all input fields";  
 private static final String *WRONG\_INPUT\_MESSAGE* = "ERROR: Enter correct values first";  
 private static final String *ONE\_PORTAL\_MESSAGE* = "ERROR: There can't be just one Portal in the Maze," +  
 " each Portal must have a next one";  
 private static final String *EQUAL\_PORTALS\_MESSAGE* = "ERROR: Some Portals have equal numbers;" +  
 " right-click on them to fix this";  
 private static final String *NO\_ESSENTIALS\_MESSAGE* = "ERROR: You must have an Entrance, an Exit and a Treasure in the Maze";  
  
 @FXML  
 private ImageView filler;  
  
 @FXML  
 private AnchorPane mazeContainer;  
  
 @FXML  
 private ImageView heightScull;  
 @FXML  
 private TextField mazeHeightInput;  
  
 @FXML  
 private ImageView widthScull;  
 @FXML  
 private TextField mazeWidthInput;  
  
 @FXML  
 private ImageView submissionScull;  
  
 @FXML  
 private Button solveButton;  
  
 @FXML  
 private VBox resultView;  
  
 @FXML  
 private Label treasureState;  
 @FXML  
 private Label exitState;  
 @FXML  
 private Label cCellsPassed;  
 @FXML  
 private Label rCellsPassed;  
  
  
 private Stage hint;  
 private Label hintText;  
 {  
 hint = new Stage();  
  
 try {  
 Parent root = FXMLLoader.*load*(getClass().getResource("/fxml/Hint.fxml"));  
 hintText = (Label) root.getChildrenUnmodifiable().get(0);  
 Scene scene = new Scene(root);  
 scene.setFill(Color.*TRANSPARENT*);  
 hint.setScene(scene);  
 hint.initModality(Modality.*NONE*);  
 hint.initStyle(StageStyle.*TRANSPARENT*);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 private Stage alarm;  
  
 {  
 alarm = new Stage();  
 try {  
 Parent root = FXMLLoader.*load*(getClass().getResource("/fxml/DangerousInputAlarm.fxml"));  
 Scene scene = new Scene(root);  
 scene.setFill(Color.*TRANSPARENT*);  
 alarm.setScene(scene);  
 alarm.initModality(Modality.*WINDOW\_MODAL*);  
 alarm.initStyle(StageStyle.*TRANSPARENT*);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Override  
 public void initialize(URL location, ResourceBundle resources) {  
 Rectangle2D screenRect = Screen.*getPrimary*().getVisualBounds();  
 double h = screenRect.getHeight();  
 if (h > 800) {  
 filler.setViewport(new Rectangle2D(0, 0, 230, h - 800));  
 filler.setVisible(true);  
 }  
  
 inputs.put(mazeWidthInput, InputState.*EMPTY*);  
 inputs.put(mazeHeightInput, InputState.*EMPTY*);  
 }  
  
 private void clearScull(ImageView scull) {  
 scull.setOnMouseClicked(null);  
 scull.setImage(new Image("/images/normalScull.png"));  
 }  
  
 private void universal(@Nullable TextField inputField,  
 InputState toReplaceFor,  
 String message,  
 ImageView scull) {  
  
 if (inputField != null) {  
 inputs.replace(inputField, toReplaceFor);  
 clearScull(submissionScull);  
 }  
  
 switch (toReplaceFor) {  
 case *EMPTY*:  
  
 case *CORRECT*: {  
 clearScull(scull);  
 break;  
 }  
  
 case *WRONG*: {  
 scull.setImage(new Image("/images/errorScull.png"));  
 handleScull(scull, message);  
 break;  
 }  
  
 case *DANGEROUS*: {  
 scull.setImage(new Image("/images/warningScull.png"));  
 handleScull(scull, message);  
 break;  
 }  
 }  
 }  
  
 private void handleScull(ImageView scull, String message) {  
 scull.setOnMouseClicked(e -> {  
 if (hint.isShowing()) {  
 hint.close();  
 } else {  
 double x = e.getScreenX() + 5;  
 double y = e.getScreenY() + 5;  
 hint.setX(x);  
 hint.setY(y);  
 hintText.setText(message);  
 if (hint.getOwner() == null) {  
 hint.initOwner(scull.getScene().getWindow());  
 }  
 hint.show();  
 }  
 });  
 }  
  
 private void checkInput(TextField inputField, ImageView scull) {  
 String inputText = inputField.getText();  
 if (inputText.isEmpty()) {  
 universal(inputField, InputState.*EMPTY*, "", scull);  
 return;  
 }  
  
 if (!inputText.matches("\\d\*")) {  
 universal(inputField, InputState.*WRONG*, *NON\_NUMERICAL\_INPUT\_MESSAGE*, scull);  
 return;  
 }  
  
 try {  
 int inputValue = Integer.*parseInt*(inputText);  
 if (inputValue < *DANGEROUS\_RANGE* && inputValue > 0) {  
 universal(inputField, InputState.*CORRECT*, "", scull);  
 return;  
 }  
  
 if (inputValue == 0) {  
 universal(inputField, InputState.*WRONG*, *ZERO\_INPUT\_MESSAGE*, scull);  
 return;  
 }  
  
 if (inputValue > *MAX\_INPUT*) {  
 universal(inputField, InputState.*WRONG*, *MAX\_INPUT\_MESSAGE*, scull);  
 return;  
 }  
  
 universal(inputField, InputState.*DANGEROUS*, *DANGEROUS\_INPUT\_MESSAGE*, scull);  
 } catch (NumberFormatException tooBig) {  
 universal(inputField, InputState.*WRONG*, *MAX\_INPUT\_MESSAGE*, scull);  
 }  
 }  
  
 @FXML  
 void checkHeightInput() {  
 checkInput(mazeHeightInput, heightScull);  
 }  
  
 @FXML  
 void checkWidthInput() {  
 checkInput(mazeWidthInput, widthScull);  
 }  
  
 private void saveAndShowMazeGrid() {  
 MainEngine.*setMazeHeight*(Integer.*parseInt*(mazeHeightInput.getText()));  
 MainEngine.*setMazeWidth*(Integer.*parseInt*(mazeWidthInput.getText()));  
 currentMaze = new MazeGrid();  
 currentMaze.setOnMouseClicked(e -> clearScull(submissionScull));  
 ObservableList<Node> mazeContainerChildren = mazeContainer.getChildren();  
 mazeContainerChildren.clear();  
 mazeContainer.getChildren().add(currentMaze);  
 UIHandler.*clearUIHandler*();  
 solveButton.setVisible(true);  
 }  
  
 @FXML  
 private void saveProperties() {  
 InputState widthState = inputs.get(mazeWidthInput);  
 InputState heightState = inputs.get(mazeHeightInput);  
  
 if (heightState == InputState.*CORRECT* && widthState == InputState.*CORRECT*) {  
 saveAndShowMazeGrid();  
 return;  
 }  
  
 if (heightState == InputState.*EMPTY* || widthState == InputState.*EMPTY*) {  
 universal(null, InputState.*WRONG*, *NOT\_ALL\_INPUTS\_MESSAGE*, submissionScull);  
 return;  
 }  
  
 if (heightState == InputState.*WRONG* || widthState == InputState.*WRONG*) {  
 universal(null, InputState.*WRONG*, *WRONG\_INPUT\_MESSAGE*, submissionScull);  
 return;  
 }  
  
 if (heightState == InputState.*DANGEROUS* || widthState == InputState.*DANGEROUS*) {  
  
 if (alarm.getOwner() == null) {  
 alarm.initOwner(mazeContainer.getScene().getWindow());  
 }  
 alarm.showAndWait();  
  
 if (UIHandler.*getContinueWithDangerousInput*()) {  
 saveAndShowMazeGrid();  
 }  
 }  
 }  
  
 @FXML  
 void solve() {  
  
 CellType.FieldState fieldState = CellType.*fieldFilled*();  
 boolean portalsOK = UIHandler.*portalNumsOK*();  
  
 if (fieldState == CellType.FieldState.*GOOD* && portalsOK) {  
 MainEngine.*setMaze*(currentMaze.getMazeAsArray());  
 MainEngine.*setPortalTransitions*(UIHandler.*getPortalTransitions*());  
 HashMap<Pair<Boolean, Boolean>, ArrayList<Pair<Coordinate, Boolean>>> results = new HashMap<>();  
 ArrayList<Pair<Coordinate, Boolean>> res = new ArrayList<>();  
 for (int i = 0; i < 4; i++) {  
 MainEngine.*solve*(i);  
 results.put(new Pair<>(MainEngine.*exitReached*(), MainEngine.*treasureFound*()), MainEngine.*getSteps*());  
 if (MainEngine.*exitReached*() && MainEngine.*treasureFound*() && (res.isEmpty() || MainEngine.*getSteps*().size() < res.size())) {  
 res = MainEngine.*getSteps*();  
 treasureState.setText("Treasure Found: " + MainEngine.*getRTreasure*().toString());  
 exitState.setText("Exit Reached: TRUE");  
 }  
 }  
 if (res.isEmpty()) {  
 for (Map.Entry<Pair<Boolean, Boolean>, ArrayList<Pair<Coordinate, Boolean>>> entry : results.entrySet()) {  
 if (res.isEmpty() || entry.getValue().size() < res.size()) {  
 res = entry.getValue();  
 treasureState.setText("Treasure Found: " + (entry.getKey().getB() ? MainEngine.*getRTreasure*().toString() : "FALSE"));  
 exitState.setText("Exit Reached: " + (entry.getKey().getA() ? "TRUE" : "FALSE"));  
 }  
 }  
 }  
  
 cCellsPassed.setText("Computational Cells Passed: " + res.size());  
  
 ListView<Label> resultList = new ListView<>();  
 int r = 0;  
 for (Pair<Coordinate, Boolean> cb : res) {  
  
 if (cb.getB()) {  
 r++;  
 }  
  
 Label resString = new Label(cb.getA().toString());  
 if (cb.getA().fits()) {  
 resString.setOnMouseEntered(e -> currentMaze.getMazeAsArray()[cb.getA().getY()][cb.getA().getX()].highlight(cb.getB()));  
 resString.setOnMouseExited(e -> {  
 Cell cell = currentMaze.getMazeAsArray()[cb.getA().getY()][cb.getA().getX()];  
 Image def = cell.getCellType().getImage();  
 cell.setImage(def);  
 });  
 }  
 resultList.getItems().add(resString);  
 }  
  
 rCellsPassed.setText("Real Cells Passed: " + r);  
 if (resultView.getChildren().size() > 5) {  
 resultView.getChildren().remove(5);  
 }  
 resultView.getChildren().add(resultList);  
 } else {  
 if (!portalsOK) {  
 universal(null, InputState.*WRONG*, *EQUAL\_PORTALS\_MESSAGE*, submissionScull);  
 return;  
 }  
  
 switch (fieldState) {  
 case *ONE\_PORTAL*: {  
 universal(null, InputState.*WRONG*, *ONE\_PORTAL\_MESSAGE*, submissionScull);  
 return;  
 }  
 case *UNUSED\_ESSENTIALS*: {  
 universal(null, InputState.*WRONG*, *NO\_ESSENTIALS\_MESSAGE*, submissionScull);  
 }  
 }  
 }  
 }  
}

**PortalSettingsController.java**

package greenbeaver.terraincognita.controllers;  
  
import greenbeaver.terraincognita.model.UIHandler;  
import greenbeaver.terraincognita.model.cellConstruction.CellType;  
import javafx.event.ActionEvent;  
import javafx.fxml.FXML;  
import javafx.fxml.Initializable;  
import javafx.scene.Node;  
import javafx.scene.control.ChoiceBox;  
import javafx.stage.Stage;  
  
import java.net.URL;  
import java.util.ResourceBundle;  
  
public class PortalSettingsController implements Initializable {  
 @FXML  
 ChoiceBox<String> variants;  
  
 @Override  
 public void initialize(URL location, ResourceBundle resources) {  
 int varAmount = CellType.*PORTAL*.getUsedAmount();  
 for (int i = 0; i < varAmount; i++) {  
 variants.getItems().add(String.*valueOf*(i + 1));  
 }  
 variants.setValue(String.*valueOf*(UIHandler.*getCurrentPortalNum*() + 1));  
 }  
  
 public void submit(ActionEvent actionEvent) {  
 UIHandler.*setPortalNum*(UIHandler.*getCurrentPortal*(), Integer.*parseInt*(variants.getValue()) - 1);  
 ((Stage) ((Node) actionEvent.getSource()).getScene().getWindow()).close();  
 }  
}

**DangerousInputAlarmController.java**

package greenbeaver.terraincognita.controllers;  
  
import greenbeaver.terraincognita.model.UIHandler;  
import javafx.event.ActionEvent;  
import javafx.fxml.FXML;  
import javafx.scene.Node;  
import javafx.stage.Stage;  
  
public class DangerousInputAlarmController {  
  
 @FXML  
 void confirm(ActionEvent actionEvent) {  
 UIHandler.*setContinueWithDangerousInput*(true);  
 ((Stage)((Node)actionEvent.getSource()).getScene().getWindow()).close();  
 }  
  
 @FXML  
 void refuse(ActionEvent actionEvent) {  
 UIHandler.*setContinueWithDangerousInput*(false);  
 ((Stage)((Node)actionEvent.getSource()).getScene().getWindow()).close();  
 }  
}

**MainEngine.java**

package greenbeaver.terraincognita.model;  
  
import greenbeaver.terraincognita.model.cellConstruction.\*;  
import org.jetbrains.annotations.NotNull;  
  
import java.util.\*;  
import java.util.concurrent.ArrayBlockingQueue;  
  
public class MainEngine {  
  
 private static class AdjLink {  
 private final Coordinate from;  
 private final Coordinate to;  
 private final boolean bothDir;  
  
 private AdjLink(Coordinate from, Coordinate to, boolean bothDir) {  
 this.from = from;  
 this.to = to;  
 this.bothDir = bothDir;  
 }  
 }  
  
 private static class LocalsTree {  
 private static class Level {  
 private final int id;  
 private Level parent; // previous Level, root represents initial (non-blind) state  
 private ArrayList<Level> parents; // ONLY USED FOR ROOT BECAUSE IT MIGHT BE REACHABLE FROM SEVERAL LEVELS  
 private Coordinate portalFromParent; // local Coordinate (portal) where the Player was placed after travelling through portal from parent  
 private Coordinate gPortalFromParent; // global analogue  
  
 private HashMap<Level, Coordinate> children; // Next Levels with Coordinates of portals leading to them  
  
 private boolean[][] lAdjacencyMatrix;  
 private Coordinate.CoordinateState[][] lCoordinateStates;  
  
 // Used when returning from a blocked position  
 private Direction last; // Direction that led to the portal that put the Player into this level  
 // These duplicate their analogues from MainEngine class itself  
 private int lFailCount;  
 private boolean lYCoordinateDefined;  
 private boolean lXCoordinateDefined;  
 private ArrayList<Pair<Coordinate, Coordinate.CoordinateState>> lLocalPath;  
 private ArrayList<AdjLink> lLocalAdjacencyLinks;  
  
 private boolean pathToExit; // true if this level must be passed to reach exit; used when returning to previous levels to then find this place again  
  
 private Level(Level parent, Coordinate portalFromParent, Coordinate gPortalFromParent) {  
 this.id = *levelId*++;  
 this.parent = parent;  
 this.portalFromParent = portalFromParent == null ? null : portalFromParent.copy();  
 this.gPortalFromParent = gPortalFromParent == null ? null : gPortalFromParent.copy();  
  
 int adjSize = parent != null ? *localCellAmount*() : *cellAmount*(); // if parent is null, than currentLevel is root and we are at the initial non-blind Level  
 lAdjacencyMatrix = new boolean[adjSize][adjSize];  
  
 int cStatesH = parent != null ? *mazeHeight* \* 2 + 1 : *mazeHeight*;  
 int cStatesW = parent != null ? *mazeWidth* \* 2 + 1 : *mazeWidth*;  
 lCoordinateStates = new Coordinate.CoordinateState[cStatesH][cStatesW];  
  
 children = new HashMap<>();  
 }  
  
 // When ALREADY\_VISITED\_CELL is hit and radialCheck returns null, the first attempt (if in blind mode) is to try escaping through the portal that was an entrance for this level  
 // This represents if such opportunity has already been used  
 private boolean triedEscapingThroughEntrance() {  
 return children.containsValue(portalFromParent);  
 }  
  
 @Override  
 public boolean equals(Object obj) {  
 assert obj != null;  
  
 if (!obj.getClass().equals(Level.class)) {  
 return false;  
 }  
  
 Level other = (Level) obj;  
  
 return other.id == this.id;  
 }  
 }  
  
 private static Level *root*;  
 private static Stack<Direction> *unblindLastMoves*;  
  
 private static void clear() {  
 *root* = new Level(null, null, null);  
 *root*.parents = new ArrayList<>();  
  
 *unblindLastMoves* = new Stack<>();  
 }  
  
 // Used when falling into a portal. Creates new Level and saves the properties of the current one in case there'll be a need to return  
 static void add(Direction direction) {  
 Coordinate p = *currentLevel*.equals(*root*) ? *oldRealCoordinate*.copy() : *localCoordinate*.copy();  
 p = p.add(direction); // Coordinate of portal at the current Level  
  
 *saveCurrentState*();  
  
 *blindMode* = true;  
  
 Level level = new Level(*currentLevel*, new Coordinate(*mazeWidth*, *mazeHeight*), *newRealCoordinate*);  
 *currentLevel*.children.put(level, p);  
 level.last = direction;  
 *currentLevel* = level; // switch to new level  
 }  
  
 private static void saveCurrentState() {  
 Coordinate.CoordinateState[][] c = Coordinate.*getCoordinateStates*();  
 for (int i = 0; i < c.length; i++) {  
 System.*arraycopy*(c[i], 0, *currentLevel*.lCoordinateStates[i], 0, c[0].length);  
 }  
  
 boolean[][] adjToWork = *currentLevel*.equals(*root*) ? *adjacencyMatrix* : *localAdjacencyMatrix*;  
 for (int i = 0; i < adjToWork.length; i++) {  
 System.*arraycopy*(adjToWork[i], 0, *currentLevel*.lAdjacencyMatrix[i], 0, adjToWork[0].length);  
 }  
  
 *currentLevel*.lFailCount = *failCount*;  
 *currentLevel*.lYCoordinateDefined = *yCoordinateDefined*;  
 *currentLevel*.lXCoordinateDefined = *xCoordinateDefined*;  
 if (!*currentLevel*.equals(*root*)) {  
 *currentLevel*.lLocalPath = new ArrayList<>(*localPath*);  
 *currentLevel*.lLocalAdjacencyLinks = new ArrayList<>(*localAdjacencyLinks*);  
 }  
 }  
  
 // Used only if blocked  
 static void previousState() {  
  
 int lastPseudo = *pseudo*.pop();  
 for (int i = *steps*.size() - 1; i >= lastPseudo ; i--) {  
 *steps*.get(i).setB(false);  
 }  
  
 Pair<Coordinate, Coordinate> toReturn = *portalStack*.pop(); // A is realCoordinate, B is localCoordinate or null if the portal was not met in the blindMode  
 Coordinate rp = toReturn.getA(); // realCoordinate of the entering portal  
 if (toReturn.getB() == null) { // if we travelled from unblind to unblind and don't need to change level  
 rp.setCoordinateState(Coordinate.CoordinateState.*KNOWN\_BAD\_PORTAL*, null);  
 Coordinate c = rp.add((*blindMode* ? *currentLevel*.last : *unblindLastMoves*.pop()).opposite());  
 *currentCell* = *maze*[c.getY()][c.getX()];  
 *blindMode* = false;  
 *currentLevel* = *root*;  
 return;  
 }  
  
 Coordinate lp = toReturn.getB();  
 Level actualParent = *currentLevel*.parent;  
 if (actualParent == null) {  
 for (Level l : *currentLevel*.parents) {  
 for (Map.Entry<Level, Coordinate> entry : l.children.entrySet()) {  
 if (*currentLevel*.equals(entry.getKey())) {  
 actualParent = l;  
 break;  
 }  
 }  
 }  
 }  
  
 boolean withDelete = !*currentLevel*.pathToExit;  
 // if the Level the Player's just explored is a part of the portal path to exit and they're just returning back to then visit this Level at the very end, we don't need to delete this level  
  
 if (!withDelete) {  
 *saveCurrentState*();  
 } else {  
 Objects.*requireNonNull*(actualParent).children.remove(*currentLevel*);  
 }  
  
 *lastTried* = *blindMode* ? *currentLevel*.last : *unblindLastMoves*.pop();  
 *lastCalculatedDirectionFailed* = true;  
 *shift* = false;  
 // these two guarantee that when returned to the previous Level, the Player will search for a new Direction to move  
 *currentLevel* = actualParent;  
 *failCount* = Objects.*requireNonNull*(*currentLevel*).lFailCount + 1;  
 // HIGHER LEVEL FROM NOW ON!!!  
 *currentLevel*.pathToExit = !withDelete;  
  
 *yCoordinateDefined* = *currentLevel*.lYCoordinateDefined;  
 *xCoordinateDefined* = *currentLevel*.lXCoordinateDefined;  
 Coordinate newCurr = lp.add(*lastTried*.opposite());  
 if (!*currentLevel*.equals(*root*)) { // non-blind level just does not have such attributes  
 *localPath* = new ArrayList<>(*currentLevel*.lLocalPath);  
 *localAdjacencyLinks* = new ArrayList<>(*currentLevel*.lLocalAdjacencyLinks);  
 *localCoordinate* = newCurr;  
 *blindMode* = true;  
 } else {  
 *blindMode* = false; // changing blindMode here...  
 }  
  
 // ... so that here, if the Player's returned to non-blind state, we would already change global Coordinate States  
 Coordinate.*setCoordinateStates*(*currentLevel*.lCoordinateStates);  
 boolean[][] adjToWork = *currentLevel*.equals(*root*) ? *adjacencyMatrix* : *localAdjacencyMatrix*;  
 for (int i = 0; i < adjToWork.length; i++) {  
 System.*arraycopy*(*currentLevel*.lAdjacencyMatrix[i], 0, adjToWork[i], 0, adjToWork.length);  
 }  
  
 // here, CoordinateState is also set already according to the current state of blindMode  
 lp.setCoordinateState(withDelete  
 ? Coordinate.CoordinateState.*KNOWN\_BAD\_PORTAL* : Coordinate.CoordinateState.*KNOWN\_PORTAL\_TO\_EXIT*,  
 null);  
 Coordinate rC = rp.add(*lastTried*.opposite());  
 *currentCell* = *maze*[rC.getY()][rC.getX()];  
 }  
  
 static void unblind() {  
 Level lParent = *currentLevel*.parent;  
 *unblindLastMoves*.push(*currentLevel*.last);  
 *blindMode* = false;  
  
 Coordinate enteringInParent = lParent.children.get(*currentLevel*);  
 lParent.children.remove(*currentLevel*);  
  
 if (!lParent.equals(*root*)) { // if the Player didn't manage to unblind after the very first Level  
 lParent.children.put(*root*, enteringInParent);  
 *root*.parents.add(lParent);  
 } else { // otherwise the two only levels are just combined into one big non-blind root  
 int from = enteringInParent.add(*currentLevel*.last.opposite()).getRawNumber();  
 int to = *currentLevel*.gPortalFromParent.getRawNumber();  
 *adjacencyMatrix*[from][to] = true;  
 }  
  
 for (Pair<Coordinate, Coordinate.CoordinateState> cAndState : *localPath*) {  
 cAndState.getA().setCoordinateState(cAndState.getB(), null);  
 }  
 for (AdjLink cAndC : *localAdjacencyLinks*) {  
 int from = cAndC.from.getRawNumber();  
 int to = cAndC.to.getRawNumber();  
 *adjacencyMatrix*[from][to] = true;  
 if (cAndC.bothDir) {  
 *adjacencyMatrix*[to][from] = true;  
 }  
 }  
  
 Coordinate relation = *currentLevel*.gPortalFromParent.subtract(*currentLevel*.portalFromParent);  
 Coordinate newCurr = *localCoordinate*.add(relation.getX(), relation.getY());  
 *currentCell* = *maze*[newCurr.getY()][newCurr.getX()];  
  
 if (*exit* != null && !*exit*.getB().equals(*root*) && *exit*.getB().equals(*currentLevel*)) {  
 Coordinate e = *exit*.getA().add(relation.getX(), relation.getY());  
 *exit* = new Pair<>(e, *root*);  
 }  
 *currentLevel* = *root*;  
 }  
 }  
  
 private static int *mazeHeight*; // set by setMazeHeight() from MazeEditorController when the maze is created  
 private static int *mazeWidth*; // set by setMazeWidth() from MazeEditorController when the maze is created  
 private static Coordinate *entrance*; // set by setEntrance() from MazeEditorController when one of the cells is made an entrance  
 private static Pair<Coordinate, LocalsTree.Level> *exit*; // initially set to null in solve()  
 private static Pair<Coordinate, LocalsTree.Level> *treasure*; // initially set to false in solve()  
 private static Cell *currentCell*; // initially set to entrance in solve(); tracks the real position of the Player; the Cell where the Player was at the start of makeMove()  
 private static Cell[][] *maze*; // set by setMaze() from MazeEditorController when the maze is created  
 private static boolean[][] *adjacencyMatrix*; // initially set to a matrix filled by false in solve()  
 private static boolean[][] *localAdjacencyMatrix*;  
 private static boolean *firstStep*; // initially set to true in solve(); shows if the Player is now trying to reach the bottom right corner of the maze to then start scanning it in zigzags  
 private static int *initialShift*;  
 private static boolean *shift*; // initially set to false in solve(); shows if the Player should now change their X coordinate in case they are at the bottom or top border  
 private static Direction *general*; // initially set to Direction.UP() in solve(); shows what overall direction the Player is moving now, not paying attention to firstStep or shift  
 private static ArrayList<Pair<Coordinate, Boolean>> *steps*; // initially set to an ArrayList with only entrance in it in solve(); stores all the cells that the Player visited or tried to visit  
 private static MoveResult *moveResult*; // initially set to null in solve()  
 private static Direction *lastTried*; // initially set to null in solve()  
 private static int *failCount*; // initially set to 0 in solve; shows how much times in a row the Player failed to move in a calculated direction  
 private static boolean *lastCalculatedDirectionFailed*; // initially set to false in solve()  
 private static Coordinate *current*; // initially set to null in solve(); shows the coordinate where the Player was at the moment when a new direction is calculated  
 private static Direction *moment*; // initially set to null in solve(); shows the direction that the Player should follow to reach the nearest unknown cell after bfs  
 private static Coordinate[] *portalTransitions*; // set by setPortalTransitions() from MazeEditorController when the maze is created  
 private static boolean *blindMode*; // initially set to false in solve(); shows if Player now knows his exact coordinate  
 private static Coordinate *localCoordinate*; // initially set to null in solve(); used instead of real coordinate during blindMode  
 private static boolean *yCoordinateDefined*;  
 private static boolean *xCoordinateDefined*;  
 private static ArrayList<Pair<Coordinate, Coordinate.CoordinateState>> *localPath*;  
 private static ArrayList<AdjLink> *localAdjacencyLinks*;  
 private static Coordinate *newLocalCoordinate*;  
 private static Coordinate *oldRealCoordinate*;  
 private static Cell *newRealCell*;  
 private static Coordinate *newRealCoordinate*;  
 private static LocalsTree.Level *currentLevel*;  
 private static boolean *mBorder*;  
 private static int *impossibleDirections*;  
 private static int *levelId*;  
 private static Coordinate *rTreasure*;  
 private static int *range*;  
 private static Stack<Pair<Coordinate, Coordinate>> *portalStack*;  
 private static boolean *inSearchForExit*;  
 private static Stack<Integer> *pseudo*;  
  
 // Getters and setters  
 public static Coordinate[] getPortalTransitions() {  
 return *portalTransitions*;  
 }  
  
 public static void setPortalTransitions(Coordinate[] portalTransitions) {  
 MainEngine.*portalTransitions* = portalTransitions;  
 }  
  
 public static int getMazeHeight() {  
 return *mazeHeight*;  
 }  
  
 public static void setMazeHeight(int newMazeHeight) {  
 *mazeHeight* = newMazeHeight;  
 }  
  
 public static int getMazeWidth() {  
 return *mazeWidth*;  
 }  
  
 public static void setMazeWidth(int newMazeWidth) {  
 *mazeWidth* = newMazeWidth;  
 }  
  
 public static Cell[][] getMaze() {  
 return *maze*;  
 }  
  
 public static void setMaze(Cell[][] maze) {  
 MainEngine.*maze* = maze;  
 }  
  
 public static void setEntrance(Coordinate entrance) {  
 MainEngine.*entrance* = entrance;  
 }  
  
 public static ArrayList<Pair<Coordinate, Boolean>> getSteps() {  
 return *steps*;  
 }  
  
 public static int cellAmount() {  
 return *mazeHeight* \* *mazeWidth*;  
 }  
  
 private static int localCellAmount() {  
 return (*mazeWidth* \* 2 + 1) \* (*mazeHeight* \* 2 + 1);  
 }  
  
 public static boolean isBlindMode() {  
 return *blindMode*;  
 }  
  
 public static Coordinate getLocalCoordinate() {  
 return *localCoordinate*;  
 }  
  
 static boolean[][] getCurrentAdjacency() {  
 return *blindMode* ? *localAdjacencyMatrix* : *adjacencyMatrix*;  
 }  
  
 public static Coordinate getRTreasure() {  
 return *rTreasure*;  
 }  
  
 public static boolean treasureFound() {  
 return *treasure* != null;  
 }  
  
 public static boolean exitReached() {  
 return *exit* != null;  
 }  
  
 private static Direction calculateDirection() {  
  
 *current* = *blindMode* ? *localCoordinate* : *currentCell*.getCoordinate(); // coordinate where the player was BEFORE moving!!!  
  
 if (*firstStep*) {  
 int cutOff = *general*.getHorizontal() ? *mazeHeight* - 1 : *mazeWidth* - 1;  
 int point = *general*.getHorizontal() ? *current*.getY() : *current*.getX();  
 if ((point < cutOff && point > 0 || *blindMode*) && *initialShift* != 0) { // if not at the bottom and haven't met any obstacles on the way down yet  
 if (*initialShift* < 0) {  
 *initialShift*++;  
 return *general*.firstPerpendicular();  
 } else {  
 *initialShift*--;  
 return *general*.firstPerpendicular().opposite();  
 }  
 } else {  
 *firstStep* = false;  
 }  
 } else if (*lastCalculatedDirectionFailed*) { // Player met a wall in a cell they tried to reach using the lastTried direction  
 *lastCalculatedDirectionFailed* = false;  
 Direction probable;  
 switch (*failCount*) {  
 case 1: {  
 probable = *lastTried*.firstPerpendicular();  
 if (*handleDirectionFailure*(probable)) return probable;  
 }  
 case 2: {  
 probable = *lastTried*.firstPerpendicular().opposite();  
 if (*handleDirectionFailure*(probable)) return probable;  
 }  
 case 3: {  
 return *lastTried*.opposite();  
 }  
 }  
 } else if (*blindMode* && *mBorder*) {  
 if (!*shift*) {  
 *shift* = true;  
 *general* = *lastTried*;  
 return *lastTried*.firstPerpendicular();  
 } else {  
 *shift* = false;  
 *mBorder* = false;  
 *general* = *general*.opposite();  
 }  
 } else if (!*blindMode*) {  
 Coordinate wouldBe = *current*.add(*general*);  
 if (!wouldBe.fits()) {  
 if (*shift*) {  
 *general* = *general*.opposite();  
 *shift* = false;  
 } else {  
 *shift* = true;  
 return *general*.firstPerpendicular();  
 }  
 }  
 }  
  
 return *general*;  
 }  
  
 private static boolean handleDirectionFailure(Direction probable) {  
 Coordinate wouldBe = *current*.add(probable);  
 if (wouldBe.fitsLocally() || wouldBe.fits()) {  
 Coordinate.CoordinateState probableState = wouldBe.getCoordinateState();  
 if (probableState == Coordinate.CoordinateState.*UNKNOWN* || probableState == Coordinate.CoordinateState.*KNOWN\_REACHABLE*) {  
 return true;  
 }  
 }  
 *failCount*++;  
 return false;  
 }  
  
 private static Direction finalCheck(Direction start) {  
 *impossibleDirections* = 0;  
  
 Direction[] variants = {start,  
 start.firstPerpendicular(),  
 start.firstPerpendicular().opposite(),  
 start.opposite()};  
  
 for (Direction direction : variants) {  
 Coordinate wouldBe = *current*.add(direction);  
 if (wouldBe.fits() || wouldBe.fitsLocally()) {  
 Coordinate.CoordinateState state = wouldBe.getCoordinateState();  
 if (state == Coordinate.CoordinateState.*UNKNOWN* || state == Coordinate.CoordinateState.*KNOWN\_REACHABLE*) {  
 return direction;  
 }  
 if (state == Coordinate.CoordinateState.*KNOWN\_UNREACHABLE* || state == Coordinate.CoordinateState.*KNOWN\_MAZE\_BORDER* || state == Coordinate.CoordinateState.*KNOWN\_BAD\_PORTAL*) {  
 *impossibleDirections*++;  
 }  
 } else {  
 *impossibleDirections*++;  
 }  
 }  
  
 return null;  
 }  
  
 private static boolean successfulMoveScenario() {  
 *failCount* = 0;  
 *steps*.add(new Pair<>(*newRealCoordinate*, true));  
  
 *neighbours*(*blindMode* ? *newLocalCoordinate* : *newRealCoordinate*, *newRealCoordinate*);  
  
 Coordinate fromC = *blindMode* ? *localCoordinate* : *oldRealCoordinate*;  
 int from = fromC.getRawNumber();  
 int to = *blindMode* ? *newLocalCoordinate*.getRawNumber() : *newRealCoordinate*.getRawNumber();  
 boolean[][] adj = *blindMode* ? *localAdjacencyMatrix* : *adjacencyMatrix*;  
 Coordinate.CoordinateState state = fromC.getCoordinateState();  
  
 *currentCell* = *newRealCell*;  
  
 adj[from][to] = true;  
 boolean both = false;  
 if (state != Coordinate.CoordinateState.*KNOWN\_PORTAL*) {  
 both = true;  
 adj[to][from] = true;  
 }  
 if (state == Coordinate.CoordinateState.*KNOWN\_PORTAL* && !*blindMode*) {  
 HashMap<Coordinate, Integer> nop = UIHandler.*getNumsOfPortals*();  
 int cn = nop.get(fromC);  
 int cond = cn == 0 ? nop.size() : cn;  
 Coordinate distantPortal = new Coordinate(0, 0);  
 Iterator<Map.Entry<Coordinate, Integer>> iter = nop.entrySet().iterator();  
 while (cond != 0) {  
 distantPortal = iter.next().getKey();  
 cond--;  
 }  
 for (Direction direction : Direction.*values*()) {  
 int neighbour = fromC.add(direction).getRawNumber();  
 if (adj[neighbour][distantPortal.getRawNumber()]) {  
 adj[to][distantPortal.getRawNumber()] = true;  
 }  
 }  
 }  
 if (!*blindMode*) {  
 *newRealCoordinate*.setCoordinateState(Coordinate.CoordinateState.*KNOWN\_REACHABLE*, null);  
 } else {  
 *localAdjacencyLinks*.add(new AdjLink(*oldRealCoordinate*.copy(), *newRealCoordinate*.copy(), both));  
 *localPath*.add(new Pair<>(*newRealCoordinate*, Coordinate.CoordinateState.*KNOWN\_REACHABLE*));  
 *localCoordinate* = *newLocalCoordinate*;  
 *newLocalCoordinate*.setCoordinateState(Coordinate.CoordinateState.*KNOWN\_REACHABLE*, null);  
 }  
  
 switch (*currentCell*.getCellType()) {  
 case *TREASURE*: {  
 if (*blindMode* && *treasure* != null && *treasure*.getB().equals(LocalsTree.*root*)) { // if the treasure's been found while in non-blind mode, than it can be used as marker to unblind  
 LocalsTree.*unblind*();  
 }  
 if (*treasure* == null) {  
 *treasure* = new Pair<>(*blindMode* ? *newLocalCoordinate* : *newRealCoordinate*, *currentLevel*);  
 *rTreasure* = *newRealCoordinate*;  
 }  
 if (*exit* != null) {  
 if (*exit*.getB().equals(*currentLevel*)) {  
 try {  
 ArrayList<Integer> path =  
 *bfs*(*blindMode* ? *newLocalCoordinate* : *newRealCoordinate*, *exit*.getA(), adj);  
 *manageBFSPath*(path, *newRealCoordinate*, *newLocalCoordinate*);  
 return true;  
 } catch (NullPointerException e) {  
 *inSearchForExit* = true;  
 return false;  
 }  
 }  
  
 LocalsTree.Level levelOfExit = *exit*.getB();  
 Stack<LocalsTree.Level> pathToExit = *findPathToExit*(levelOfExit, *currentLevel*);  
  
 LocalsTree.Level currentLvl = *currentLevel*;  
 Coordinate currentC = *blindMode* ? *newLocalCoordinate* : *newRealCoordinate*;  
 if (pathToExit != null) {  
 while (!pathToExit.isEmpty()) {  
 LocalsTree.Level next = pathToExit.pop();  
 Coordinate nextC = new Coordinate(0, 0);  
 for (Map.Entry<LocalsTree.Level, Coordinate> entry : currentLvl.children.entrySet()) {  
 if (next.equals(entry.getKey())) {  
 nextC = entry.getValue().add(next.last.opposite());  
 }  
 }  
 boolean[][] cAdj = currentLvl.equals(LocalsTree.*root*) ? *adjacencyMatrix* : currentLvl.lAdjacencyMatrix;  
 *blindMode* = !currentLvl.equals(LocalsTree.*root*);  
  
 ArrayList<Integer> path = *bfs*(currentC, nextC, cAdj);  
 *manageBFSPath*(path, currentLvl.gPortalFromParent, currentLvl.portalFromParent);  
  
 currentLvl = next;  
 currentC = currentLvl.portalFromParent;  
 }  
  
 *blindMode* = !currentLvl.equals(LocalsTree.*root*);  
 ArrayList<Integer> path = *bfs*(currentC, *exit*.getA(), currentLvl.lAdjacencyMatrix);  
 *manageBFSPath*(path, currentLvl.gPortalFromParent, currentLvl.portalFromParent);  
 return true;  
 }  
 }  
 break;  
 }  
  
 case *ENTRANCE*: {  
 LocalsTree.*unblind*();  
 break;  
 }  
  
 case *EXIT*: {  
  
 if (*exit* != null && *blindMode* && *exit*.getB().equals(LocalsTree.*root*)) {  
 LocalsTree.*unblind*();  
 }  
  
 if (*exit* == null) {  
 *exit* = new Pair<>(*blindMode* ? *newLocalCoordinate* : *newRealCoordinate*, *currentLevel*);  
 }  
  
 if (*treasure* != null) {  
 return true;  
 }  
  
 *currentLevel*.pathToExit = true;  
  
 break;  
 }  
 }  
  
 return false;  
 }  
  
 private static void manageBFSPath(ArrayList<Integer> path, Coordinate subtractFrom, Coordinate toSubtract) {  
 if (*blindMode*) {  
 Coordinate relation = subtractFrom.subtract(toSubtract);  
 for (int i = 1; i < path.size(); i++) {  
 Coordinate localByRaw = Coordinate.*getByRawNumber*(path.get(i));  
 Coordinate realByLocal = localByRaw.add(relation.getX(), relation.getY());  
 *steps*.add(new Pair<>(realByLocal, true));  
 }  
 } else {  
 for (int i = 1; i < path.size(); i++) {  
 Coordinate c = Coordinate.*getByRawNumber*(path.get(i));  
 *steps*.add(new Pair<>(c, true));  
 }  
 }  
 }  
  
 private static Stack<LocalsTree.Level> findPathToExit(@NotNull LocalsTree.Level current,  
 @NotNull LocalsTree.Level searching) {  
 Stack<LocalsTree.Level> path = new Stack<>();  
  
 if (current.parent != null) {  
 path.push(current);  
 }  
  
 LocalsTree.Level curr = current.parent == null ? current : current.parent;  
 boolean found = current.parent != null && curr.equals(searching);  
  
 while (!curr.equals(searching) && curr.parent != null) {  
 path.push(curr);  
 curr = curr.parent;  
 if (curr.equals(searching)) {  
 found = true;  
 }  
 }  
  
 if (found) {  
 return path;  
 }  
  
 if (curr.parents != null) {  
 path.push(curr);  
 ArrayList<LocalsTree.Level> probableAdditionToPath = new ArrayList<>();  
  
 for (LocalsTree.Level p : curr.parents) {  
 probableAdditionToPath.clear();  
 LocalsTree.Level lvl = p;  
  
 while (!lvl.equals(searching) && lvl.parent != null) {  
 probableAdditionToPath.add(lvl);  
 lvl = lvl.parent;  
 if (lvl.equals(searching)) {  
 found = true;  
 }  
 }  
  
 if (found) {  
 for (LocalsTree.Level l : probableAdditionToPath) {  
 path.push(l);  
 }  
 return path;  
 }  
 }  
 }  
  
 return null;  
 }  
  
 private static void neighbours(Coordinate center, Coordinate realCenter) {  
 int cRaw = center.getRawNumber();  
  
 Coordinate relation = realCenter.subtract(center);  
  
 boolean[][] adj = *blindMode* ? *localAdjacencyMatrix* : *adjacencyMatrix*;  
  
 for (Direction direction : Direction.*values*()) {  
 Coordinate neighbour = center.add(direction);  
 int nRaw = neighbour.getRawNumber();  
 if (neighbour.fitsLocally() || neighbour.fits()) {  
 Coordinate.CoordinateState nState = neighbour.getCoordinateState();  
 Coordinate.CoordinateState cState = center.getCoordinateState();  
 boolean both = false;  
 boolean a = false;  
 boolean inverse = false;  
 if (nState == Coordinate.CoordinateState.*KNOWN\_REACHABLE*) {  
 adj[cRaw][nRaw] = true;  
 if (cState != Coordinate.CoordinateState.*KNOWN\_PORTAL* && cState != Coordinate.CoordinateState.*KNOWN\_PORTAL\_TO\_EXIT*) {  
 adj[nRaw][cRaw] = true;  
 both = true;  
 }  
 a = true;  
 }  
 if ((nState == Coordinate.CoordinateState.*KNOWN\_PORTAL* || nState == Coordinate.CoordinateState.*KNOWN\_PORTAL\_TO\_EXIT*)  
 && cState != Coordinate.CoordinateState.*KNOWN\_PORTAL* && cState != Coordinate.CoordinateState.*KNOWN\_PORTAL\_TO\_EXIT*) {  
 adj[nRaw][cRaw] = true;  
 inverse = true;  
 a = true;  
 }  
 if (a && *blindMode*) {  
 Coordinate rNeighbour = neighbour.add(relation.getX(), relation.getY());  
 AdjLink toAdd = inverse  
 ? new AdjLink(rNeighbour, realCenter.copy(), false)  
 : new AdjLink(realCenter.copy(), rNeighbour, both);  
 *localAdjacencyLinks*.add(toAdd);  
 }  
 }  
 }  
 }  
  
 private static boolean makeMove() {  
 if (*inSearchForExit* && *exit* != null && *exit*.getB().equals(*currentLevel*)) {  
 try {  
 ArrayList<Integer> path =  
 *bfs*(*blindMode* ? *newLocalCoordinate* : *newRealCoordinate*, *exit*.getA(), *blindMode* ? *localAdjacencyMatrix* : *adjacencyMatrix*);  
 *manageBFSPath*(path, *newRealCoordinate*, *newLocalCoordinate*);  
 return true;  
 } catch (NullPointerException ignored) {  
 }  
 }  
 Direction dir = *moment* == null ? *finalCheck*(*calculateDirection*()) : *moment*; // if we've just used some method that gives us the proper Direction for this moment, we don't need to calculate it  
  
 if (dir == null) { // no more cells to try  
 if (*impossibleDirections* == 4) { // not even a not-KNOWN\_BAD portal nearby  
 return *block*();  
 } else {  
 Coordinate p = *blindMode* ? new Coordinate(*mazeWidth*, *mazeHeight*)  
 : *currentLevel*.gPortalFromParent;  
 try {  
 Coordinate move = *blindMode* ? p.subtract(*localCoordinate*)  
 : p.subtract(*currentCell*.getCoordinate());  
 *moment* = Direction.*getByConstructor*(move.getX(), move.getY());  
 p.setCoordinateState(Coordinate.CoordinateState.*UNKNOWN*, null);  
 } catch (Exception e) {  
 for (Direction direction : Direction.*values*()) {  
 if (*currentCell*.getCoordinate().add(direction).getCoordinateState() == Coordinate.CoordinateState.*KNOWN\_PORTAL*) {  
 *moment* = direction;  
 *currentCell*.getCoordinate().add(direction).setCoordinateState(Coordinate.CoordinateState.*UNKNOWN*, null);  
 }  
 }  
 }  
 }  
 return false;  
 }  
 *moveResult* = *currentCell*.move(dir);  
 *moment* = null;  
  
  
 // Old local Coordinate is localCoordinate itself  
 *newLocalCoordinate* = *localCoordinate*.add(dir);  
  
 // Old real Cell is currentCell itself  
 *newRealCell* = *moveResult*.getResult();  
  
 *oldRealCoordinate* = *currentCell*.getCoordinate();  
 *newRealCoordinate* = *newRealCell*.getCoordinate();  
  
 switch (*moveResult*) {  
 case *SUCCESSFUL*: { // does count as a step  
 return *successfulMoveScenario*();  
 }  
  
 case *UNREACHABLE\_CELL*: { // does count as a step  
 Coordinate forCalculation = *blindMode* ? *newLocalCoordinate* : *newRealCoordinate*;  
 forCalculation.setCoordinateState(Coordinate.CoordinateState.*KNOWN\_UNREACHABLE*, null);  
 *steps*.add(new Pair<>(*newRealCoordinate*.copy(), true)); // in steps, we always store real Coordinates to then show them to the User  
 if (*blindMode*) {  
 *localPath*.add(new Pair<>(*newRealCoordinate*.copy(), Coordinate.CoordinateState.*KNOWN\_UNREACHABLE*));  
 }  
 *lastCalculatedDirectionFailed* = true;  
 if (*failCount*++ == 0) {  
 *lastTried* = dir;  
 }  
 *firstStep* = false;  
 *shift* = false;  
  
 if (*failCount* >= 4) {  
 if (*block*()) return true;  
 }  
 break;  
 }  
  
 case *ALREADY\_VISITED\_CELL*: { // does not count as a step  
  
 *neighbours*(*blindMode* ? *localCoordinate* : *oldRealCoordinate*, *oldRealCoordinate*);  
  
 *failCount* = 0;  
 RadialCheck radialCheck = new RadialCheck(*blindMode* ? *localCoordinate* : *oldRealCoordinate*); // searches from a coordinate where the Player was before trying to make move  
 Pair<Coordinate, Direction> d = radialCheck.find();  
 if (d == null) { // if there are no more unknown cells possible to visit  
 if (*blindMode*) {  
 Coordinate relation = *oldRealCoordinate*.subtract(*localCoordinate*);  
 if (!*currentLevel*.triedEscapingThroughEntrance()) { // if in blindMode, the Player should first try escaping according to rules through the same portal they've got to the currentLevel  
 Coordinate pfp = *currentLevel*.portalFromParent;  
 ArrayList<Integer> temp = new ArrayList<>(4);  
 int pfpN = pfp.getRawNumber();  
 for (Direction direction : Direction.*values*()) {  
 Coordinate t = pfp.add(direction);  
 if (t.fitsLocally()  
 && t.getCoordinateState() == Coordinate.CoordinateState.*KNOWN\_REACHABLE*) {  
 int tN = t.getRawNumber();  
 temp.add(tN);  
 *localAdjacencyMatrix*[tN][pfpN] = true;  
 }  
 }  
 ArrayList<Integer> pathToLastEnteringPortal =  
 *bfs*(*localCoordinate*, *currentLevel*.portalFromParent, *localAdjacencyMatrix*);  
 for (int t : temp) {  
 *localAdjacencyMatrix*[t][pfpN] = false;  
 }  
 int s = pathToLastEnteringPortal.size();  
 // last move in the calculated path must be made through makeMove  
 if (s == 1) { // if the Player only needs to make one step to reach the portal  
 Coordinate move = *currentLevel*.portalFromParent.subtract(*localCoordinate*);  
 *moment* = Direction.*getByConstructor*(move.getX(), move.getY());  
 } else {  
 Coordinate preLast = null;  
 for (int i = 0; i < s; i++) {  
 Coordinate localByRaw = Coordinate.*getByRawNumber*(pathToLastEnteringPortal.get(i));  
 Coordinate realByLocal = localByRaw.add(relation.getX(), relation.getY());  
 if (i == s - 2) {  
 preLast = realByLocal;  
 *currentCell* = *maze*[realByLocal.getY()][realByLocal.getX()];  
 *localCoordinate* = localByRaw;  
 }  
 if (i < s - 1) {  
 if (i != 0) {  
 *steps*.add(new Pair<>(realByLocal, true));  
 }  
 } else {  
 assert preLast != null;  
 Coordinate lastMove = realByLocal.subtract(preLast);  
 *moment* = Direction.*getByConstructor*(lastMove.getX(), lastMove.getY());  
 *lastTried* = *moment*;  
 }  
 }  
 }  
 // changing back to UNKNOWN so that moveResult.PORTAL triggers  
 *currentLevel*.portalFromParent.setCoordinateState(Coordinate.CoordinateState.*UNKNOWN*, null);  
 } else {  
 if (*treasure* != null && *treasure*.getB().equals(*currentLevel*)) { // if it turns out that the treasure is only reachable through this Level and this Level can't be left normally, than the game is unwinnable  
 return true;  
 }  
 LocalsTree.*previousState*();  
 }  
 } else {  
 return true;  
 }  
 } else {  
 Coordinate nowGoingTo = d.getA();  
 *moment* = d.getB();  
  
 *lastTried* = *moment*;  
  
 Coordinate forCalculation = *blindMode* ? *localCoordinate* : *oldRealCoordinate*;  
 if (!nowGoingTo.equals(forCalculation)) {  
  
 ArrayList<Integer> path =  
 *bfs*(forCalculation, nowGoingTo, *blindMode* ? *localAdjacencyMatrix* : *adjacencyMatrix*);  
  
 if (*blindMode*) {  
 Coordinate relation = *oldRealCoordinate*.subtract(*localCoordinate*);  
 for (int num : path) {  
 Coordinate localByRaw = Coordinate.*getByRawNumber*(num);  
 Coordinate realByLocal = localByRaw.add(relation.getX(), relation.getY());  
 *steps*.add(new Pair<>(realByLocal, true));  
 }  
 *localCoordinate* = nowGoingTo;  
 Coordinate curr = nowGoingTo.add(relation.getX(), relation.getY());  
 *currentCell* = *maze*[curr.getY()][curr.getX()];  
 } else {  
 for (Integer num : path) {  
 *steps*.add(new Pair<>(Coordinate.*getByRawNumber*(num), true));  
 }  
 *currentCell* = *maze*[nowGoingTo.getY()][nowGoingTo.getX()];  
 }  
 }  
 }  
  
 break;  
 }  
  
 // can only happen in blindMode  
 case *MAZE\_BORDER*: {  
 Coordinate overTheBorder = *oldRealCoordinate*.add(dir);  
  
 if (!dir.getHorizontal()) {  
 *yCoordinateDefined* = true;  
 } else {  
 *xCoordinateDefined* = true;  
 }  
  
 *firstStep* = false;  
 *mBorder* = true;  
 *shift* = false;  
 *failCount*++;  
  
 *lastTried* = dir;  
  
 *steps*.add(new Pair<>(overTheBorder, true));  
 *newLocalCoordinate*.setCoordinateState(Coordinate.CoordinateState.*KNOWN\_MAZE\_BORDER*, dir);  
  
 if (*yCoordinateDefined* && *xCoordinateDefined*) {  
 LocalsTree.*unblind*();  
 }  
  
 break;  
 }  
  
 case *PORTAL*: {  
 Coordinate inPortal = *blindMode* ? *localCoordinate*.add(dir) : *oldRealCoordinate*.add(dir);  
  
 *steps*.add(new Pair<>(*oldRealCoordinate*, true));  
 *pseudo*.push(*steps*.size());  
 *steps*.add(new Pair<>(inPortal, true));  
 *steps*.add(new Pair<>(*newRealCoordinate*, true));  
  
 inPortal.setCoordinateState(Coordinate.CoordinateState.*KNOWN\_PORTAL*, null);  
 *portalStack*.push(new Pair<>(*oldRealCoordinate*.add(dir), *blindMode* ? *localCoordinate*.add(dir) : null));  
 LocalsTree.*add*(dir);  
  
 *localCoordinate* = new Coordinate(*mazeWidth*, *mazeHeight*);  
 *currentCell* = *newRealCell*;  
 *clearLocals*();  
 *localCoordinate*.setCoordinateState(Coordinate.CoordinateState.*KNOWN\_PORTAL*, null);  
 }  
 }  
  
 return false;  
 }  
  
 private static boolean block() {  
 if ((*treasure* == null || !*treasure*.getB().equals(*currentLevel*)) && !*portalStack*.isEmpty()) {  
 LocalsTree.*previousState*();  
 } else {  
 if (*treasure* != null && *treasure*.getB().equals(*currentLevel*)) {  
 *treasure* = null;  
 *rTreasure* = null;  
 }  
 return true;  
 }  
 return false;  
 }  
  
 private static void clearLocals() {  
  
 for (int i = 0; i < *localCellAmount*(); i++) {  
 for (int j = 0; j < *localCellAmount*(); j++) {  
 *localAdjacencyMatrix*[i][j] = false;  
 }  
 }  
  
 *firstStep* = true;  
  
 Random random = new Random();  
 int directionIndex = random.nextInt(2) + *range*;  
 if (directionIndex > 3) {  
 directionIndex = 0;  
 }  
  
 *general* = Direction.*values*()[directionIndex];  
 int bound = *mazeWidth* <= 9 ? 3 : *mazeWidth* / 3;  
 *initialShift* = random.nextInt(bound) \* (random.nextInt(3) - 1);  
 *shift* = false;  
 *failCount* = 0;  
 Coordinate.*clearLocalCoordinateStates*();  
 *yCoordinateDefined* = false;  
 *xCoordinateDefined* = false;  
 *localPath*.clear();  
 *localAdjacencyLinks*.clear();  
 *mBorder* = false;  
 }  
  
 private static ArrayList<Integer> bfs(Coordinate startC, Coordinate destC, boolean[][] matrixToUse) {  
 int start = startC.getRawNumber();  
 int dest = destC.getRawNumber();  
  
 int actualSize = *blindMode* ? *localCellAmount*() : *cellAmount*();  
 ArrayList<ArrayList<Integer>> paths = new ArrayList<>(actualSize);  
 for (int i = 0; i < actualSize; i++) {  
 paths.add(new ArrayList<>());  
 }  
  
 boolean[] visited = new boolean[actualSize];  
 visited[start] = true;  
  
 ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<>(actualSize);  
 queue.add(start);  
  
 while (!queue.isEmpty()) {  
 int a = queue.poll();  
 for (int i = 0; i < actualSize; i++) {  
 if (matrixToUse[a][i] && !visited[i]) {  
 paths.set(i, new ArrayList<>(paths.get(a)));  
 paths.get(i).add(i);  
 if (i == dest) {  
 return paths.get(i);  
 }  
 visited[i] = true;  
 queue.add(i);  
 }  
 }  
 }  
  
 throw new NullPointerException("No path from " + startC.toString() + " to " + destC.toString());  
 }  
  
 public static void solve(int range) {

*exit* = null;  
 *treasure* = null;  
 *currentCell* = *maze*[*entrance*.getY()][*entrance*.getX()];  
 *adjacencyMatrix* = new boolean[*cellAmount*()][*cellAmount*()];  
 *localAdjacencyMatrix* = new boolean[*localCellAmount*()][*localCellAmount*()];  
 *firstStep* = true;  
 *initialShift* = 0;  
 *shift* = false;  
 *general* = Direction.*UP*;  
 *steps* = new ArrayList<>(Collections.*singletonList*(new Pair<>(*entrance*, true)));  
 *moveResult* = null;  
 *lastTried* = *general*;  
 *failCount* = 0;  
 *lastCalculatedDirectionFailed* = false;  
 *current* = null;  
 *moment* = null;  
 *blindMode* = false;  
 *localCoordinate* = new Coordinate(*mazeWidth*, *mazeHeight*);  
 *yCoordinateDefined* = true;  
 *xCoordinateDefined* = true;  
 *localPath* = new ArrayList<>();  
 *localAdjacencyLinks* = new ArrayList<>();  
 *levelId* = 0;  
 LocalsTree.*clear*();  
 *newLocalCoordinate* = null;  
 *oldRealCoordinate* = null;  
 *newRealCell* = null;  
 *newRealCoordinate* = null;  
 *currentLevel* = LocalsTree.*root*;  
 *mBorder* = false;  
 Coordinate.*setNewField*();  
 *localPath* = new ArrayList<>();  
 *impossibleDirections* = 0;  
 *rTreasure* = null;  
 MainEngine.*range* = range;  
 *portalStack* = new Stack<>();  
 *inSearchForExit* = false;  
 *pseudo* = new Stack<>();  
  
 boolean completed = false;  
  
 while (!completed) {  
 completed = *makeMove*();  
 }  
 }  
}

**MazeGrid.java**

package greenbeaver.terraincognita.model;  
  
import greenbeaver.terraincognita.model.cellConstruction.Cell;  
import greenbeaver.terraincognita.model.cellConstruction.CellType;  
import greenbeaver.terraincognita.model.cellConstruction.Coordinate;  
import javafx.scene.layout.GridPane;  
  
public class MazeGrid extends GridPane {  
  
 private Cell[][] mazeAsArray;  
  
 public MazeGrid() {  
 setGridLinesVisible(true);  
 CellType.*flush*();  
 mazeAsArray = new Cell[MainEngine.*getMazeHeight*()][MainEngine.*getMazeWidth*()];  
  
 for (int i = 0; i < MainEngine.*getMazeWidth*(); i++) {  
 for (int j = 0; j < MainEngine.*getMazeHeight*(); j++) {  
 Cell cell = new Cell(new Coordinate(i, j));  
 add(cell, i, j);  
 mazeAsArray[j][i] = cell;  
 }  
 }  
 }  
  
 public Cell[][] getMazeAsArray() {  
 return mazeAsArray;  
 }  
}

**Pair.java**

package greenbeaver.terraincognita.model;  
  
public class Pair<T1, T2> {  
 private final T1 a;  
 private T2 b;  
  
 public Pair(T1 a, T2 b) {  
 this.a = a;  
 this.b = b;  
 }  
  
 public T1 getA() {  
 return a;  
 }  
  
 public T2 getB() {  
 return b;  
 }  
  
 public void setB(T2 b) {  
 this.b = b;  
 }  
}

**RadialCheck.java**

package greenbeaver.terraincognita.model;  
  
import greenbeaver.terraincognita.model.cellConstruction.Coordinate;  
import greenbeaver.terraincognita.model.cellConstruction.Direction;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
  
class RadialCheck {  
 private final Coordinate initial;  
 private final ArrayList<ArrayList<Coordinate>> sides;  
  
 private final boolean[][] adjacency;  
 private final boolean[] visited;  
 private boolean marker;  
  
 private Pair<Coordinate, Direction> priority1;  
 private Pair<Coordinate, Direction> priority2;  
 private Pair<Coordinate, Direction> priority3;  
  
 private enum ValueFound {  
 *PRIORITY\_1*,  
 *PRIORITY\_2*,  
 *PRIORITY\_3*,  
 *NONE* }  
  
 RadialCheck(Coordinate initial) {  
 this.initial = initial;  
 sides = new ArrayList<>(4);  
 for (int i = 0; i < 4; i++) {  
 sides.add(new ArrayList<>());  
 }  
 priority1 = null;  
 priority2 = null;  
 priority3 = null;  
  
 adjacency = MainEngine.*getCurrentAdjacency*();  
 visited = new boolean[adjacency.length];  
 }  
  
 private boolean dfs(Coordinate start, Coordinate searching) {  
 Arrays.*fill*(visited, false);  
 marker = false;  
  
 dfs(start.getRawNumber(), searching.getRawNumber());  
  
 return marker;  
 }  
  
 private void dfs(int start, int searching) {  
  
 if (marker) {  
 return;  
 }  
  
 visited[start] = true;  
  
 if (start == searching) {  
 marker = true;  
 return;  
 }  
  
 for (int i = 0; i < adjacency.length; i++) {  
 if (adjacency[start][i] && !visited[i]) {  
 dfs(i, searching);  
 }  
 }  
 }  
  
 // checks if the probable Coordinate can be reached from the coordinate that's to the (direction) from it  
 private Pair<Coordinate, Direction> calculateCorner(Direction direction, Coordinate probable) {  
 Coordinate check = probable.add(direction);  
 if (suitable(check)) {  
 return new Pair<>(check, direction.opposite());  
 }  
  
 return null;  
 }  
  
 private ValueFound addCorners(int level, boolean startOfLine) {  
 int[][] corners = {{-level, -level}, {level, -level}, {level, level}, {-level, level}};  
  
 for (int i = 0; i < 4; i++) {  
 Coordinate probable = initial.add(corners[i][0], corners[i][1]);  
 if (probable.fitsLocally() || probable.fits()) {  
 Direction second;  
 Direction first;  
 switch (i) {  
 case 0: {  
 first = Direction.*RIGHT*;  
 second = Direction.*DOWN*;  
 break;  
 }  
  
 case 1: {  
 first = Direction.*DOWN*;  
 second = Direction.*LEFT*;  
 break;  
 }  
  
 case 2: {  
 first = Direction.*LEFT*;  
 second = Direction.*UP*;  
 break;  
 }  
 case 3: {  
 first = Direction.*UP*;  
 second = Direction.*RIGHT*;  
 break;  
 }  
 default:  
 throw new IllegalStateException("Unexpected value: " + i);  
 }  
  
 if (startOfLine) { // if true, this corner is added to the actual current side and checked for being an answer  
 if (probable.getCoordinateState() == Coordinate.CoordinateState.*UNKNOWN*) { // this block sets a low-priority answer if the currently examined corner is reachable from any side  
 Pair<Coordinate, Direction> frst = calculateCorner(first, probable);  
 if (frst != null) {  
 priority2 = frst;  
 return ValueFound.*PRIORITY\_2*;  
 }  
  
 Pair<Coordinate, Direction> scnd = calculateCorner(second, probable);  
 if (scnd != null) {  
 priority2 = scnd;  
 return ValueFound.*PRIORITY\_2*;  
 }  
  
 Pair<Coordinate, Direction> thrd = calculateCorner(first.opposite(), probable);  
 if (thrd != null) {  
 priority3 = thrd;  
 return ValueFound.*PRIORITY\_3*;  
 }  
  
 Pair<Coordinate, Direction> frth = calculateCorner(second.opposite(), probable);  
 if (frth != null) {  
 priority3 = frth;  
 return ValueFound.*PRIORITY\_3*;  
 }  
 }  
  
 sides.get(i).add(probable);  
 } else { // This is a second time we enter this for the same corner and thus only need to add it to its other neighbouring side. By this time we are already sure that this corner isn't an answer  
 int also = (i == 0) ? 3 : (i - 1);  
 sides.get(also).add(probable);  
 }  
 }  
 }  
 return ValueFound.*NONE*;  
 }  
  
 // fills the "cross" around the initial element, immediately returns if an unknown cell is found  
 private ValueFound initialFill() {  
 for (int i = 0; i < 4; i++) {  
 Direction direction = Direction.*values*()[i];  
 Coordinate probable = initial.add(direction);  
 if (probable.fitsLocally() || probable.fits()) {  
 if (probable.getCoordinateState() == Coordinate.CoordinateState.*UNKNOWN*) {  
 priority1 = new Pair<>(initial, direction);  
 return ValueFound.*PRIORITY\_1*;  
 }  
  
 sides.get(i).add(probable);  
 }  
 }  
 addCorners(1, false);  
  
 return ValueFound.*NONE*;  
 }  
  
 private void updateLines(int level) {  
 for (int i = 0; i < 4; i++) {  
 Direction direction = Direction.*values*()[i]; // Up -> Right -> Down -> Left  
 ArrayList<Coordinate> current = sides.get(i);  
 ArrayList<Coordinate> currentCopy = new ArrayList<>(current); // save the (level - 1) state of the side  
 current.clear();  
 addCorners(level, true);  
 for (Coordinate from : currentCopy) {  
 Coordinate probable = from.add(direction);  
  
 if (probable.fitsLocally() || probable.fits()) {  
 if (probable.getCoordinateState() == Coordinate.CoordinateState.*UNKNOWN*) {  
  
 if (suitable(from)) {  
 priority1 = new Pair<>(from, direction);  
 return;  
 }  
  
 Direction p = direction.firstPerpendicular(); // probable coordinate might not be reachable going straight radially from the center, but if it has a "bridge" neighbour in the same ring, it would still be better than corner  
 if (suitable(probable.add(p))) {  
 priority2 = new Pair<>(probable.add(p), p.opposite());  
 } else if (suitable(probable.add(p.opposite()))) {  
 priority2 = new Pair<>(probable.add(p.opposite()), p);  
 } else if (suitable(probable.add(direction))) {  
 priority3 = new Pair<>(probable.add(direction), direction.opposite());  
 }  
 }  
  
 current.add(probable);  
 }  
 }  
 }  
  
 addCorners(level, false);  
 }  
  
 private boolean suitable(Coordinate toCheck) {  
 if (toCheck.fitsLocally() || toCheck.fits()) {  
 Coordinate.CoordinateState state = toCheck.getCoordinateState();  
 return  
 (state == Coordinate.CoordinateState.KNOWN\_REACHABLE  
 || state == Coordinate.CoordinateState.KNOWN\_PORTAL\_TO\_EXIT  
 || state == Coordinate.CoordinateState.KNOWN\_PORTAL)  
 && dfs(initial, toCheck);  
 }  
 return false;  
 }  
  
 Pair<Coordinate, Direction> find() {  
 ValueFound firstTry = addCorners(1, true);  
  
 ValueFound secondTry = initialFill();  
  
 if (secondTry == ValueFound.PRIORITY\_1) {  
 return priority1;  
 }  
  
 switch (firstTry) {  
 case PRIORITY\_2:  
 return priority2;  
 case PRIORITY\_3:  
 return priority3;  
 }  
  
 int h = MainEngine.isBlindMode() ? MainEngine.getMazeHeight() \* 2 + 1 : MainEngine.getMazeHeight();  
 int w = MainEngine.isBlindMode() ? MainEngine.getMazeWidth() \* 2 + 1 : MainEngine.getMazeWidth();  
  
 int distanceToBottom = h - initial.getY();  
 int distanceToRight = w - initial.getX();  
  
 int maxHDistance = Math.max(distanceToRight, initial.getX());  
 int maxVDistance = Math.max(distanceToBottom, initial.getY());  
  
 int cutoff = Math.max(maxHDistance, maxVDistance);  
  
 int level = 2;  
 while (priority1 == null && priority2 == null && priority3 == null) {  
 if (level > cutoff) {  
 return null;  
 }  
 updateLines(level++);  
 }  
  
 if (priority1 != null) {  
 return priority1;  
 }  
  
 return priority2 != null ? priority2 : priority3;  
 }  
}

**UIHandler.java**

package greenbeaver.terraincognita.model;  
  
import greenbeaver.terraincognita.model.cellConstruction.CellType;  
import greenbeaver.terraincognita.model.cellConstruction.Coordinate;  
  
import java.util.HashMap;  
import java.util.Map;  
  
public class UIHandler {  
 private static Coordinate *currentPortal*;  
 private static boolean *continueWithDangerousInput*;  
 private static final HashMap<Coordinate, Integer> *numsOfPortals*;  
 private static final int[] *amounts*;  
 static {  
 *numsOfPortals* = new HashMap<>();  
 *amounts* = new int[10];  
 }  
  
 public static void clearUIHandler() {  
 *continueWithDangerousInput* = false;  
 *currentPortal* = null;  
 *numsOfPortals*.clear();  
 for (int i = 0; i < 10; i++) {  
 *amounts*[i] = 0;  
 }  
 }  
  
 public static Coordinate getCurrentPortal() {  
 return *currentPortal*;  
 }  
  
 public static int getCurrentPortalNum() {  
 return *numsOfPortals*.get(*currentPortal*);  
 }  
  
 public static void setCurrentPortal(Coordinate currentPortal) {  
 UIHandler.*currentPortal* = currentPortal;  
 }  
  
 public static void createPortal(Coordinate coordinate) {  
 *numsOfPortals*.put(coordinate, 0);  
 *amounts*[0]++;  
 }  
  
 public static void removePortal(Coordinate coordinate) {  
 *amounts*[*numsOfPortals*.get(coordinate)]--;  
 *numsOfPortals*.remove(coordinate);  
 }  
  
 public static void setPortalNum(Coordinate portal, int num) {  
 *amounts*[*numsOfPortals*.get(portal)]--;  
 *numsOfPortals*.replace(portal, num);  
 *amounts*[num]++;  
 }  
  
 public static boolean portalNumsOK() {  
 for (int i = 0; i < 10; i++) {  
 if (*amounts*[i] != 0 && *numsOfPortals*.size() <= i || *amounts*[i] > 1) {  
 return false;  
 }  
 }  
  
 return true;  
 }  
  
 public static Coordinate[] getPortalTransitions() {  
 int amount = CellType.*PORTAL*.getUsedAmount();  
 Coordinate[] transitions = new Coordinate[amount];  
 for (Map.Entry<Coordinate, Integer> entry: *numsOfPortals*.entrySet()) {  
 transitions[entry.getValue()] = entry.getKey();  
 }  
 return transitions;  
 }  
  
 public static int getNumOfPortal(Coordinate portal) {  
 return *numsOfPortals*.get(portal);  
 }  
  
 public static void setContinueWithDangerousInput(boolean continueWithDangerousInput) {  
 UIHandler.*continueWithDangerousInput* = continueWithDangerousInput;  
 }  
  
 public static boolean getContinueWithDangerousInput() {  
 return *continueWithDangerousInput*;  
 }  
  
 public static HashMap<Coordinate, Integer> getNumsOfPortals() {  
 return *numsOfPortals*;  
 }  
}

**Util.java**

package greenbeaver.terraincognita.model;  
  
import javafx.scene.image.Image;  
  
public class Util {  
 public static final Image *FLOOR* = new Image("/images/tiles/floor.png");  
 public static final Image *ENTRANCE* = new Image("/images/tiles/entrance.png");  
 public static final Image *EXIT* = new Image("/images/tiles/exit.png");  
 public static final Image *TREASURE* = new Image("/images/tiles/treasure.png");  
 public static final Image *WALL* = new Image("/images/tiles/wall.png");  
 public static final Image *PORTAL* = new Image("/images/tiles/portal.png");  
  
 public static final Image *H\_FLOOR* = new Image("/images/tiles/floorRealHighlight.png");  
 public static final Image *H\_ENTRANCE* = new Image("/images/tiles/entranceRealHighlight.png");  
 public static final Image *H\_EXIT* = new Image("/images/tiles/exitRealHighlight.png");  
 public static final Image *H\_TREASURE* = new Image("/images/tiles/treasureRealHighlight.png");  
 public static final Image *H\_WALL* = new Image("/images/tiles/wallRealHighlight.png");  
 public static final Image *H\_PORTAL* = new Image("/images/tiles/portalRealHighlight.png");  
  
 public static final Image *PH\_FLOOR* = new Image("/images/tiles/floorPseudoHighlight.png");  
 public static final Image *PH\_ENTRANCE* = new Image("/images/tiles/entrancePseudoHighlight.png");  
 public static final Image *PH\_EXIT* = new Image("/images/tiles/exitPseudoHighlight.png");  
 public static final Image *PH\_TREASURE* = new Image("/images/tiles/treasurePseudoHighlight.png");  
 public static final Image *PH\_WALL* = new Image("/images/tiles/wallPseudoHighlight.png");  
 public static final Image *PH\_PORTAL* = new Image("/images/tiles/portalPseudoHighlight.png");  
}

**Cell.java**

package greenbeaver.terraincognita.model.cellConstruction;  
  
import greenbeaver.terraincognita.model.MainEngine;  
import greenbeaver.terraincognita.model.UIHandler;  
import greenbeaver.terraincognita.model.Util;  
import javafx.fxml.FXMLLoader;  
import javafx.scene.Node;  
import javafx.scene.Parent;  
import javafx.scene.Scene;  
import javafx.scene.image.ImageView;  
import javafx.scene.input.MouseButton;  
import javafx.scene.input.MouseEvent;  
import javafx.stage.Modality;  
import javafx.stage.Stage;  
import javafx.stage.StageStyle;  
  
import java.io.IOException;  
  
public class Cell extends ImageView {  
  
 private CellType cellType;  
 private final Coordinate coordinate;  
  
 public Cell(Coordinate coordinate) {  
 super(Util.*FLOOR*);  
 this.coordinate = coordinate;  
 cellType = CellType.*EMPTY*;  
  
 setFitWidth(50);  
 setFitHeight(50);  
 setOnMouseClicked(event -> {  
 try {  
 onClick(event);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 });  
 }  
  
 public MoveResult move(Direction direction) {  
 Coordinate newCoordinate = coordinate.add(direction);  
  
 if (!newCoordinate.fits()) {  
 return MoveResult.*MAZE\_BORDER*;  
 }  
  
 Cell probableResult = MainEngine.*getMaze*()[newCoordinate.getY()][newCoordinate.getX()];  
  
 Coordinate temp = MainEngine.*isBlindMode*() ? MainEngine.*getLocalCoordinate*().add(direction) : newCoordinate;  
 Coordinate.CoordinateState state = temp.getCoordinateState();  
 if (probableResult.getCellType() == CellType.*PORTAL* && state == Coordinate.CoordinateState.*UNKNOWN*) {  
 Coordinate[] transitions = MainEngine.*getPortalTransitions*();  
 int portalIndex = UIHandler.*getNumOfPortal*(probableResult.coordinate);  
 Coordinate actualNewCoordinate = transitions[(portalIndex == transitions.length - 1) ? 0 : portalIndex + 1];  
 Cell actualResult = MainEngine.*getMaze*()[actualNewCoordinate.getY()][actualNewCoordinate.getX()];  
 MoveResult.*setResult*(actualResult);  
 return MoveResult.*PORTAL*;  
 } else {  
  
 MoveResult.*setResult*(probableResult);  
  
 if (!probableResult.getCellType().isReachable() || state == Coordinate.CoordinateState.*KNOWN\_BAD\_PORTAL*) {  
 return MoveResult.*UNREACHABLE\_CELL*;  
 }  
  
 return state == Coordinate.CoordinateState.*UNKNOWN* ? MoveResult.*SUCCESSFUL* : MoveResult.*ALREADY\_VISITED\_CELL*;  
 }  
 }  
  
 public CellType getCellType() {  
 return cellType;  
 }  
  
 public Coordinate getCoordinate() {  
 return coordinate;  
 }  
  
 private void onClick(MouseEvent event) throws IOException {  
 if (event.getButton() == MouseButton.*PRIMARY*) {  
 if (cellType == CellType.*PORTAL*) {  
 UIHandler.*removePortal*(coordinate);  
 }  
 cellType = cellType.switchType();  
  
 switch (cellType) {  
 case *PORTAL*: {  
 UIHandler.*createPortal*(coordinate);  
 break;  
 }  
  
 case *ENTRANCE*: {  
 MainEngine.*setEntrance*(coordinate);  
 break;  
 }  
 }  
  
 super.setImage(cellType.getImage());  
 } else if (event.getButton() == MouseButton.*SECONDARY* && cellType == CellType.*PORTAL*) {  
 UIHandler.*setCurrentPortal*(coordinate);  
 Stage numSettings = new Stage();  
 numSettings.initStyle(StageStyle.*TRANSPARENT*);  
 Parent root = FXMLLoader.*load*(getClass().getResource("/fxml/PortalSettings.fxml"));  
 numSettings.setScene(new Scene(root));  
 numSettings.initModality(Modality.*WINDOW\_MODAL*);  
 numSettings.initOwner(((Node) event.getSource()).getScene().getWindow());  
 numSettings.setX(event.getScreenX());  
 numSettings.setY(event.getScreenY());  
 numSettings.showAndWait();  
 }  
 }  
  
 public void highlight(boolean real) {  
 this.setImage(real ? cellType.getHImage() : cellType.getPhImage());  
 }  
}

**CellType.java**

package greenbeaver.terraincognita.model.cellConstruction;  
  
import greenbeaver.terraincognita.model.MainEngine;  
import greenbeaver.terraincognita.model.Util;  
import javafx.scene.image.Image;  
  
public enum CellType {  
 *TREASURE*(true, 1, 0, Util.*TREASURE*, Util.*H\_TREASURE*, Util.*PH\_TREASURE*),  
 *ENTRANCE*(true, 1, 0, Util.*ENTRANCE*, Util.*H\_ENTRANCE*, Util.*PH\_ENTRANCE*),  
 *EXIT*(true, 1, 0, Util.*EXIT*, Util.*H\_EXIT*, Util.*PH\_EXIT*),  
  
 *EMPTY*(true, MainEngine.*cellAmount*(), MainEngine.*cellAmount*(), Util.*FLOOR*, Util.*H\_FLOOR*, Util.*PH\_FLOOR*),  
 *WALL*(false, MainEngine.*cellAmount*(), 0, Util.*WALL*, Util.*H\_WALL*, Util.*PH\_WALL*),  
 *PORTAL*(true, 10, 0, Util.*PORTAL*, Util.*H\_PORTAL*, Util.*PH\_PORTAL*);  
  
 private final boolean reachable;  
 private int maxAmount;  
 private final Image image;  
 private final Image hImage;  
 private final Image phImage;  
 private int usedAmount;  
  
 CellType(boolean reachable, int maxAmount, int usedAmount, Image image, Image hImage, Image phImage) {  
 this.reachable = reachable;  
 this.maxAmount = maxAmount;  
 this.usedAmount = usedAmount;  
 this.image = image;  
 this.hImage = hImage;  
 this.phImage = phImage;  
 }  
  
 public boolean isReachable() {  
 return reachable;  
 }  
  
 private boolean unavailable() {  
 return usedAmount >= maxAmount;  
 }  
  
 public CellType switchType() {  
 usedAmount--;  
  
 int index = (ordinal() + 1 == *values*().length) ? 0 : ordinal() + 1;  
 while (*values*()[index].unavailable()) {  
 index = (index == *values*().length - 1) ? 0 : index + 1;  
 }  
  
 CellType actual = *values*()[index];  
 actual.usedAmount++;  
 return actual;  
 }  
  
 public Image getImage() {  
 return image;  
 }  
  
 public Image getHImage() {  
 return hImage;  
 }  
  
 public Image getPhImage() {  
 return phImage;  
 }  
  
 public enum FieldState {  
 *GOOD*,  
 *ONE\_PORTAL*,  
 *UNUSED\_ESSENTIALS* }  
  
 public static FieldState fieldFilled() {  
 if (*TREASURE*.unavailable() && *ENTRANCE*.unavailable() && *EXIT*.unavailable() && *PORTAL*.usedAmount != 1) {  
 return FieldState.*GOOD*;  
 }  
  
 if (*PORTAL*.usedAmount == 1) {  
 return FieldState.*ONE\_PORTAL*;  
 }  
  
 return FieldState.*UNUSED\_ESSENTIALS*;  
 }  
  
 public int getUsedAmount() {  
 return usedAmount;  
 }  
  
 public static void flush() {  
 for (CellType type: *values*()) {  
 type.usedAmount = type == *EMPTY* ? MainEngine.*cellAmount*() : 0;  
 if (type == *WALL* || type == *EMPTY*) {  
 type.maxAmount = MainEngine.*cellAmount*();  
 }  
 }  
 }  
}

**Coordinate.java**

package greenbeaver.terraincognita.model.cellConstruction;  
  
import greenbeaver.terraincognita.model.MainEngine;  
import org.jetbrains.annotations.Nullable;  
  
public class Coordinate {  
  
 public enum CoordinateState {  
 *UNKNOWN*,  
 *KNOWN\_UNREACHABLE*,  
 *KNOWN\_REACHABLE*,  
 *KNOWN\_BAD\_PORTAL*,  
 *KNOWN\_PORTAL*,  
 *KNOWN\_PORTAL\_TO\_EXIT*,  
 *KNOWN\_MAZE\_BORDER* }  
  
 private static CoordinateState[][] *coordinateStates*;  
 private static CoordinateState[][] *localCoordinateStates*;  
  
 public static void clearLocalCoordinateStates() {  
 for (int i = 0; i < MainEngine.*getMazeHeight*() \* 2 + 1; i++) {  
 for (int j = 0; j < MainEngine.*getMazeWidth*() \* 2 + 1; j++) {  
 *localCoordinateStates*[i][j] = CoordinateState.*UNKNOWN*;  
 }  
 }  
 }  
  
 public static void setCoordinateStates(CoordinateState[][] newCoordinateStates) {  
 CoordinateState[][] settingNow = MainEngine.*isBlindMode*() ? *localCoordinateStates* : *coordinateStates*;  
 for (int i = 0; i < settingNow.length; i++) {  
 System.*arraycopy*(newCoordinateStates[i], 0, settingNow[i], 0, settingNow[0].length);  
 }  
 }  
  
 public static CoordinateState[][] getCoordinateStates() {  
 return MainEngine.*isBlindMode*() ? *localCoordinateStates* : *coordinateStates*;  
 }  
  
 public static void setNewField() {  
 *localCoordinateStates* =  
 new CoordinateState[2 \* MainEngine.*getMazeHeight*() + 1][2 \* MainEngine.*getMazeWidth*() + 1];  
 *clearLocalCoordinateStates*();  
  
 *coordinateStates* = new CoordinateState[MainEngine.*getMazeHeight*()][MainEngine.*getMazeWidth*()];  
 for (int i = 0; i < MainEngine.*getMazeHeight*(); i++) {  
 for (int j = 0; j < MainEngine.*getMazeWidth*(); j++) {  
 *coordinateStates*[i][j] = MainEngine.*getMaze*()[i][j].getCellType() == CellType.*ENTRANCE* ? CoordinateState.*KNOWN\_REACHABLE* : CoordinateState.*UNKNOWN*;  
 }  
 }  
 }  
  
 private final int x;  
 private final int y;  
  
 public Coordinate(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 public Coordinate copy() {  
 return new Coordinate(x, y);  
 }  
  
 public Coordinate add(int toX, int toY) {  
 return new Coordinate(x + toX, y + toY);  
 }  
  
 public Coordinate add(Direction direction) { return add(direction.getToX(), direction.getToY()); }  
  
 public Coordinate subtract(Coordinate coordinate) {  
 return new Coordinate(x - coordinate.x, y - coordinate.y);  
 }  
  
 public int getX() {  
 return x;  
 }  
  
 public int getY() {  
 return y;  
 }  
  
 public boolean fits() {  
 return x >= 0 && y >= 0 && x < MainEngine.*getMazeWidth*() && y < MainEngine.*getMazeHeight*();  
 }  
  
 public boolean fitsLocally() {  
 return MainEngine.*isBlindMode*() && x >= 0 && y >= 0 && x < MainEngine.*getMazeWidth*() \* 2 + 1 && y < MainEngine.*getMazeHeight*() \* 2 + 1;  
 }  
  
 public int getRawNumber() {  
 return (MainEngine.*isBlindMode*() ? MainEngine.*getMazeWidth*() \* 2 + 1 : MainEngine.*getMazeWidth*()) \* y + x;  
 }  
  
 public static Coordinate getByRawNumber(int rawNumber) {  
 int div = MainEngine.*isBlindMode*() ? MainEngine.*getMazeWidth*() \* 2 + 1 : MainEngine.*getMazeWidth*();  
 int y = rawNumber / div;  
 int x = rawNumber % div;  
  
 return new Coordinate(x, y);  
 }  
  
 public CoordinateState getCoordinateState() {  
 return MainEngine.*isBlindMode*() ? *localCoordinateStates*[y][x] : *coordinateStates*[y][x];  
 }  
  
 public void setCoordinateState(CoordinateState coordinateState, @Nullable Direction last) {  
 if (MainEngine.*isBlindMode*()) {  
 if (coordinateState == CoordinateState.*KNOWN\_MAZE\_BORDER*) {  
 assert last != null;  
 int multiplier = last.isPositive() ? 1 : -1;  
 if (last.getHorizontal()) {  
 for (int i = 0; i < MainEngine.*getMazeHeight*() \* 2 + 1; i++) {  
 *localCoordinateStates*[i][x] = coordinateState;  
 int otherBorder = x + multiplier \* (MainEngine.*getMazeWidth*() + 1);  
 *localCoordinateStates*[i][otherBorder] = coordinateState;  
 }  
 } else {  
 for (int i = 0; i < MainEngine.*getMazeWidth*() \* 2 + 1; i++) {  
 *localCoordinateStates*[y][i] = coordinateState;  
 int otherBorder = y + multiplier \* (MainEngine.*getMazeHeight*() + 1);  
 *localCoordinateStates*[otherBorder][i] = coordinateState;  
 }  
 }  
 } else {  
 *localCoordinateStates*[y][x] = coordinateState;  
 }  
 } else {  
 *coordinateStates*[y][x] = coordinateState;  
 }  
 }  
  
 @Override  
 public String toString() {  
 return "X: " + x + "; Y: " + y;  
 }  
  
 @Override  
 public boolean equals(Object obj) {  
 if (obj.getClass() != getClass()) {  
 return false;  
 }  
  
 Coordinate other = (Coordinate) obj;  
 return other.x == x && other.y == y;  
 }  
}

**Direction.java**

package greenbeaver.terraincognita.model.cellConstruction;  
  
public enum Direction {  
 *UP*(0, -1, false, true),  
 *RIGHT*(1, 0, true, false),  
 *DOWN*(0, 1, false, false),  
 *LEFT*(-1, 0, true, true);  
  
 private final int toX;  
 private final int toY;  
  
 private final boolean horizontal;  
 private final boolean positive;  
  
 Direction(int toX, int toY, boolean horizontal, boolean positive) {  
 this.toX = toX;  
 this.toY = toY;  
 this.horizontal = horizontal;  
 this.positive = positive;  
 }  
  
 public static Direction getByConstructor(int byX, int byY) {  
 for (Direction direction: *values*()) {  
 if (direction.toX == byX && direction.toY == byY) {  
 return direction;  
 }  
 }  
 throw new IllegalStateException("no direction for X: " + byX + " Y:" + byY);  
 }  
  
 public int getToX() {  
 return toX;  
 }  
  
 public int getToY() {  
 return toY;  
 }  
  
 public Direction opposite() {  
 switch (this) {  
 case *UP*: {  
 return *DOWN*;  
 }  
  
 case *DOWN*: {  
 return *UP*;  
 }  
  
 case *RIGHT*: {  
 return *LEFT*;  
 }  
  
 case *LEFT*: {  
 return *RIGHT*;  
 }  
  
 default: throw new IllegalStateException("Unexpected value: " + this);  
 }  
 }  
  
 public Direction firstPerpendicular() {  
 if (horizontal) {  
 return *UP*;  
 }  
 return *LEFT*;  
 }  
  
 public boolean getHorizontal() {  
 return horizontal;  
 }  
  
 public boolean isPositive() {  
 return positive;  
 }  
}

**MoveResult.java**

package greenbeaver.terraincognita.model.cellConstruction;  
  
public enum MoveResult {  
 *SUCCESSFUL*,  
 *MAZE\_BORDER*,  
 *UNREACHABLE\_CELL*,  
 *ALREADY\_VISITED\_CELL*,  
 *PORTAL*;  
  
 static Cell *result* = null;  
  
 static void setResult(Cell newResult) {  
 *result* = newResult;  
 }  
  
 public Cell getResult() {  
 return *result*;  
 }  
}

**DangerousInputAlarm.fxml**

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import javafx.geometry.\*?>  
<?import java.lang.\*?>  
<?import java.util.\*?>  
<?import javafx.scene.\*?>  
<?import javafx.scene.control.\*?>  
<?import javafx.scene.layout.\*?>  
  
<AnchorPane id="alarm\_window" prefHeight="150.0" prefWidth="400.0" styleClass="panel\_with\_centered\_image" stylesheets="@../styles/MainStylesheet.css" xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1" fx:controller="greenbeaver.terraincognita.controllers.DangerousInputAlarmController">  
 <children>  
 <VBox alignment="CENTER" layoutX="233.0" layoutY="174.0" spacing="20.0" styleClass="transparent\_pane" AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0" AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">  
 <children>  
 <Label text="Are you sure you want to continue with problematic inputs?">  
 <styleClass>  
 <String fx:value="various\_text" />  
 <String fx:value="mayan\_text" />  
 </styleClass></Label>  
 <HBox alignment="CENTER" spacing="200.0">  
 <children>  
 <Button mnemonicParsing="false" onAction="#confirm" text="Yes">  
 <styleClass>  
 <String fx:value="small\_button" />  
 <String fx:value="mayan\_text" />  
 <String fx:value="only\_text\_button" />  
 </styleClass></Button>  
 <Button mnemonicParsing="false" onAction="#refuse" text="No">  
 <styleClass>  
 <String fx:value="small\_button" />  
 <String fx:value="mayan\_text" />  
 <String fx:value="only\_text\_button" />  
 </styleClass></Button>  
 </children>  
 </HBox>  
 </children>  
 <padding>  
 <Insets top="160.0" />  
 </padding>  
 </VBox>  
 </children>  
</AnchorPane>

**HelpView.fxml**

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import javafx.scene.image.\*?>  
<?import javafx.geometry.\*?>  
<?import java.lang.\*?>  
<?import java.util.\*?>  
<?import javafx.scene.\*?>  
<?import javafx.scene.control.\*?>  
<?import javafx.scene.layout.\*?>  
  
<ScrollPane hbarPolicy="NEVER" maxHeight="-Infinity" minWidth="-Infinity" pannable="true" prefWidth="1300.0" styleClass="transparent\_pane" stylesheets="@../styles/MainStylesheet.css" vbarPolicy="NEVER" xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1">  
 <content>  
 <VBox prefWidth="1300.0" spacing="15.0" styleClass="transparent\_pane">  
 <children>  
 <Label text="This is the Terra Incognita Solver - the best App whenever you're stuck at a treasure island with no clue about how to escape.">  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 <VBox.margin>  
 <Insets />  
 </VBox.margin>  
 <padding>  
 <Insets bottom="4.0" left="4.0" right="4.0" top="4.0" />  
 </padding>  
 </Label>  
 <Label text="First of all, enter the parameters of the Island you're trying to... explore. Both Width and Height may vary between 1 and 35 inclusive, but it's strongly recommended not to get lost at Islands much bigger than 20 x 20: their explorations may cause some issues with computation time and visualisation.">  
 <padding>  
 <Insets bottom="4.0" left="4.0" right="4.0" top="4.0" />  
 </padding>  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </Label>  
 <Label text="Next step would be to fill the Map with all the stuff you're to meet at this Island. You can do this by simply left-clicking the Cells of the Map. Tiles would appear in this order:">  
 <padding>  
 <Insets bottom="4.0" left="4.0" right="4.0" top="4.0" />  
 </padding>  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </Label>  
 <Label graphicTextGap="20.0" text="Just a basic tile - the Floor. No secret traps here. Probably...">  
 <padding>  
 <Insets bottom="4.0" left="15.0" right="4.0" top="4.0" />  
 </padding>  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 <graphic>  
 <ImageView fitHeight="50.0" fitWidth="50.0" pickOnBounds="true" preserveRatio="true">  
 <image>  
 <Image url="@../images/tiles/floor.png" />  
 </image>  
 </ImageView>  
 </graphic>  
 <VBox.margin>  
 <Insets />  
 </VBox.margin>  
 </Label>  
 <Label graphicTextGap="20.0" text="Old but gold: these Walls have been built thousands of years ago by some nameless civilization, but still don't let you pass through some Cells successfully.">  
 <graphic>  
 <ImageView fitHeight="50.0" fitWidth="50.0" pickOnBounds="true" preserveRatio="true">  
 <image>  
 <Image url="@../images/tiles/wall.png" />  
 </image>  
 </ImageView>  
 </graphic>  
 <padding>  
 <Insets bottom="4.0" left="15.0" right="4.0" top="4.0" />  
 </padding>  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </Label>  
 <Label graphicTextGap="20.0" text="Seriously, don't even try imagining where these anomalies appeared from on this peace of earth. Just remember that each Portal would toss you to the one that has number that's 1 bigger than this one's. To change the numbers of Portals, right-click on them, and you'll get a dropdown menu for this. Of course, no Portals can have same numbers - pay attention to this while filling the Map!">  
 <graphic>  
 <ImageView fitHeight="50.0" fitWidth="50.0" pickOnBounds="true" preserveRatio="true">  
 <image>  
 <Image url="@../images/tiles/portal.png" />  
 </image>  
 </ImageView>  
 </graphic>  
 <padding>  
 <Insets bottom="4.0" left="15.0" right="4.0" top="4.0" />  
 </padding>  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </Label>  
 <Label graphicTextGap="20.0" text="The only reason of this whole trouble: Treasure! This one must definitely be collected before escaping!">  
 <graphic>  
 <ImageView fitHeight="50.0" fitWidth="50.0" pickOnBounds="true" preserveRatio="true">  
 <image>  
 <Image url="@../images/tiles/treasure.png" />  
 </image>  
 </ImageView>  
 </graphic>  
 <padding>  
 <Insets bottom="4.0" left="15.0" right="4.0" top="4.0" />  
 </padding>  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </Label>  
 <Label graphicTextGap="20.0" text="The spot where the REAL adventure began when your poor old airplane crashed.">  
 <graphic>  
 <ImageView fitHeight="50.0" fitWidth="50.0" pickOnBounds="true" preserveRatio="true">  
 <image>  
 <Image url="@../images/tiles/entrance.png" />  
 </image>  
 </ImageView>  
 </graphic>  
 <padding>  
 <Insets bottom="4.0" left="15.0" right="4.0" top="4.0" />  
 </padding>  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </Label>  
 <Label graphicTextGap="20.0" text="Again, you're not here to think - you're here to grab the goods and flee! And thus, no matter how it got here, this old telephone station is your only hope to complete the second part of this plan.">  
 <graphic>  
 <ImageView fitHeight="50.0" fitWidth="50.0" pickOnBounds="true" preserveRatio="true">  
 <image>  
 <Image url="@../images/tiles/exit.png" />  
 </image>  
 </ImageView>  
 </graphic>  
 <padding>  
 <Insets bottom="4.0" left="15.0" right="4.0" top="4.0" />  
 </padding>  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </Label>  
 <Label text="When all the preparations are done, hit the Solve button. In a few moments the App will provide you all the information about the exploration of the Island you've described">  
 <padding>  
 <Insets bottom="4.0" left="4.0" right="4.0" top="4.0" />  
 </padding>  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </Label>  
 <Label text="At the very right, there'll appear a list of all Cells you'll have to visit during your journey. However, some of them might only be the parts of computations, while you actually should take great care not to even approach them. So, you'll be able to know this all by simply hovering over the text representation of a Coordinate in this List; that exact Coordinate would immediately be highlighted in the Map. The color would be blue or red if the Cell should actually be a part of the Path, and yellow ot orange if not.">  
 <padding>  
 <Insets bottom="4.0" left="4.0" right="4.0" top="4.0" />  
 </padding>  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </Label>  
 </children>  
 <padding>  
 <Insets left="80.0" right="50.0" top="60.0" />  
 </padding>  
 </VBox>  
 </content>  
</ScrollPane>

**Hint.fxml**

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import java.lang.\*?>  
<?import java.util.\*?>  
<?import javafx.scene.\*?>  
<?import javafx.scene.control.\*?>  
<?import javafx.scene.layout.\*?>  
  
<AnchorPane styleClass="transparent\_pane" stylesheets="@../styles/MainStylesheet.css" xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1">  
 <children>  
 <Label fx:id="hintLabel" layoutX="46.0" AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0" AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">  
 <styleClass>  
 <String fx:value="hint" />  
 <String fx:value="various\_text" />  
 <String fx:value="mayan\_text" />  
 </styleClass>  
 </Label>  
 </children>  
</AnchorPane>

**MainScreen.fxml**

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import java.lang.\*?>  
<?import javafx.geometry.\*?>  
<?import javafx.geometry.Insets?>  
<?import javafx.scene.control.\*?>  
<?import javafx.scene.layout.\*?>  
  
<VBox id="sea\_background" fx:id="mainContainer" minHeight="400.0" minWidth="600.0" stylesheets="@../styles/MainStylesheet.css" xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1" fx:controller="greenbeaver.terraincognita.controllers.MainController">  
 <children>  
 <AnchorPane id="top\_panel" prefHeight="50.0" styleClass="panel\_with\_centered\_image">  
 <children>  
 <HBox alignment="CENTER" spacing="15.0" AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0" AnchorPane.topAnchor="0.0">  
 <children>  
 <Button id="edit\_maze\_button" mnemonicParsing="false" onAction="#loadMazeEditor" text="Edit Maze">  
 <styleClass>  
 <String fx:value="only\_text\_button" />  
 <String fx:value="mayan\_text" />  
 </styleClass></Button>  
 <Button id="help\_button" mnemonicParsing="false" onAction="#loadHelp" text="Help">  
 <styleClass>  
 <String fx:value="only\_text\_button" />  
 <String fx:value="mayan\_text" />  
 </styleClass></Button>  
 </children>  
 <opaqueInsets>  
 <Insets />  
 </opaqueInsets>  
 <padding>  
 <Insets bottom="5.0" left="7.0" />  
 </padding>  
 </HBox>  
 <HBox alignment="CENTER" layoutX="400.0" layoutY="-25.0" spacing="3.0" AnchorPane.bottomAnchor="0.0" AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">  
 <children>  
 <Button id="iconify\_button" mnemonicParsing="false" onAction="#fold" styleClass="only\_image\_button" text="Fold" />  
 <Button id="close\_button" mnemonicParsing="false" onAction="#closeApp" styleClass="only\_image\_button" text="Close" />  
 </children>  
 <padding>  
 <Insets bottom="2.0" right="7.0" />  
 </padding>  
 </HBox>  
 </children>  
 </AnchorPane>  
 </children>  
</VBox>

**MazeEditor.fxml**

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import javafx.scene.text.\*?>  
<?import javafx.geometry.\*?>  
<?import javafx.scene.image.\*?>  
<?import javafx.scene.\*?>  
<?import java.lang.\*?>  
<?import javafx.scene.control.\*?>  
<?import javafx.scene.layout.\*?>  
  
<AnchorPane minHeight="-Infinity" minWidth="-Infinity" prefHeight="750.0" prefWidth="1300.0" styleClass="transparent\_pane" stylesheets="@../styles/MainStylesheet.css" xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1" fx:controller="greenbeaver.terraincognita.controllers.MazeEditorController">  
 <children>  
 <VBox id="left\_panel" fillWidth="false" maxHeight="-Infinity" maxWidth="-Infinity" spacing="50.0" styleClass="panel\_with\_centered\_image" AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0" AnchorPane.topAnchor="0.0">  
 <children>  
 <HBox alignment="CENTER" spacing="4.0" styleClass="left\_panel\_tile\_h">  
 <children>  
 <ImageView fx:id="heightScull" fitHeight="52.0" fitWidth="52.0" pickOnBounds="true" preserveRatio="true" styleClass="scull">  
 <image>  
 <Image url="@../images/normalScull.png" />  
 </image>  
 <viewport>  
 <Rectangle2D height="52.0" width="52.0" />  
 </viewport>  
 </ImageView>  
 <VBox spacing="3.0" styleClass="left\_panel\_tile\_v">  
 <children>  
 <Label graphicTextGap="0.0" prefWidth="62.0" text="Maze Height">  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="input" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </Label>  
 <TextField fx:id="mazeHeightInput" maxWidth="-Infinity" minWidth="-Infinity" onKeyTyped="#checkHeightInput" prefWidth="62.0">  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="input" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </TextField>  
 </children>  
 </VBox>  
 </children>  
 </HBox>  
 <HBox layoutX="10.0" layoutY="10.0" spacing="4.0" styleClass="left\_panel\_tile\_h">  
 <children>  
 <ImageView fx:id="widthScull" fitHeight="52.0" fitWidth="52.0" pickOnBounds="true" preserveRatio="true" styleClass="scull">  
 <viewport>  
 <Rectangle2D height="52.0" width="52.0" />  
 </viewport>  
 </ImageView>  
 <VBox styleClass="left\_panel\_tile\_v">  
 <children>  
 <Label text="Maze Width">  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="input" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </Label>  
 <TextField fx:id="mazeWidthInput" maxWidth="-Infinity" minWidth="-Infinity" onKeyTyped="#checkWidthInput" prefWidth="40.0">  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="input" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </TextField>  
 </children>  
 </VBox>  
 </children>  
 </HBox>  
 <HBox layoutX="10.0" layoutY="160.0" spacing="4.0" styleClass="left\_panel\_tile\_h">  
 <children>  
 <ImageView fx:id="submissionScull" fitHeight="52.0" fitWidth="52.0" pickOnBounds="true" preserveRatio="true" styleClass="scull" />  
 <VBox styleClass="left\_panel\_tile\_v">  
 <children>  
 <Button mnemonicParsing="false" onAction="#saveProperties" prefWidth="45.0" text="OK">  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="only\_text\_button" />  
 <String fx:value="small\_button" />  
 </styleClass></Button>  
 <Button id="solve\_button" fx:id="solveButton" mnemonicParsing="false" onAction="#solve" text="Solve!" visible="false">  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="only\_text\_button" />  
 </styleClass></Button>  
 </children>  
 </VBox>  
 </children>  
 </HBox>  
 </children>  
 <padding>  
 <Insets left="4.0" top="80.0" />  
 </padding>  
 </VBox>  
 <ScrollPane hbarPolicy="NEVER" pannable="true" styleClass="transparent\_pane" vbarPolicy="NEVER" AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="230.0" AnchorPane.rightAnchor="260.0" AnchorPane.topAnchor="0.0">  
 <content>  
 <AnchorPane fx:id="mazeContainer" styleClass="transparent\_pane">  
 <padding>  
 <Insets left="50.0" top="80.0" />  
 </padding>  
 </AnchorPane>  
 </content>  
 </ScrollPane>  
 <ScrollPane fitToHeight="true" fitToWidth="true" maxWidth="260.0" minWidth="260.0" prefWidth="260.0" styleClass="transparent\_pane" AnchorPane.bottomAnchor="0.0" AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">  
 <content>  
 <VBox fx:id="resultView" alignment="TOP\_CENTER" maxWidth="260.0" minWidth="260.0" prefWidth="260.0" spacing="10.0" styleClass="transparent\_pane">  
 <children>  
 <Label text="Results">  
 <styleClass>  
 <String fx:value="result\_text" />  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 </styleClass>  
 </Label>  
 <Label fx:id="treasureState" text="Treasure Found: ">  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 <String fx:value="result\_text" />  
 </styleClass>  
 </Label>  
 <Label fx:id="exitState" text="Exit Reached: ">  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 <String fx:value="result\_text" />  
 </styleClass>  
 </Label>  
 <Label fx:id="cCellsPassed" text="Computational Cells Passed: ">  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 <String fx:value="result\_text" />  
 </styleClass>  
 </Label>  
 <Label fx:id="rCellsPassed" text="Real Cells Passed: ">  
 <styleClass>  
 <String fx:value="mayan\_text" />  
 <String fx:value="various\_text" />  
 <String fx:value="result\_text" />  
 </styleClass>  
 </Label>  
 </children>  
 <padding>  
 <Insets top="20.0" />  
 </padding>  
 </VBox>  
 </content>  
 </ScrollPane>  
 <ImageView fx:id="filler" fitHeight="100.0" fitWidth="230.0" pickOnBounds="true" preserveRatio="true" visible="false" AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0" AnchorPane.rightAnchor="0.0">  
 <image>  
 <Image url="@../images/backgrounds/bottomFiller.png" />  
 </image>  
 <viewport>  
 <Rectangle2D />  
 </viewport>  
 </ImageView>  
 </children>  
</AnchorPane>

**PortalSettings.fxml**

<?xml version="1.0" encoding="UTF-8"?>  
  
<?import java.lang.\*?>  
<?import java.util.\*?>  
<?import javafx.scene.\*?>  
<?import javafx.scene.control.\*?>  
<?import javafx.scene.layout.\*?>  
  
<AnchorPane prefHeight="100.0" style="-fx-background-color: #192522; -fx-border-color: #2b5045; -fx-border-width: 4;" stylesheets="@../styles/MainStylesheet.css" xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1" fx:controller="greenbeaver.terraincognita.controllers.PortalSettingsController">  
 <children>  
 <ChoiceBox fx:id="variants" prefWidth="50.0" AnchorPane.leftAnchor="5.0" AnchorPane.rightAnchor="5.0" AnchorPane.topAnchor="5.0">  
 <styleClass>  
 <String fx:value="various\_text" />  
 <String fx:value="mayan\_text" />  
 </styleClass></ChoiceBox>  
 <Button layoutY="54.0" mnemonicParsing="false" onAction="#submit" text="OK" AnchorPane.bottomAnchor="5.0" AnchorPane.rightAnchor="5.0">  
 <styleClass>  
 <String fx:value="only\_text\_button" />  
 <String fx:value="small\_button" />  
 <String fx:value="mayan\_text" />  
 </styleClass></Button>  
 </children>  
</AnchorPane>

# **ОШИБКИ И ПРЕДУПРЕЖДЕНИЯ**

|  |  |
| --- | --- |
| Код ошибки | Описание ошибки |
| 1 | В поля ввода параметров лабиринта введены знаки, не являющиеся цифрами |
| 2 | В поля ввода параметров лабиринта введены нулевые значения |
| 3 | В поля ввода параметров лабиринта введены значения >35 |
| 4 | В поля ввода параметров лабиринта ничего не введено |
| 5 | На поле отсутствуют Вход, Выход или Сокровище |
| 6 | На поле один портал |
| 7 | Не у всех порталов разные номера |
|  |  |
| Код предупреждения | Описание предупреждения |
| 1 | В поля ввода параметров лабиринта введены значения больше 20 |

# 

# **ТЕСТЫ**

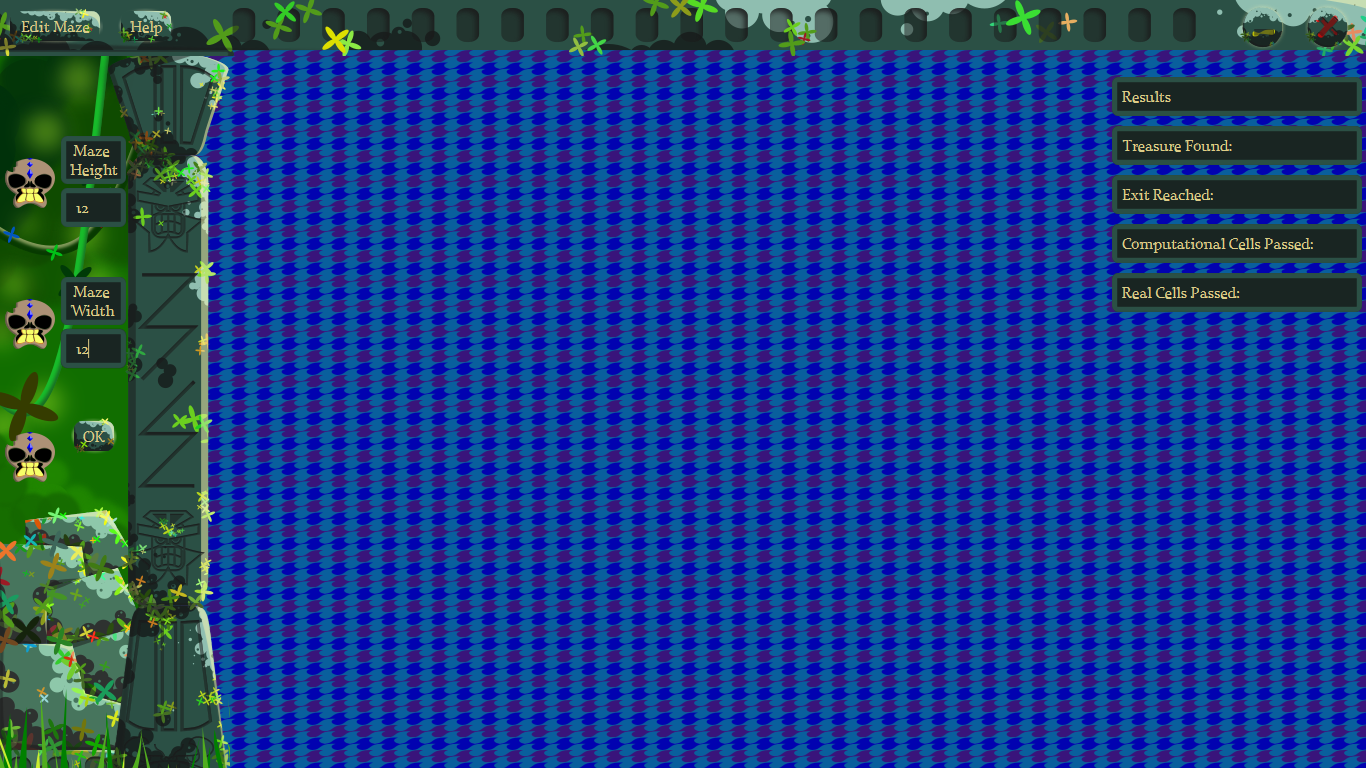
Все функции из класса Tester тестируют методы перечисления Direction и класса Coordinate

**Tester.java**

package greenbeaver.terraincognita;  
  
import greenbeaver.terraincognita.model.cellConstruction.Coordinate;  
import greenbeaver.terraincognita.model.cellConstruction.Direction;  
import org.junit.Test;  
  
import static org.junit.Assert.*assertEquals*;  
import static org.junit.Assert.*assertTrue*;  
  
public class Tester {  
  
 // checks that Direction returns proper perpendicular Direction  
 @Test  
 public void perpendicular() {  
 Direction left = Direction.*LEFT*;  
 *assertEquals*(left.firstPerpendicular(), Direction.*UP*);  
 Direction down = Direction.*DOWN*;  
 *assertEquals*(Direction.*LEFT*, down.firstPerpendicular());  
 }  
  
 // checks that Direction returns proper opposite Direction  
 @Test  
 public void opposite() {  
 Direction up = Direction.*UP*;  
 *assertEquals*(Direction.*DOWN*, up.opposite());  
 Direction right = Direction.*RIGHT*;  
 *assertEquals*(Direction.*LEFT*, right.opposite());  
 }  
  
 // checks that Direction can be received correctly by its toX and toY parameters  
 @Test  
 public void byConstructor() {  
 boolean exceptionHappened = false;  
 try {  
 Direction d = Direction.*getByConstructor*(6, 7);  
 } catch (IllegalStateException e) {  
 exceptionHappened = true;  
 }  
 *assertTrue*(exceptionHappened);  
  
 *assertEquals*(Direction.*LEFT*, Direction.*getByConstructor*(-1, 0));  
 }  
  
 // checks that Directions add proper values to X and Y coordinates  
 @Test  
 public void xy() {  
 *assertEquals*(1, Direction.*DOWN*.getToY());  
 *assertEquals*(0, Direction.*UP*.getToX());  
 }  
  
 // checks that Coordinate is properly converted to String  
 @Test  
 public void coordinateString() {  
 *assertEquals*("X: 45; Y: 11", new Coordinate(45, 11).toString());  
 }  
}

# 

**СКРИНШОТЫ ПРОГРАММЫ**



Рис

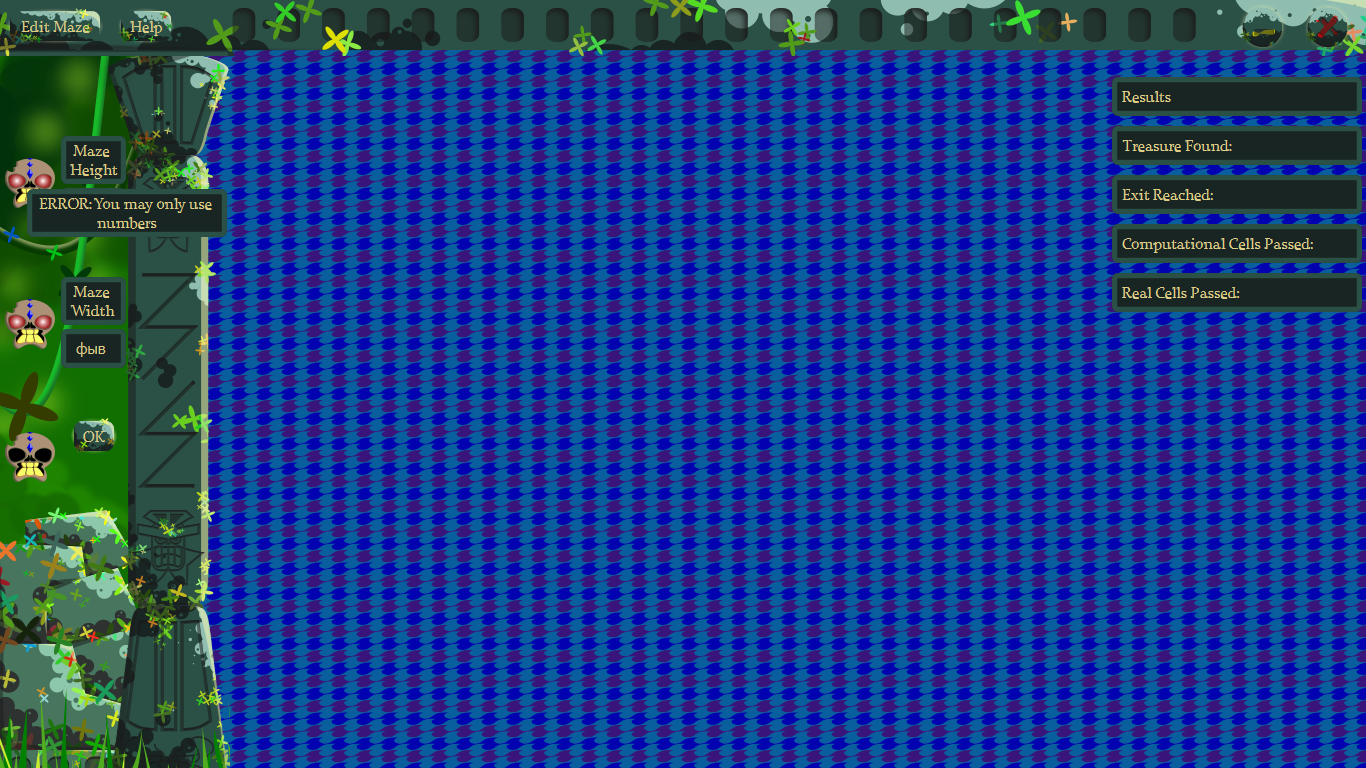


Рис. 1. Введены не численные значения

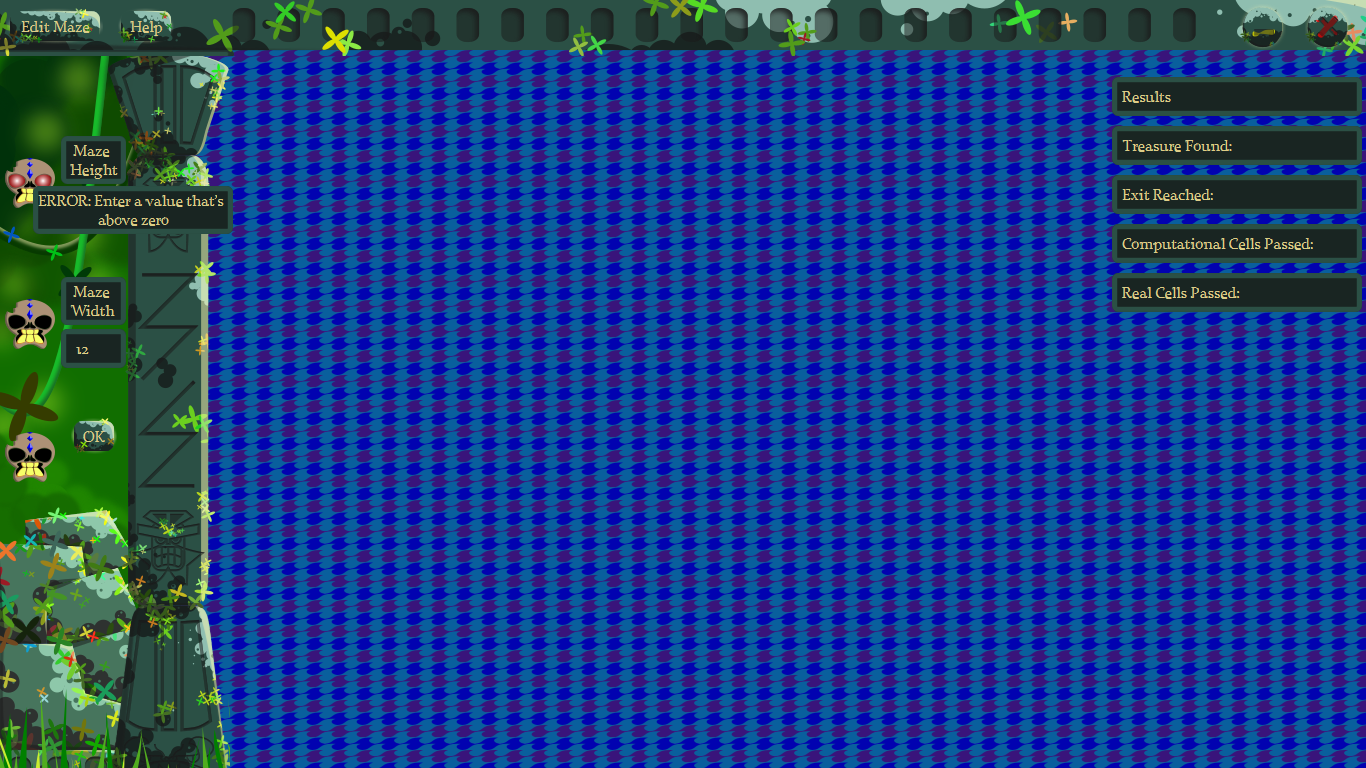


Рис. 2. Введено нулевое значение



Рис. 3. Введено значение больше 20

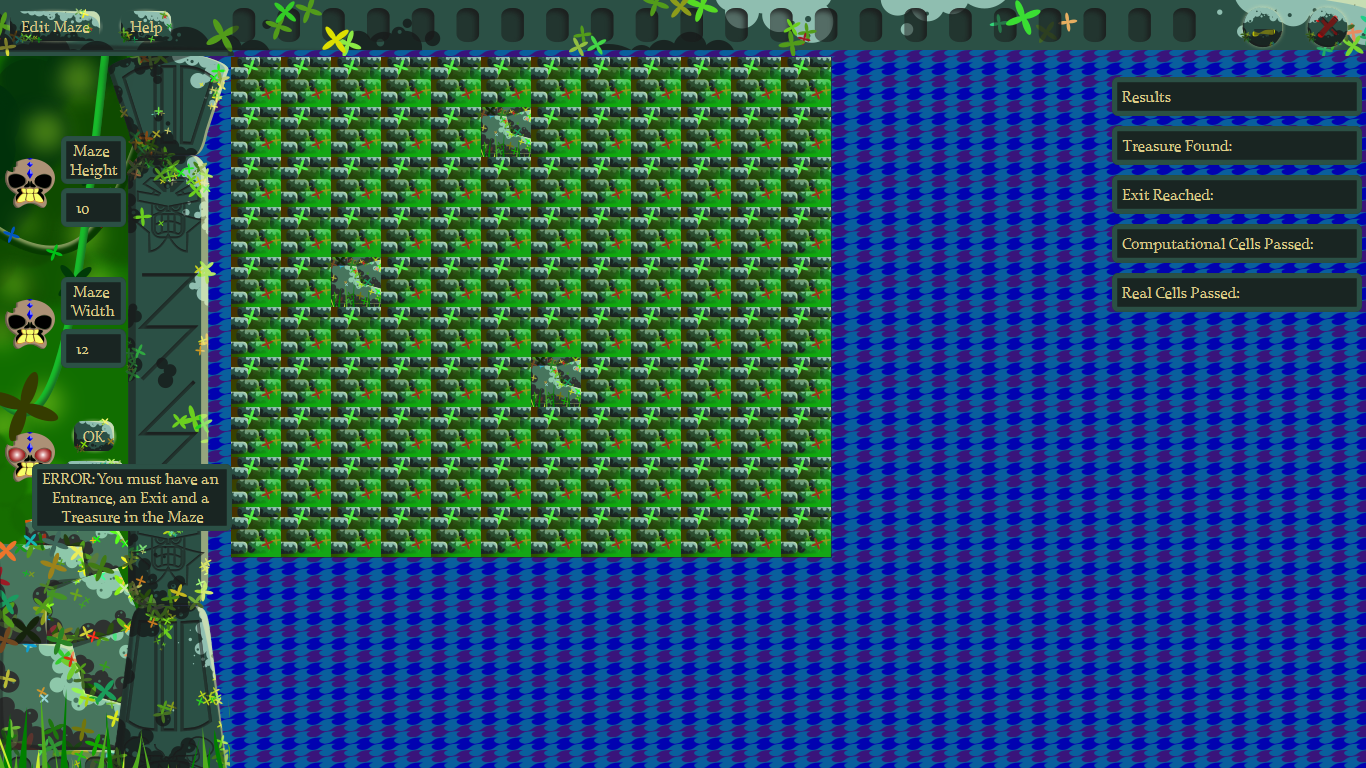


Рис. 4. На поле не хватает Входа, Выхода и Сокровища

