Advanced Software Engineering

Anhand der Applikation “AlgorithmVisualizer”

Stand:	17.05.2021
Autor:	Kai Müller
Kurs:	TINF18B5

Inhaltsverzeichnis

1. Einleitung

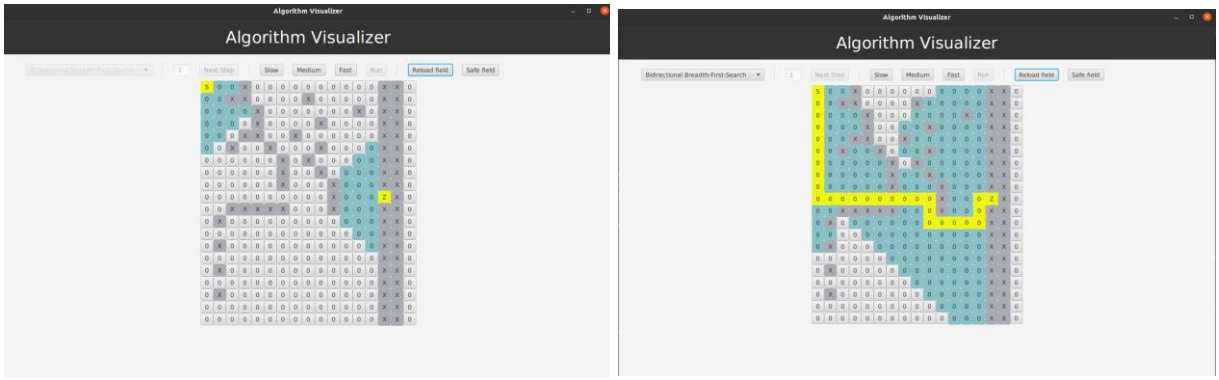
Die Applikation „AlgorithmVisualizer“ kann den Suchvorgang von diversen Algorithmen visualisieren. Diese reichen von klassischer Breadth-First-Search, über spezielle wie Bidirectional-Breadth-First-Search bis hin zu Suchalgorithmen mit heuristischem Ansatz, wie Greedy-First-Search.

Der User kann während der Suche noch das Suchfeld bearbeiten. Das Suchfeld kann gespeichert werden und beim erneuten Start der Applikation wieder geladen werden.

Der Suchalgorithmus kann durch den User Schritt-für-Schritt durchgeführt werden oder automatisch. Hierbei kann die Geschwindigkeit angepasst werden. Sobald der Algorithmus einen Pfad vom Start zum Ziel gefunden hat, wird dieser Visualisiert. Hierbei gilt zu beachten, dass nicht alle Definitionen den kürzesten Pfad zurück liefern. Es ist jedoch gut zu sehen, wie viele Felder ein Algorithmus besucht, bis er das Ziel erreicht. In der Applikation können Wände hinzugefügt oder gelöscht werden. Das Start- und Zielfeld muss über die JSON-Datei angegeben werden.

Innerhalb der JSON-Datei repräsentieren folgende Werte diese Felder:

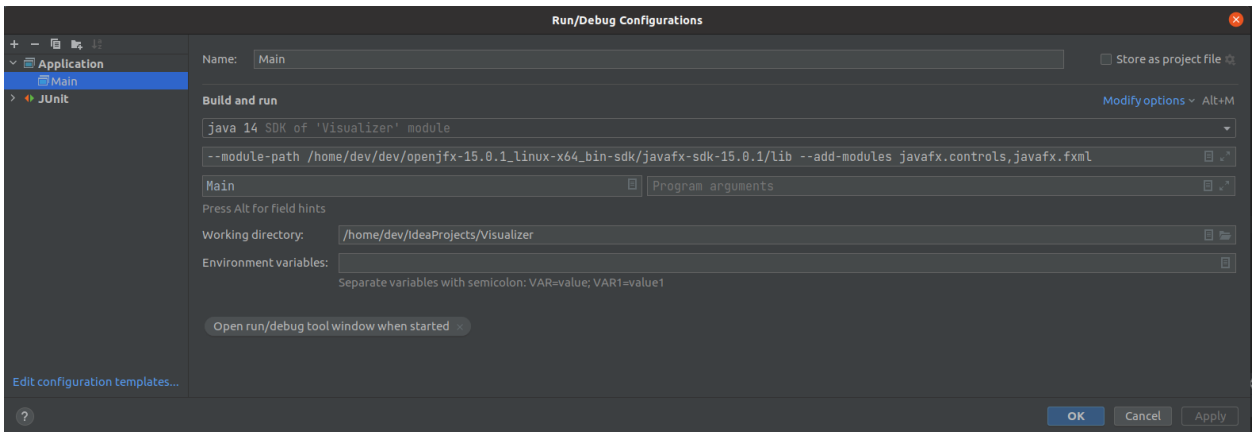
- S -> Startfeld
- Z -> Zielfeld
- 0 -> Besuchbares Feld
- X -> Wand



Innerhalb des Dokuments wurden Klassennamen und Methodennamen zur Verdeutlichung in kursiver Schrift geschrieben.

2. How to make it run

- Java 14 oder 16 installieren
    - 14 auf Linux getestet, 16 auf Windows getestet
- Pull https://github.com/LaaKii/AlgorithmVisualizer/tree/master
    - masterpath
- In IntelliJ öffnen
- Start Application (Main)
- (SOLLTE AUCH OHNE DIE FOLGENEN SCHRITTEN FUNKTIONIEREN)
    - Wenn nicht startet, muss evtl. noch:
        - JavaFx sdk installiert werden (javafx-sdk-15.0.1)
        - Folgendes in Run Configuration eintragen:
            - --module-path /home/dev/dev/openjfx-15.0.1_linux-x64_bin-sdk/javafx-sdk-15.0.1/lib --add-modules javafx.controls,javafx.fxml



3. Domain Driven Design

3.1. Analyse der Ubiquitous Language

Nachstehend werden Beispiele, die sowohl in der Domäne als auch im Code verwendet werden, aufgezeigt und beschrieben.

- SearchField
    - Suchfeld, welches von der Applikation dargestellt wird und vom User bearbeitet werden kann.
- Wall
    - Wand im Suchfeld. Kann vom User gesetzt oder entfernt werden.
- Index
    - Feld im Suchfeld
- Button
    - Knopf der gedrückt werden kann
- Direction
    - Richtung in die „etwas“ geht. In diesem Fall der Suchalgorithmus.

Die genannten Felder können also immer innerhalb einer Diskussion zwischen dem Entwickler und dem User benutzt werden. Innerhalb der Domäne sind diese Begriffe eindeutig und jeder weiß um was es geht.

3.2. Analyse und Begründung

3.2.1. Repositories

- Alle Implementierungen des Interfaces *FileProcessor*
    - Kapseln die Logik für das Speichern des Suchfeldes.
    - Kapseln die Logik für das Erzeugen des Suchfeldes.

Die einzelnen Implementierungen behandeln das komplette Filehandling von Lesen bis Schreiben.

3.2.2. Entities

Als Entitiy kann zum Beispiel die Klasse *Index* betrachtet werden. Ein Index hat veränderliche Eigenschaften und einen eigenen Lebenszyklus. Ein Index kann verschiedene Felder repräsentieren. Als eindeutige Identität können die Koordinaten

verwendet werden. Jeden Index gibt es innerhalb des Lebenszyklus der Applikation nur einmal, wenn das Feld über eine Datei eingelesen wird.
Weitere Klassen, die eine Entity darstellen sind:

- Button
- Alle FieldType Implementierungen
    - Wall
    - StartField
    - EmptyField

3.2.3. Value Object

Die Klasse *Field* wurde in ein Value Object transformiert. Hierfür wurde zunächst die Klasse sowie alle Klassenvariablen auf final gesetzt. Dies kann unter der Commit-ID „ac2d90ae92b11ac6b00e63cec358186521902bf2“ genauer betrachtet werden. Die weiteren Regeln, welche für ein Value Objects gelten müssen, wurden unter der Commit-ID „5659a1fb9db9f51595687a4b25916fffa56f27fa“ gepusht.
Field eignet sich als Value Object, da es unveränderlich innerhalb der Domäne ist, und eine Sache (nämlich ein Feld) genauer beschreibt. (2-Dimensional, 3-Dimensional, …). Es kann auch keine ungültige Version eines Value Objects erzeugt werden, da der implizite Konstruktor nicht verwendbar ist (wurde auf private gesetzt), sondern nur die zwei modifizierten.

3.2.4. Aggregates

Aggregates stellen Zusammenfassungen von Entities und Value Objects dar.
Beispiele für ein Aggregate in der Applikation ist *SearchField.* Das *SearchField b*einhaltet zwei Index -> StartIndex und EndIndex. Index ist wie bereits berschrieben eine Entity.

4. SOLID

4.1. Single Responsibility Principle

4.1.1. Analyse / Begründung / Anpassungen

Laut SRP darf jede Klasse/Methode/Objekt/Variable genau eine Aufgabe erfüllen.

Die erste Klasse welches dieses Prinzip verletzt ist *BreadthFirstSearch.* Die *doSearch* Methode erfüllt mehr als eine Aufgabe ohne entsprechende Methoden aufzurufen. Die nachstehende Abbildung zeigt dieses nicht konforme Verhalten innerhalb der Methode auf.

```
33        for (Index index : currentIndex) {
34            searchFinished = searchIndexInEveryDirection(index);
35            if (searchFinished) {
36                System.out.println("Target found at: [" + index.getRow() + "][" + index.getColumn() + "]");
37                ResultDisplayer.displayResult(buttons, index, searchField);
38                return true;
39            }
40        }
41        if (firstSearch) {
42            buttons[startField.getRow()][startField.getColumn()].setVisited(true);
43            firstSearch = false;
44        }
45
46        currentIndex = List.copyOf(indexToContinueSearch);
47        if (!isSearchForGreedyFirstSearch){
48            if (currentIndex.size() == 0) {
49                Alert alert = new Alert(Alert.AlertType.INFORMATION);
50                alert.setTitle("Information");
51                alert.setHeaderText("Target could not be found");
52                String s = "Either there is no target or the target couldn't be reached";
53                alert.setContentText(s);
54                alert.show();
55                return true;
56            }
57        }
58        indexToContinueSearch.clear();
59        return false;
60    }
```

Innerhalb der *doSearch* Methode wird ebenso ein Alert erzeugt. Dieses Verhalten ist in nachstehenden Screenshot verbessert worden.

```
46
47          currentIndex = List.copyOf(indexToContinueSearch);
48          if (!isSearchForGreedyFirstSearch){
49              if (currentIndex.size() == 0) {
50                  showSearchFinishedAlert();
51                  return true;
52              }
53          }
54          indexToContinueSearch.clear();
55          return false;
56      }
57
58      private void showSearchFinishedAlert() {
59          AlertManager alertManager = new AlertManager.Builder()
60                  .setTitle("Information")
61                  .setHeaderText("Target could not be found")
62                  .setContextText("Either there is no target or the target couldn't be reached")
63                  .setAlertType(Alert.AlertType.INFORMATION)
64                  .build();
65          alertManager.showAlert();
66      }
```

Es ist zu erkennen, dass eine neue Klasse *AlertManager* angelegt worden ist, welche das Builder Pattern benutzt. Auf das Builder-Pattern wird im Kapitel der Entwurfsmuster genauer eingegangen. Der Aufruf des Alerts wurde in die Methode *showSearchFinishedAlert()* ausgelagert. Der Commit ist die ID „1491b915704c4a6f64eece6a880051b7e9e51b01“.

Die folgenden Methoden der Klassen unterliegen dem SRP. Beispiel:

```
public boolean searchIndexInEveryDirection(Index index) {
    if (checkBelow(index) || checkAbove(index) || checkLeft(index) || checkRight(index)) {
        return true;
    } else {
        return false;
    }
}

private boolean checkLeft(Index index) {
    if (index.getColumn() - 1 >= 0  && !buttons[index.getRow()][index.getColumn() - 1].isStartField() && !buttons[index.getRow()][index.getColumn() - 1].isVisited()) {
        Index leftIndex = Index.copy(index);
        leftIndex.setPreviousIndex(index);
        leftIndex.setColumn(leftIndex.getColumn() - 1);
        return checkEndPosition(leftIndex);
    }
    return false;
}
```

Lediglich die Methode *checkEndPosition(…)* unterliegt nicht diesem Prinzip und muss angepasst werden. Die Methode ist folgender Abbildung zu sehen.

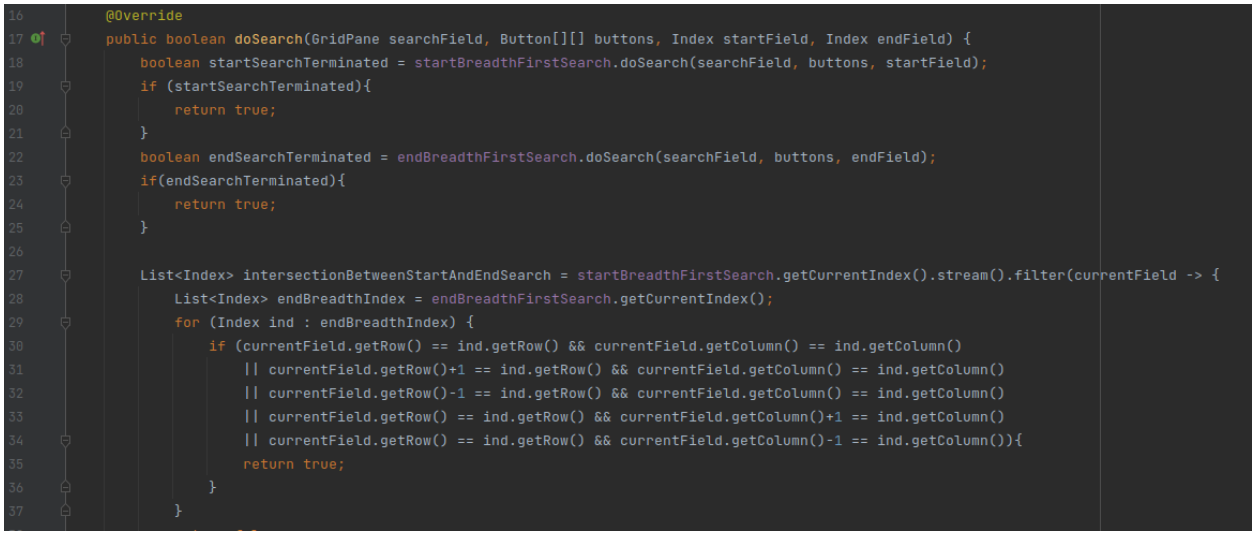
```
117 @     private boolean checkEndPosition(Index index) {
118           int row = index.getRow();
119           int column = index.getColumn();
120           buttons[index.getRow()][index.getColumn()].setVisited(true);
121           Button visitedButton = buttons[row][column];
122           if (visitedButton.getText().equals("Z")) {
123               return true;
124           } else if (visitedButton.getText().equals("X")) {
125               return false;
126           } else {
127               searchField.getChildren().remove(visitedButton);
128               visitedButton.setStyle("-fx-background-color: #89c1c7 ");
129               searchField.add(visitedButton, column, row);
130               indexToContinueSearch.add(index);
131               return false;
132           }
133       }
```

Der untere else-Fall verletzt das Prinzip. Um dies zu lösen, wurden die Anweisungen in eine Methode ausgelagert. Das Ergebnis ist in nachstehender Abbildung betrachtbar.

```
117 @   private boolean checkEndPosition(Index index) {
118         int row = index.getRow();
119         int column = index.getColumn();
120         buttons[index.getRow()][index.getColumn()].setVisited(true);
121         Button visitedButton = buttons[row][column];
122         if (visitedButton.getText().equals("Z")) {
123             return true;
124         } else if (visitedButton.getText().equals("X")) {
125             return false;
126         } else {
127             markFieldAsVisited(index, row, column, visitedButton);
128             return false;
129         }
130     }
131
132     private void markFieldAsVisited(Index index, int row, int column, Button visitedButton) {
133         searchField.getChildren().remove(visitedButton);
134         visitedButton.setStyle("-fx-background-color: #89c1c7 ");
135         searchField.add(visitedButton, column, row);
136         indexToContinueSearch.add(index);
137     }
```

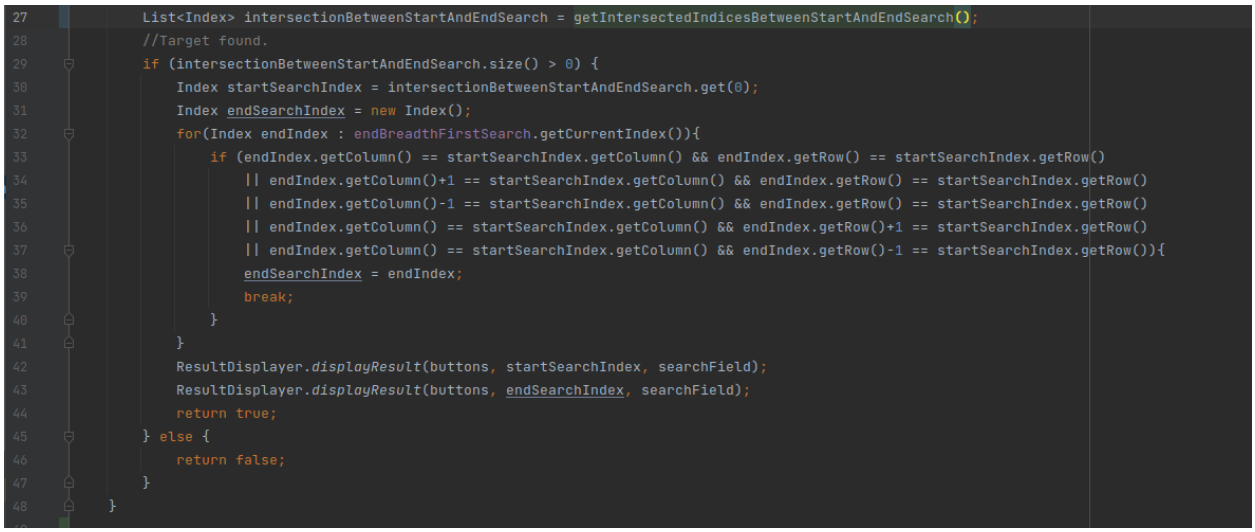
Die commit-ID für diese Anpassung lautet: „f33d99ece05a37cf2e14999a06db7e653309c06f“.

Als nächstes wird die Klasse *BidirectionalBreadthFirstSearch* betrachtet. Hierbei verletzt direkt die erste Methode *onSearch* das SRP. Nachstehend ist ein Ausschnitt der Methode zu betrachten.



```
16      @Override
17      public boolean doSearch(GridPane searchField, Button[][] buttons, Index startField, Index endField) {
18          boolean startSearchTerminated = startBreadthFirstSearch.doSearch(searchField, buttons, startField);
19          if (startSearchTerminated){
20              return true;
21          }
22          boolean endSearchTerminated = endBreadthFirstSearch.doSearch(searchField, buttons, endField);
23          if(endSearchTerminated){
24              return true;
25          }
26
27          List<Index> intersectionBetweenStartAndEndSearch = startBreadthFirstSearch.getCurrentIndex().stream().filter(currentField -> {
28              List<Index> endBreadthIndex = endBreadthFirstSearch.getCurrentIndex();
29              for (Index ind : endBreadthIndex) {
30                  if (currentField.getRow() == ind.getRow() && currentField.getColumn() == ind.getColumn()
31                      || currentField.getRow()+1 == ind.getRow() && currentField.getColumn() == ind.getColumn()
32                      || currentField.getRow()-1 == ind.getRow() && currentField.getColumn() == ind.getColumn()
33                      || currentField.getRow() == ind.getRow() && currentField.getColumn()+1 == ind.getColumn()
34                      || currentField.getRow() == ind.getRow() && currentField.getColumn()-1 == ind.getColumn()){
35                      return true;
36                  }
37              }
```

Es wird deutlich, dass Zeile 27 bis 38 nicht direkt etwas mit der Suche zu tun haben. Deshalb wird dieses Verhalten in eine eigene Methode ausgelagert.



```
27          List<Index> intersectionBetweenStartAndEndSearch = getIntersectedIndicesBetweenStartAndEndSearch();
28          //Target found.
29          if (intersectionBetweenStartAndEndSearch.size() > 0) {
30              Index startSearchIndex = intersectionBetweenStartAndEndSearch.get(0);
31              Index endSearchIndex = new Index();
32              for(Index endIndex : endBreadthFirstSearch.getCurrentIndex()){
33                  if (endIndex.getColumn() == startSearchIndex.getColumn() && endIndex.getRow() == startSearchIndex.getRow()
34                      || endIndex.getColumn()+1 == startSearchIndex.getColumn() && endIndex.getRow() == startSearchIndex.getRow()
35                      || endIndex.getColumn()-1 == startSearchIndex.getColumn() && endIndex.getRow() == startSearchIndex.getRow()
36                      || endIndex.getColumn() == startSearchIndex.getColumn() && endIndex.getRow()+1 == startSearchIndex.getRow()
37                      || endIndex.getColumn() == startSearchIndex.getColumn() && endIndex.getRow()-1 == startSearchIndex.getRow()){
38                      endSearchIndex = endIndex;
39                      break;
40                  }
41              }
42              ResultDisplayer.displayResult(buttons, startSearchIndex, searchField);
43              ResultDisplayer.displayResult(buttons, endSearchIndex, searchField);
44              return true;
45          } else {
46              return false;
47          }
48      }
```

Des Weiteren wurde der untere Teil der Methode ebenso ausgelagert. Dies ist in folgender Abbildung erkennbar.

```java
public boolean doSearch(GridPane searchField, Button[][] buttons, Index startField, Index endField) {
    boolean startSearchTerminated = startBreadthFirstSearch.doSearch(searchField, buttons, startField);
    if (startSearchTerminated){
        return true;
    }
    boolean endSearchTerminated = endBreadthFirstSearch.doSearch(searchField, buttons, endField);
    if(endSearchTerminated){
        return true;
    }

    List<Index> intersectionBetweenStartAndEndSearch = getIntersectedIndicesBetweenStartAndEndSearch();
    return checkIfSearchIsFinished(searchField, buttons, intersectionBetweenStartAndEndSearch);
}

private boolean checkIfSearchIsFinished(GridPane searchField, Button[][] buttons, List<Index> intersectionBetweenStartAndEndSearch) {
    //Target found.
    if (intersectionBetweenStartAndEndSearch.size() > 0) {
        Index startSearchIndex = intersectionBetweenStartAndEndSearch.get(0);
        Index endSearchIndex = new Index();
        for(Index endIndex : endBreadthFirstSearch.getCurrentIndex()){
            if (endIndex.getColumn() == startSearchIndex.getColumn() && endIndex.getRow() == startSearchIndex.getRow()
                || endIndex.getColumn()+1 == startSearchIndex.getColumn() && endIndex.getRow() == startSearchIndex.getRow()
                || endIndex.getColumn()-1 == startSearchIndex.getColumn() && endIndex.getRow() == startSearchIndex.getRow()
                || endIndex.getColumn() == startSearchIndex.getColumn() && endIndex.getRow()+1 == startSearchIndex.getRow()
                || endIndex.getColumn() == startSearchIndex.getColumn() && endIndex.getRow()-1 == startSearchIndex.getRow()){
                endSearchIndex = endIndex;
                break;
            }
        }
        ResultDisplayer.displayResult(buttons, startSearchIndex, searchField);
        ResultDisplayer.displayResult(buttons, endSearchIndex, searchField);
        return true;
    } else {
        return false;
    }
```

Die Commit-ID für die Anpassungen ist: „f678ceb2acf3276c8296fee1a51054d70ab63b98“. Die *doSearch()* Methode sieht nun wie folgt aus:

```java
@Override
public boolean doSearch(GridPane searchField, Button[][] buttons, Index startField, Index endField) {
    boolean startSearchTerminated = startBreadthFirstSearch.doSearch(searchField, buttons, startField);
    if (startSearchTerminated){
        return true;
    }
    boolean endSearchTerminated = endBreadthFirstSearch.doSearch(searchField, buttons, endField);
    if(endSearchTerminated){
        return true;
    }

    return checkIfSearchIsFinished(searchField, buttons, getIntersectedIndicesBetweenStartAndEndSearch());
}
```

Die fertige Anpassung sind unter der Commit-ID „24654b88717e69189174936f97b1eac8f3ba1a89“ einsehbar. Somit folgt die komplette Klasse dem SRP.

Die nächste Klasse, welche betrachtet wird, ist *DepthFirstSearch*. Hier wird zunächst die *doSearch* Methode analysiert. Es wird deutlich, dass das SRP verletzt wurde.

```
@Override
public boolean doSearch(GridPane searchField, Button[][] buttons, Index startField) {
    this.searchField = searchField;
    this.buttons = buttons;
    if (firstRun) {
        currentField = startField;
        firstRun = false;
    }

    if (searchIndexInDepth(currentField)) {
        System.out.println("Target found at: [" + currentField.getRow() + "][" + currentField.getColumn() + "]");
        ResultDisplayer.displayResult(buttons, currentField, searchField);
        return true;
    }else if (targetCannotBeReached){
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Information");
        alert.setHeaderText("Target could not be found");
        String s = "Either there is no target or the target couldn't be reached";
        alert.setContentText(s);
        alert.show();
        return true;
    }
    return false;
}
```

Zunächst, wird wie bei *BreadthFirstSearch* der Alert ausgelagert. Dies ist in folgender Abbildung sowie unter der Commit-ID „f2535b4906487733d4c3d3796419b871143c2b8c“ erkennbar.

```
@Override
public boolean doSearch(GridPane searchField, Button[][] buttons, Index startField) {
    this.searchField = searchField;
    this.buttons = buttons;
    if (firstRun) {
        currentField = startField;
        firstRun = false;
    }

    if (searchIndexInDepth(currentField)) {
        System.out.println("Target found at: [" + currentField.getRow() + "][" + currentField.getColumn() + "]");
        ResultDisplayer.displayResult(buttons, currentField, searchField);
        return true;
    }else if (targetCannotBeReached){
        showSearchFinishedDialog();
        return true;
    }
    return false;
}

private void showSearchFinishedDialog() {
    Alert alert = new Alert(Alert.AlertType.INFORMATION);
    alert.setTitle("Information");
    alert.setHeaderText("Target could not be found");
    String s = "Either there is no target or the target couldn't be reached";
    alert.setContentText(s);
    alert.show();
}
```

Alle weiteren Methoden, außer die Methode der *checkEndPosition*() entsprechen der SRP. Diese Methode wird nun angepasst. Es ist das gleiche Vorgehen wie bei der

checkEndPosition() bei der *BreadthFirstSearch* Klasse. Das Ergebnis ist unter der Commit-ID „52f14ad1ec07e3ae1f0629fbc3ab832cc105c0d6“ ersichtlich.

Als nächste Klasse wird *GreedyFirstSearch* betrachtet. Hier muss (wie in den anderen Klassen auch) die Methode *onSearch()* angepasst werden. Diese Methode ist zu groß zum Screenshotten. Es empfiehlt sich hierfür also die Commits zu verfolgen. Der erste Schritt war es die einzelnen Suchschritte der verfügbaren Indizes auszulagern. Hierfür wurde eine neue Methode angelegt. Dies kann in Commit-ID „52f14ad1ec07e3ae1f0629fbc3ab832cc105c0d6“ nachverfolgt werden. Als nächstes muss die neue Methode *canAnyIndexReachTheGoal()* ebenso dem SRP folgen. Dies wurde durch verschiedene Auslagerungen erreicht. Die Methode ist nun um einiges lesbarer und verständlicher. Die Änderungen sind unter der Commit-ID „59ff1f838bae86c3ebed7dd39d4041d475a2d6ba“ ersichtlich. Wie in den anderen Klassen ist wieder in der *checkEndPosition()* Methode das SRP verletzt.

```
@    private boolean checkEndPosition(Index index) {
         int row = index.getRow();
         int column = index.getColumn();
         buttons[row][column].setVisited(true);
         Button visitedButton = buttons[row][column];
         if (visitedButton.getText().equals("Z")) {
             return true;
         } else {
             searchField.getChildren().remove(visitedButton);
             visitedButton.setStyle("-fx-background-color: #89c1c7 ");
             searchField.add(visitedButton, column, row);
             indexToContinueSearch.add(index);
             return false;
         }
     }
```

Der Else Fall muss angepasst werden. Hierfür wird der Fall in eine eigene Methode ausgelagert.

```
98 @    private boolean checkEndPosition(Index index) {
99          int row = index.getRow();
100         int column = index.getColumn();
101         buttons[row][column].setVisited(true);
102         Button visitedButton = buttons[row][column];
103         if (visitedButton.getText().equals("Z")) {
104             return true;
105         } else {
106             markFieldAsVisited(index, row, column, visitedButton);
107             return false;
108         }
109     }
110
111     private void markFieldAsVisited(Index index, int row, int column, Button visitedButton) {
112         searchField.getChildren().remove(visitedButton);
113         visitedButton.setStyle("-fx-background-color: #89c1c7 ");
114         searchField.add(visitedButton, column, row);
115         indexToContinueSearch.add(index);
116     }
```

Obenstehendes Bild ist das Ergebnis der Anpassung. Die Commit-ID lautet „59ff1f838bae86c3ebed7dd39d4041d475a2d6ba“. Die Klasse *Index* muss nicht angepasst werden, da diese dem SRP folgt.

Als nächstes wird im package „fileprocessing“ der *JSONFileProcessor* angepasst. Hier ist in der *readFile()* Methode das SRP verletzt. Es wird innerhalb der Methode eine File gelesen, aber auch geparsed. Das Parsen muss in einer eigenen Methode passieren. Dies ermöglicht zusätzlich auch leichteres Testen. Das Problem ist in der nächsten Abbildung ersichtlich.

```java
public Button[][] readFile(Path filePath) {
    Button[][] result = new Button[0][];
    JSONParser jsonParser = new JSONParser();
    try (FileReader reader = new FileReader(filePath.toString()))
    {
        //Read JSON file
        Object obj = jsonParser.parse(reader);

        JSONArray matrix = (JSONArray) obj;
        result = new Button[matrix.size()][((JSONArray)matrix.get(0)).size()];
        for(int i = 0; i<matrix.size(); i++){
            JSONArray row = ((JSONArray)matrix.get(i));
            for(int j = 0; j<row.size(); j++){
                Button tempButton = new Button(String.valueOf(row.get(j)));
                result[i][j] = tempButton;
            }
        }

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ParseException e) {
        e.printStackTrace();
    }
    return result;
}
```

Das Parsing wurde in eine eigene Methode ausgelagert. Des Weiteren wurde die Methode von *readFile* zu *processFile* umbenannt. Das Ergebnis ist in nachstehender Abbildung sowie in der Commit-ID „643a6f342231b846e3b2b920ca8e3508f866fb13“ersichtlich.

```java
public Button[][] processFile(Path filePath) {
    Button[][] result = new Button[0][];
    JSONParser jsonParser = new JSONParser();
    try (FileReader reader = new FileReader(filePath.toString()))
    {
        result = parseFile(jsonParser, reader);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return result;
}

private Button[][] parseFile(JSONParser jsonParser, FileReader reader) {
    Object obj = null;
    try {
        obj = jsonParser.parse(reader);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ParseException e) {
        e.printStackTrace();
    }

    JSONArray matrix = (JSONArray) obj;
    Button[][] result = new Button[matrix.size()][((JSONArray) matrix.get(0)).size()];
    for(int i = 0; i<matrix.size(); i++){
```

Es entstand somit eine neue Methode, welche nur für das Parsen zuständig ist. Die *parseFile()* Methode muss weiter zerlegt werden, damit auch hier das SRP verfolgt wird. Diese Änderung kann unter der Commit-ID „db6aa7f776913c9973daa68c7c22666045a37b17“ eingesehen werden. Nach dem diese Änderung durchgeführt wurde, folgt die Klasse dem SRP.

Der *XMLFileProcessor* muss nicht berücksichtig werden, da hierzu noch keine Implementierung vorhanden ist.

Im package „Frontend“ muss in der Klasse *VisualizerField* die Methode *nextSearchStep()* angepasst werden. Hier muss der Alert ausgelagert werden.

```java
public boolean nextSearchStep(){
    if (searchAlgorithm !=null){
      return searchAlgorithm.doSearch(getGrid(), getCurrentButtonField(), startField, endField);
    } else{
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle("Algorithm Error");
        alert.setHeaderText("Algorithm not set");
        String s ="Either the algorithm couldn't be recognized or it isn't set";
        alert.setContentText(s);
        alert.show();
        throw new NullPointerException("Search Algorithm isn't set");
    }
}
```

Nach der Anpassung sieht der Code wie folgt aus.

```
72    public boolean nextSearchStep(){
73        if (searchAlgorithm !=null){
74          return searchAlgorithm.doSearch(getGrid(), getCurrentButtonField(), startField, endField);
75        } else{
76            showAlgorithmError();
77            throw new NullPointerException("Search Algorithm isn't set");
78        }
79    }
80
81    private void showAlgorithmError(){
82        AlertManager alertManager = new AlertManager.Builder()
83                .setTitle("Algorithm Error")
84                .setHeaderText("Algorithm not set")
85                .setContextText("Either the algorithm couldn't be recognized or it isn't set")
86                .setAlertType(Alert.AlertType.ERROR)
87                .build();
88        alertManager.showAlert();
89    }
```

Die Commit-ID hierzu ist „87f1e9814fcb44bd205399a6932b3f76c443f473“.
Innerhalb der Klasse *FieldChecker* wurde das Prinzip ebenso verletzt. Die Klasse ist dafür da beliebige Indizes zu checken. Sie beinhaltet jedoch auch eine Methode *getNextIndices(),* welche außerdem viele weitere Methoden innerhalb der Klassen aufruft (die nichts mit *FieldChecker* wirklich zu tun haben), welche die jeweils nächsten Indices zurückliefert. Dies hat mit der eigentlichen Klasse nicht viel zu tun. Deshalb wird diese Funktionalität in eine eigene Klasse *IndicesProvider* ausgelagert. Das Ergebnis kann unter der Commit-ID „01aa99ec3507809ca934d67f6d8adb4bafb7fa41“ betrachtet werden.

Die Klasse *VisualizerHeader* könnte ebenso noch angepasst werden, in dem in der Methode *createAlgorithmHeader* die einzelnen Buttons in einer eigenen Methode angepasst werden. Dieser Schritt muss jedoch nur einmal gemacht werden (nämlich im Header). Eine mögliche Auslagerung wird im nächsten Screenshot gezeigt. Dies wird jedoch nicht commited, da ich finde, dass die Lesbarkeit und Wartbarkeit nicht erhöht wird, da Objekte geändert werden, die man nicht auf Anhieb sieht.
Vorher:

```
68          slowSearchSpeedButton.setOnAction(e->{
69              algorithmSpeed.set(2000);
70              slowSearchSpeedButton.setDisable(true);
71              midSearchSpeedButton.setDisable(false);
72              fastSearchSpeedButton.setDisable(false);
73          });
74          midSearchSpeedButton.setOnAction(e->{
75              algorithmSpeed.set(1000);
76              midSearchSpeedButton.setDisable(true);
77              slowSearchSpeedButton.setDisable(false);
78              fastSearchSpeedButton.setDisable(false);
79          });
```

Nachher:

```
68          initSlowSearch(algorithmSpeed, slowSearchSpeedButton, midSearchSpeedButton, fastSearchSpeedButton);
69          initMidSearch(algorithmSpeed, slowSearchSpeedButton, midSearchSpeedButton, fastSearchSpeedButton);
70          initFastSearch(algorithmSpeed, slowSearchSpeedButton, midSearchSpeedButton, fastSearchSpeedButton);
71
```

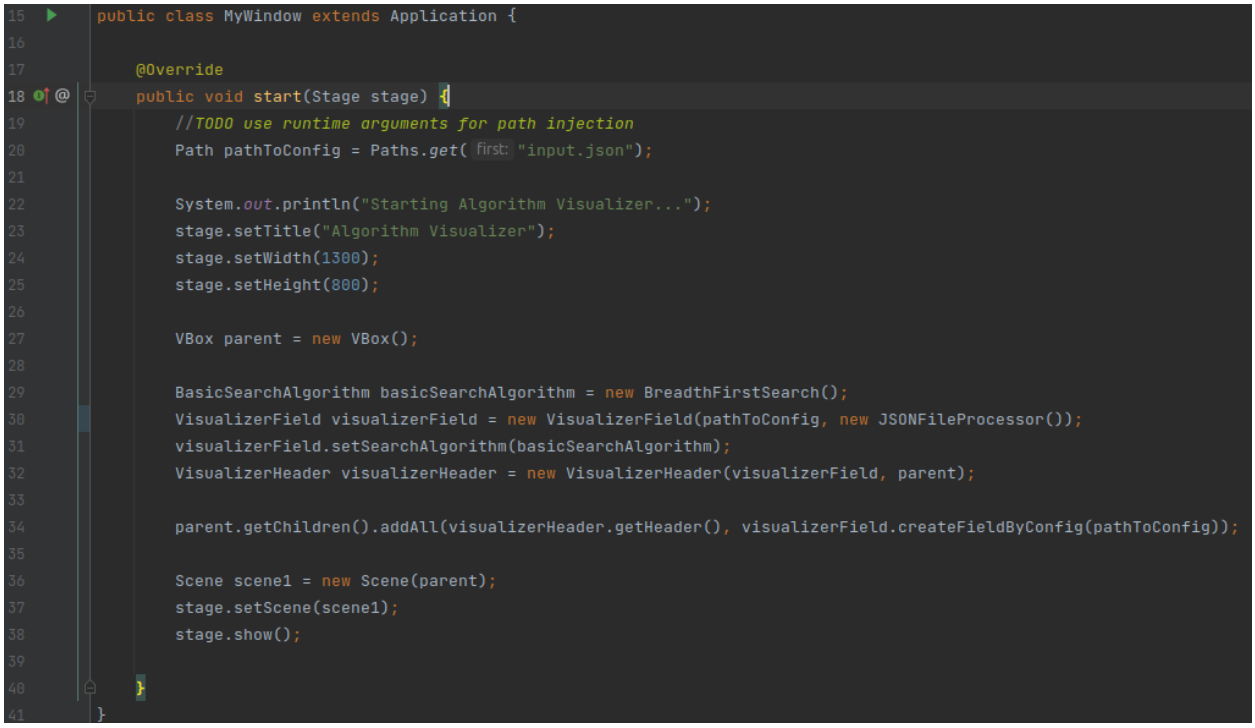
```
152 @   private void initMidSearch(AtomicInteger algorithmSpeed, Button slowSearchSpeedButton, Button midSearchSpeedButton, Button fastSearchSpeedButton) {
153         midSearchSpeedButton.setOnAction(e->{
154             algorithmSpeed.set(1000);
155             midSearchSpeedButton.setDisable(true);
156             slowSearchSpeedButton.setDisable(false);
157             fastSearchSpeedButton.setDisable(false);
158         });
159     }
160
161 @   private void initFastSearch(AtomicInteger algorithmSpeed, Button slowSearchSpeedButton, Button midSearchSpeedButton, Button fastSearchSpeedButton) {
162         fastSearchSpeedButton.setOnAction(e->{
163             algorithmSpeed.set(500);
164             fastSearchSpeedButton.setDisable(true);
165             midSearchSpeedButton.setDisable(false);
166             slowSearchSpeedButton.setDisable(false);
167         });
168     }
169
170 @   private void initSlowSearch(AtomicInteger algorithmSpeed, Button slowSearchSpeedButton, Button midSearchSpeedButton, Button fastSearchSpeedButton) {
171         slowSearchSpeedButton.setOnAction(e->{
172             algorithmSpeed.set(2000);
173             slowSearchSpeedButton.setDisable(true);
174             midSearchSpeedButton.setDisable(false);
175             fastSearchSpeedButton.setDisable(false);
176         });
177     }
```

4.2. Open / Closed Principle

4.2.1. Analyse / Begründung / Anpassungen

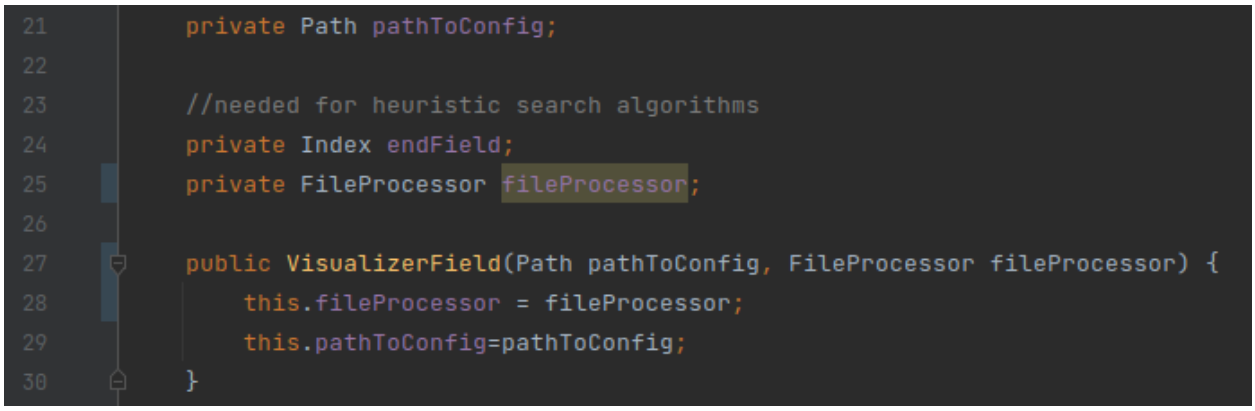
Es gilt zu sagen, dass das Open / Closed Principle von Anfang an versucht wurde zu verfolgen. Im Code gibt es keine Stelle, welche „instanceof“ verwendet. Ein Beispiel, welches das Open / Closed Principle verfolgt ist die Dateiverarbeitung. Hierfür gibt es ein Interface *FileProcessor* und zwei Implementierungen *JSONFileProcessor* und *XMLFileProcessor*. Leider muss gesagt werden, dass der XMLFileProcessor nicht vollständig ausprogrammiert wurde. Jedoch kann das OCP trotzdem dargestellt werden.

Nach kleinen Anpassungen im Code wird dies deutlicher. Die Klasse *MyWindow* ist zuständig um das Feld/Labyrinth des Frontends aus einer JSON Datei zu laden. Dies kann in folgender Abbildung betrachtet werden.



```
15  public class MyWindow extends Application {
16
17      @Override
18      public void start(Stage stage) {
19          //TODO use runtime arguments for path injection
20          Path pathToConfig = Paths.get( first: "input.json");
21
22          System.out.println("Starting Algorithm Visualizer...");
23          stage.setTitle("Algorithm Visualizer");
24          stage.setWidth(1300);
25          stage.setHeight(800);
26
27          VBox parent = new VBox();
28
29          BasicSearchAlgorithm basicSearchAlgorithm = new BreadthFirstSearch();
30          VisualizerField visualizerField = new VisualizerField(pathToConfig, new JSONFileProcessor());
31          visualizerField.setSearchAlgorithm(basicSearchAlgorithm);
32          VisualizerHeader visualizerHeader = new VisualizerHeader(visualizerField, parent);
33
34          parent.getChildren().addAll(visualizerHeader.getHeader(), visualizerField.createFieldByConfig(pathToConfig));
35
36          Scene scene1 = new Scene(parent);
37          stage.setScene(scene1);
38          stage.show();
39
40      }
41  }
```

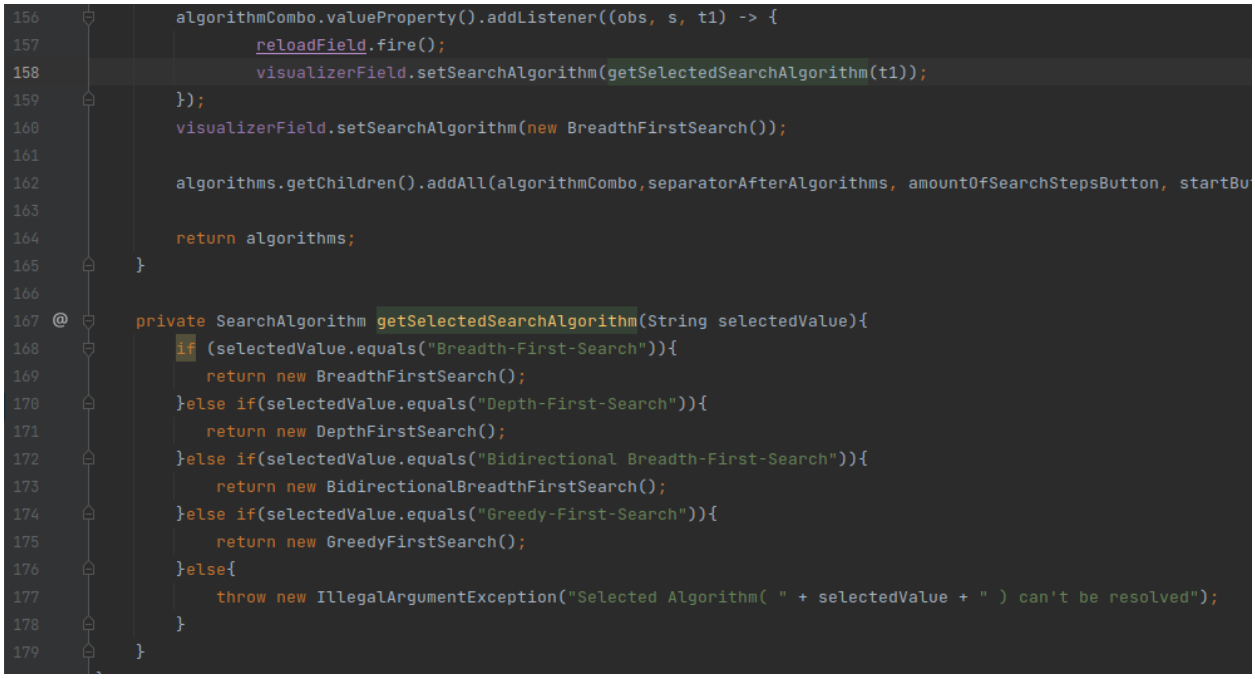
In Zeile 30 wird das Open / Closed Principle deutlich. Es wird der JSONFileProcessor übergeben. Es könnte jedoch auch einfach der XMLFileProcessor übergeben werden und am Code müsste nichts geändert werden.



```
21      private Path pathToConfig;
22
23      //needed for heuristic search algorithms
24      private Index endField;
25      private FileProcessor fileProcessor;
26
27      public VisualizerField(Path pathToConfig, FileProcessor fileProcessor) {
28          this.fileProcessor = fileProcessor;
29          this.pathToConfig=pathToConfig;
30      }
```

In der obigen Abbildung wird deutlich, dass ein *FileProcessor* übergeben werden muss. Dies kann (aktuell) ein *JSONFileProcessor* oder ein *XMLFileProcessor* sein. Je nachdem, welche konkrete Implementierung übergeben wurde, wird dann die *processFile* Methode von *JSONFileProcessor* oder *XMLFileProcessor* aufgerufen. Dies kann beliebig durch weitere Klassen erweitert werden. Es ist jedoch geschlossen für Änderungen. Durch kleine Anpassungen wurde das OCP realisiert. Dies kann in Commit-ID „66fbc468a0bbb8e6fb9784ae56a03c4a17c458c0“ angesehen werden. Eine größere

Implementierung des OCP stellen die einzelnen Suchalgorithmen dar. Hierzu gibt es ein Interface *SearchAlgorithm*. Dieses Interface ist wiederum auf 2 Interface *HeuristicSearchAlgorithm* und *BasicSearchAlgorithm* unterteilt. Aufgrund des OCP können allerlei neuer Suchalgorithmen (mit oder ohne Heuristik) eingebaut werden und können „direkt“ im Frontend verwendet werden. Dies wird in folgendem Screenshot deutlich:



In Zeile 158 wird der Suchalgorithmus, abhängig von der Auswahl, gesetzt. Gesetzt wird dann die konkrete Implementierung des Interfaces. Auf den konkreten Klassen kann dann später die *doSearch* Methode aufgerufen werden. Es können also beliebig neue Algorithmen in der Methode *getSelectedSearchAlgorithm* eingefügt werden. Das OCP wird gewährt.

Das gleiche Verhalten wurde auch noch für Suchalgorithmen geplant. Diese wurden jedoch nicht mehr implementiert.

4.3. Liskov Substitution Principle

Das LSP besagt, dass eine Basisklasse jederzeit durch ihre abgeleiteten Klassen ersetzt werden können. Innerhalb dieses Projektes wurde nur zwei Mal Vererbung verwendet. Auf diese beiden Fälle wird nun genauer eingegangen.

4.3.1. Analyse / Begründung / Anpassungen

Es wurde eine Klasse *NumberTextField* erstellt, welche die Klasse *TextField* von JavaFX erweitert. Sie sorgt dafür, dass nur Zahlen in ein Textfeld eingegeben werden können.

```
15      @Override
16      public void replaceText(int i, int i1, String s) {
17          if (isTextNumericOnly(s)) {
18              super.replaceText(i,i1,s);
19              setStyle("");
20          } else {
21              setStyle("-fx-control-inner-background: #a81830");
22          }
23      }
24
25      @Override
26      public void replaceSelection(String s) {
27          if (isTextNumericOnly(s)) {
28              super.replaceSelection(s);
29              setStyle("");
30          } else {
31              setStyle("-fx-control-inner-background: #a81830");
32          }
33      }
34
35  @   private boolean isTextNumericOnly(String inputText) { return inputText.matches( regex: "\\d+"); }
```

Um dieses Ergebnis zu erreichen, wurden die zwei Klassen *replaceText* und *replaceSelection* überschrieben. Würde man diese zwei Methoden mit der Methode *isTextNumericOnly* in die Basisklasse verschieben, würde der Code immer noch funktionieren (auch ohne Nebeneffekte). Dies kann jedoch nicht durchgeführt / getestet werden, da die Basisklasse eine Klasse aus JavaFx ist.

Die zweite Klasse, welche Vererbung verwendet ist die Klasse *Button*. Sie erweitert die JavaFX Klasse *Button*.

```
3   public class Button extends javafx.scene.control.Button {
4
5       private boolean visited = false;
6
7       public boolean isVisited() { return visited; }
10
11      public void setVisited(boolean visited) { this.visited = visited; }
14
15  @   public void switchField(String field){
16          if (field.equals("0")){
17              super.setText("X");
18              super.setStyle("-fx-background-color: #99bfa3");
19          } else if(field.equals("X")){
20              super.setText("0");
21              super.setStyle("");
22          }
23      }
24
25      public Button(String text) { super.setText(text); }
28
29      public boolean isStartField() { return getText().equals("S"); }
32  }
```

Auch hier kann der Code so in die Basisklasse ausgelagert werden und wäre noch funktionsfähig.

Allem in allem, ist zu sagen, dass er sehr schwer ist, dass LSP durchzusetzen, wenn nur zwei Mal Vererbung verwendet wird und diese Erweiterung von Klassen aus JavaFX darstellen.

4.4. Interface Segregation Principle

Die Interfaces, welche im Code benutzt werden, sind *BasicSearchAlgorithms, HeuristicSearchAlgorithm*, *SearchAlgorithm* und *FileProcessor*. Das ISP muss also gewahrt werden. Nachstehend wird sich um den *FileProcessor gekümmert, da dieser das ISP* verletzt.

4.4.1. Analyse / Begründung / Anpassungen

Der *FileProcessor* stellt zwei Methoden *processFile()* und *writeFile()* zur Verfügung. Die konkreten Implementierungen müssen also immer eine Datei verarbeiten und schreiben können. Sollte jedoch eine neue konkrete Implementierung kommen, welche beispielsweise Dateien in einem speziellen Format von einem Webserver lädt, müsste das *FileProcessor* Interface eine neue Methode *establishWebserverConnection()* haben. Die Methode ist wichtig, da eventuell neue Implementierungen kommen, die auf unterschiedlichen Weisen von Servern lesen. Würde das Interface also angepasst werden, müssten dies auch der *JSONFileProcessor* sowie der *XMLFileProcessor* implementieren (Obwohl sie die Methode nicht brauchen). Deshalb wird ein neues Interface mit dem Namen *RemoteConnectionEtablisher* angelegt. Des Weiteren wird beispielhaft eine konkrete Implementierung eingefügt, um das ISP zu verdeutlichen. Die neue Klasse *RestRequestProcessor* ist nun in der Lage Daten per Restaufruf zu lesen und diese dann als File verarbeiten und wieder zurückschreiben. Sollten neue „internetbasierte“ *FileProcessor* hinzukommen, kann das *RemoteConnectionEtablisher* Interface wieder verwendet werden. Alle Klassen müssen jedoch immer das *FileProcessor* Interface implementieren. Der Aufrufer (*VisualizierField)* benötigt nur dieses Interface, beziehungsweise die Implementierung.
Die durchgeführten Änderungen können in der Commit-ID „87f1e9814fcb44bd205399a6932b3f76c443f473“ genauer betrachtet werden.

4.5. Dependency Inversion Principle

Folgend wird ein Beispiel aufgezeigt, welches dem DIP nach der Anpassung folgt.

4.5.1. Analyse / Begründung / Anpassungen

Die Klasse *GreedyFirstSearch* beinhaltet den Algorithmus, welcher den GreedyFirstSearch Suchalgorithmus darstellt. Hierfür müssen jeweils bestimmte (nächste) Felder überprüft werden, ob diese „frei“, „nicht zugänglich“ oder das Ziel ist. Die Überprüfung übernimmt die Klasse *FieldChecker*. Der Aufruf wird in folgendem Screenshot dargestellt.

```
51      private boolean canAnyIndexReachTheGoal(GridPane searchField, Button[][] buttons, Index endField) {
52          boolean breadthFirstSearchNeeded = false;
53          for (Index index : currentIndex) {
54              Direction directionToGo = directionToGoNext(index, endField);
55              if (fieldChecker.canNextFieldByDirectionBeReached(index, directionToGo, buttons)) {
56                  breadthFirstSearchNeeded =false;
57                  flushOfAllVisitedNeeded=true;
58                  if (checkIfTargetCanBeReached(searchField, buttons, index, directionToGo)){
59                      return true;
60                  }
61                  break;
62              } else{
63                  breadthFirstSearchNeeded = true;
64              }
65          }
```

In Zeile 55 ist der Aufruf erkennbar. Der Algorithmus ist somit direkt abhängig von dem Fieldchecker. Sollte sich die Implementierung des *fieldCheckers* ändern, muss der Code in dem Screenshot ebenfalls angepasst werden. Des Weiteren können keine weitere *FieldChecker* einfach verwendet werden. Als Beispiel könnte ein neuer *FieldChecker* implementiert werden, welcher innerhalb eines drei-dimensionalen Feldes funktioniert. Aufgrund der zwei oben genannten Gründe muss eine bessere Abstraktion eingeführt werden, damit DIP gewährt wird. Um das Beispiel etwas zu verdeutlichen, wurde der ehemalige *Fieldchecker* in *TwoDimensionalFieldChecker* umbenannt und es wurde eine Klasse *ThreeDimensionalFieldChecker* angelegt. Die Klasse *GreedyFirstSearch* verwendet jetzt nur noch das Interface *FieldChecker* anstatt der konkreten Implementierung. Die Änderungen können unter der Commit-ID „44d011e18eff5e566d1ec7a7f67b243100a56729“ genauer betrachtet werden.

Des Weiteren wurde das DIP innerhalb der Klasse verletzt, da die konkrete Klasse *IndicesProvider* verwendet wird. Sollte hier jedoch eine Änderung auftreten muss auch die Klasse *GreedyFirstSearch* sich wahrscheinlich anpassen. Des Weiteren können keine weitere *IndicesProvider* erstellt und einfach verwenden werden (Also im Prinzip gleiches Problem wie Beispiel vorher). Der Aufruf des *IndicesProvider* wird in folgendem Screenshot dargestellt.

```
91      private boolean checkIfTargetCanBeReached(GridPane searchField, Button[][] buttons, Index index, Direction directionToGo) {
92          List<Index> nextIndices = indicesProvider.getNextIndices(index, directionToGo, buttons);
93          for (Index nextIndex : nextIndices) {
94              if (checkEndPosition(nextIndex)) {
95                  System.out.println("Target found at: [" + nextIndex.getRow() + "][" + nextIndex.getColumn() + "]");
96                  ResultDisplayer.displayResult(buttons, nextIndex, searchField);
97                  return true;
98              }
99          }
100         return false;
101     }
```

Zeile 91 zeigt den Aufruf des *IndicesProvider*. Um das Problem zu lösen, wird wie vorher schon vorgegangen -> Abstrahierung. Es wird ein Interface *FieldinfoProvider* erstellt,

welches dann von dem *IndiceProvider* implementiert wird. *IndiceProvider* wird in *TwoDimensionalIndiceProvider* umbenannt. Des Weiteren wird (wieder) beispielhaft eine Klasse *ThreeDimensionalIndiceProvider* erstellt, damit das Vorhaben deutlicher wird. Unter der Commit-ID „6d5de496894cf1ef405784acc0cb7e924b53b5ea“ können die Änderungen nachvollzogen werden.

5. GRASP

5.1. Information Expert

IE besagt, dass für eine neue Aufgabe derjenige zuständig ist, der schon das meiste Wissen für die Aufgabe hat.
Dies kann innerhalb des Projektes zum Beispiel mit der Klasse *Index* gezeigt werden. Diese Klasse repräsentiert einen Index im Feld und hat im Laufe der Entwicklung eine neue Methode *isSameField(Index fieldToCheck)* erhalten. Diese Methode vergleicht die X- und Y-Achse mit dem „this“ Feld und mit dem übergegebenen Feld. Würde hierfür eine Hilfsklasse angelegt worden, mit einer Methodensignatur wie: *isSameField(Index field1, Index field2)*, würde das Information Expert verletzen.

5.2. Creator

Das Erzeuger-Prinzip wurde eingehalten, da Instanzen von Klassen erzeugt wurden, die entweder eine Aggregation von B ist (z.B. *VisualizerField -> Index*), Objekte von B verarbeitet (z.B. *VisualizerField -> GridPane*) oder von B abhängt (z.B. *VisualizerField -> Path*).

5.3. Controller

Der Controller ist die erste Schnittstelle nach der GUI und delegiert an andere Module. In Falle des AlgorithmVisualizer stellt dies *VisualizerField* dar.

```
71      public boolean nextSearchStep(){
72          if (searchAlgorithm !=null){
73            return searchAlgorithm.doSearch(getGrid(), getCurrentButtonField(), startField, endField);
74          } else{
75              showAlgorithmError();
76              throw new NullPointerException("Search Algorithm isn't set");
77          }
78      }
```

In der Abbildung ist zu erkennen, wie (die erste Schnittstelle nach der GUI) an den Suchalgorithmus delegiert. Der Suchalgorithmus übernimmt dann die Delegierung / Verarbeitung der Klassen, welche nicht für die Nicht-Benutzeroberfläche bestimmt sind.

5.4. Low Coupling

Nachstehend werde Beispiele zur losen Kopplung aufgezeigt. Diese hängen teils auch schon mit vorherigen Beispielen zusammen.

Als Beispiel kann *VisualizerField* angesehen werden. Die Klasse beinhaltet unteranderem eine Abhängigkeit zu dem Interface *SearchAlgorithm*. Es wäre keine lose Kopplung, wenn die Klasse direkt eine Abhängigkeit zu einem konkreten Algorithmus hätte. Der Algorithmus kann jederzeit ausgetauscht werden. Die lose Kopplung zeigt auch auf, dass jederzeit konkrete Implementierungen aus zwei verschiedenen Interfaces (*BasicSearchAlgorithm* und *HeuristicSearchAlgorithm*) als Implementierung für *SearchAlgorithm* verwendet werden können.

Ein weiteres Beispiel ist noch in der Klasse *VisualizerField*. Die Klasse hält eine Abhängigkeit zu dem Interface *FileProcessor*. Auch hier kann wieder individuell zwischen der konkreten Implementierung entschieden werden ohne das *VisualizerField* angepasst werden muss.

5.5. High Cohesion

Um die hohe Kohäsion zu untersuchen, werden verschiedene Klassen betrachtet.

Zunächst *BreadthFirstSearch*:

Klassenvariablen	Erwartet in der Klasse?
indexToContinueSearch	Ja
currentIndex	Ja
searchField	Ja
buttons	Nein
firstSearch	Ja
isSearchForGreedyFirstSearch	Ja

Das Array mit den Buttons wird eigentlich in der Klasse nicht erwartet. Es wird eigentlich erwartet, dass die buttons innerhalb des searchFieldes verarbeitet werden.

Nach Inspektion des Codes ist es aufgefallen, dass alle Suchalgorithmen das gleiche Problem haben und ein button array sowie das searchField haben. Somit wurde es in allen Klassen angepasst. Der (ziemlich große) Commit kann unter folgender ID angesehen werden „f17b5647a21d2624565b68cd172adb268de0ddaf“. Nach der Anpassung weisen alle Algorithmen eine hohe Kohäsion auf, da alle Klassenvariablen erwartet werden und benutzt werden. Im Nachgang ist noch aufgefallen, dass

VisualizerField noch das alte Array mit buttons verwendet. Dies wurde unter der Commit-ID „62134690b0df164167ac19277922deb1c56e27f7“ entfernt. Insgesamt kann gesagt werden, dass durch die neue Klasse *SearchField* eine höhere Kohäsion innerhalb vieler Klassen herrscht.

5.6. Polymorphism

Polymorphie wird oft innerhalb des Projektes verwendet und wurden in diesem Dokument auch schon indirekt beschrieben. Als Beispiel können die Suchalgorithmen betrachtet werden, welche alle eine konkrete Implementierung desselben Interfaces darstellen. Ein weiteres Beispiel für Polymorphie im Projekt ist die Dateiverarbeitung. Alle konkrete Implementierung von dem Interface *FileProcessor* können unabhängig verwendet werden.

```
16  public class VisualizerField {
17
18      //private GridPane grid;
19      private SearchField searchField;
20      private SearchAlgorithm searchAlgorithm;
21      private Path pathToConfig;
22
23      //needed for heuristic search algorithms
24      private FileProcessor fileProcessor;
25
26      public VisualizerField(Path pathToConfig, FileProcessor fileProcessor) {
27          this.fileProcessor = fileProcessor;
28          this.pathToConfig=pathToConfig;
29      }
```

In der Abbildung ist zu erkennen, dass in den Konstruktor des *VisualizerField* ein *FileProcessor* mitgegeben wird. Dieser kann eine beliebige Implementierung sein, da gegen das Interface programmiert ist.

```
39      public Node createFieldByConfig(Path pathToConfig) {
40          searchField = new SearchField();
41          searchField.initField(fileProcessor.processFile(pathToConfig));
42          return searchField.getGrid();
43      }
```

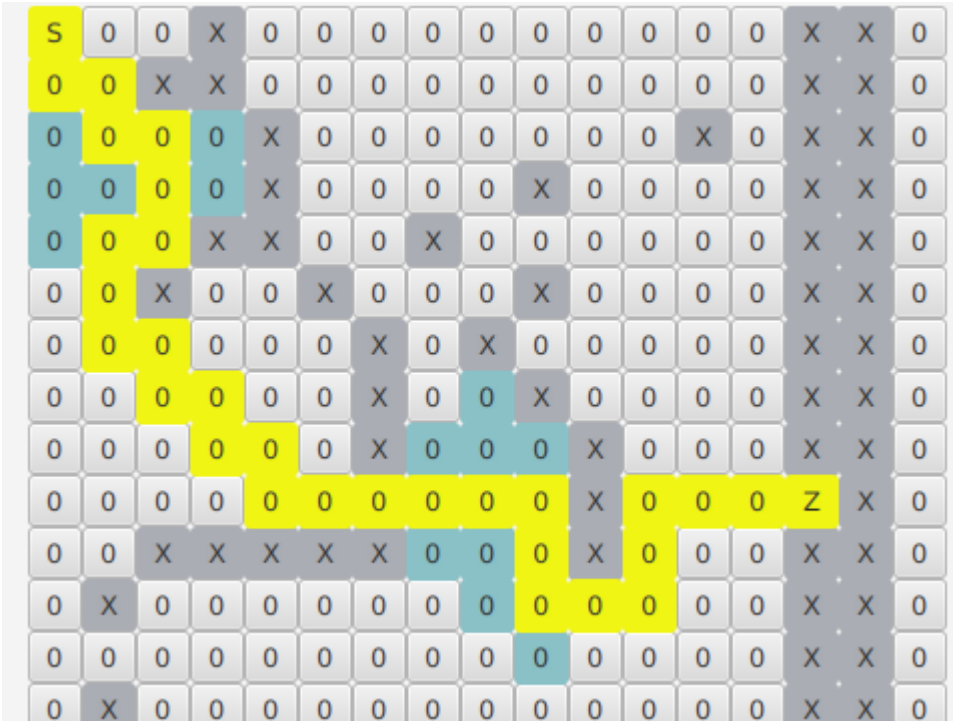
Nachstehendes Bild zeigt auf, wie das *VisualizerField* erzeugt wird. Es wurde ein Beispiel hinzugefügt, damit deutlich wird, dass unterschiedliche Implementierungen in den Konstruktor mitgegeben werden können:

```
30          VisualizerField visualizerField = new VisualizerField(pathToConfig, new JSONFileProcessor());
31          // Auch möglich und Code würde noch funktionieren
32          VisualizerField visualizerField2 = new VisualizerField(pathToConfig, new XMLFileProcessor());
```

Es ist zu erkennen, dass der Code noch genauso wie vorher funktioniert, aber jetzt werden XML-Dateien anstatt JSON-Dateien verarbeitet.

5.7. Pure Fabrication

Innerhalb dieses Projektes gibt es Pure Fabrication. Ein Beispiel stellt der *AlertManager* dar. Dieser ist nicht Teil der Domäne und ist eine Hilfsklasse. Er ist dazu da, einen Alert zu erzeugen und diesen im Frontend darzustellen. Eine weitere Pure Fabrication ist *ResultDisplayer*. Die Klasse ist dazu da, den gefundenen Pfad im Frontend darzustellen (Gelber Pfad).



5.8. Indirection / Delegation

Delegation führt unteranderem *VisualizerField* durch. Als Beispiel kann folgende Abbildung betrachtet werden. Sie stellt zeigt die Methode *nextSearchStep* aus der Klasse *VisualizerField*.

```
45      public boolean nextSearchStep(){
46          if (searchAlgorithm !=null){
47              return searchAlgorithm.doSearch(searchField);
48          } else{
49              showAlgorithmError();
50              throw new NullPointerException("Search Algorithm isn't set");
51          }
52      }
```

Das *VisualizerField* delegiert „nur“ an den jeweiligen Suchalgorithmus weiter. Sie überprüft vorher noch ob ein Suchalgorithmus gesetzt ist. Dies könnte angepasst werden, in dem der Suchalgorithmus als Parameter übergeben wird. Dann würde der Wechsel zwischen Algorithmen aber nicht mehr so leicht funktionieren.

5.9. Protected Variations

Protected Variations werden durch die Programmierung gegen die Interfaces gewährleistet. Als Beispiel kann wieder die Dateiverarbeitung betrachtet werden.

6. DRY

Das DRY Prinzip wurde eingehalten, da kein Code kopiert wurde, sondern in Methoden ausgelagert wurde. Das Konzept wurde bereits der Implementierung berücksichtigt.

7. Unit Testing

7.1. Tests

Nachstehend wird genauer auf die einzelnen Tests eingegangen. Die packages des Testverzeichnises sind genauso aufgebaut wie die packages des Hauptcodes. Klasse, welche nur angelegt worden sind, um ein Prinzip aufzuzeigen werden nicht getestet, da (noch) kein Code vorhanden ist (Bsp. *ThreeDimensionalFieldChecker)*.

Das package „common“ beinhaltet unteranderem die FieldChecker und IndiceProvider. Diese wurden mithilfe von JUnit getestet. Die Tests sind unter folgender Commit-ID einsehbar „a9b0881287874e449868957beac4b966b22a2fe5“. Durch die Tests konnte sogar eine obsolete Methode identifiziert, welche dann ebenso innerhalb des Commits entfernt wurde.

Im Zuge der Tests konnten weitere obsolete Felder identifiziert werden. Diese sind alle in den jeweiligen Commits enthalten.

Die Tests für die Suchalgorithmen wurden erstellt und gepusht. Diese erstrecken sich über 2 Commits. Die IDs sind folgende: „705fcae5aa29eaa40cb5b2d083b71c4bacb5bd2f“ und „87002116cc9f8ed26e6ca1a5a1b2dee8c4658e81“.

Das Package FileProcessing wurde ebenso getestet. Hierbei wurde jedoch nur der *JSONFileProcessor* getestet, da er die einzige Klasse ist, die bereits konkret implementiert wurde. Die Signatur der Methode *processFile* des Interfaces *FileProcessor* wurde angepasst, damit ein *FileParser* mitgegeben werden kann. Dies hat den Vorteil, dass eine Fakeimplementierung *FakeJSONParser* im Testcase verwendet werden kann. Die Tests zu der Klasse *JSONFileProcessor* sowie die Fakeklasse und weitere Änderungen sind unter der Commit-ID „e40fd824f36f445bc10b3c506d15dd60ced86c8f“ einsehbar.

Das package Frontend erhielt einen Test für das *NumberTextField.* Die Tests sind einsehbar unter den Commits „e40fd824f36f445bc10b3c506d15dd60ced86c8f“ und „961644ce100d841d0b1174a78ee10e9cc22e1163“. Weitere Klassen des Frontend-Packages stellen schwerere Tests dar, da es sich um viele GUI Elemente handelt. Diese wurden manuell durch mich getestet, während das Programm lief.

7.2. ATRIP-Regeln

Die ATRIP-Regeln wurden so gut es ging eingehalten. Dies wird in diesem Kapitel gezeigt.

Automatic (A) wurde eingehalten, da durch den Befehl von *mvn install* alle Tests ausgeführt werden, bevor das Projekt gebaut wird. Dies war einer der Gründe wieso sich für Maven als Buildtool entschieden wurde.

Thorough (T) wurde so gut es ging eingehalten. Es wurden viele diverse Pfade innerhalb des Codes getestet.

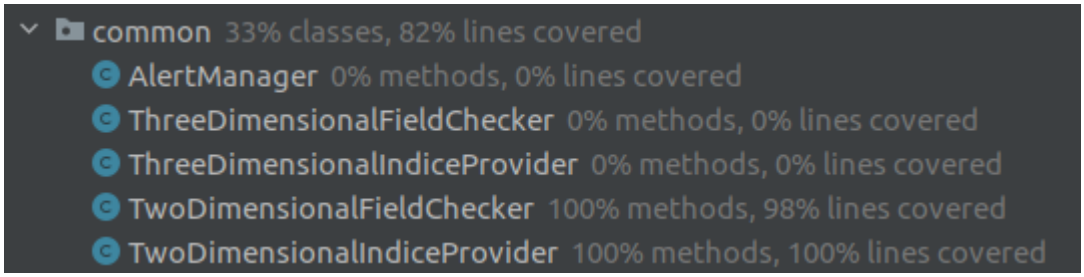
Repeatable (R) wurde eingehalten, da die Tests immer wieder ausgeführt werden können und immer dasselbe Ergebnis liefern.

Independent (I) wurde eingehalten, da die Tests so atomar wie möglich gehalten sind. Wo es möglich war, wurde jede Methode einzeln getestet, auch wenn die Methode noch so klein war. Alle Tests können einzelnen durchgeführt werden und „räumen“, wenn nötig die benötigten Ressourcen wieder auf. Kein Test hängt von einem anderen Test ab.

Professional (P) wurde eingehalten, da aus den Testnamen deutlich wird, was der Test macht. Es gibt jedoch nicht mehr Testcode als Produktionscode. Dies ist unteranderem auf die Menge an Frontend-Elemente zurückzuführen.

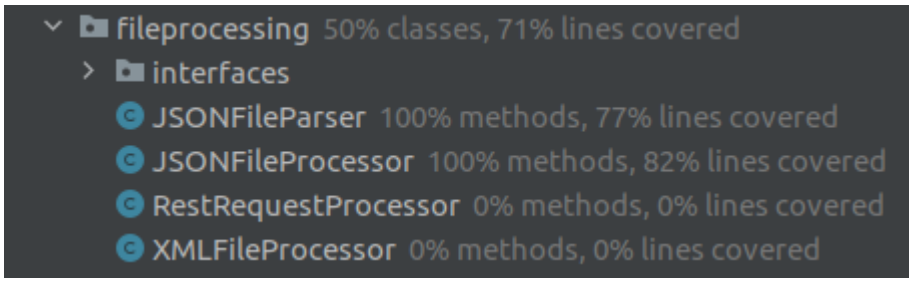
7.3. Code Coverage

Die Code Coverage liegt komplett bei 57%. Vom wichtigsten Paket des Projektes (common) beträgt die Code Coverage jedoch 82%, was als sehr gut empfunden wird.



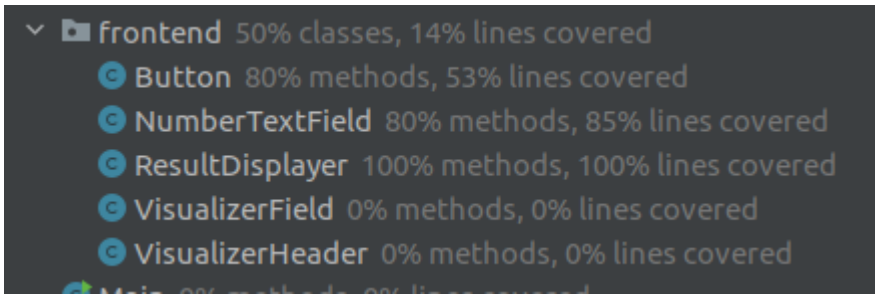
Innerhalb dieses Packages ist die Prüfung, ob ein nächstes Feld besucht werden muss oder nicht. Also die eigentliche Logik des Programms. Die 3D-Klassen können ignoriert werden, da aktuell noch keine Implementierung vorhanden ist. Die Suchalgorithmen wurden ebenso getestet und weisen 60% auf. Dies ist darauf zurückführen, dass die Klasse *BidirectionalBreadthFirstSearch* nicht so ausgiebig getestet wurde. Dies ist jedoch verkraftbar, da eigentlich intern nur zwei Mal *BreadthFirstSearch* aufgerufen wird. *BreadthFirstSearch* weist 80% CC auf.

Im package fileprocessing wurde der einzige, derzeit implementierte Fileprocessor getestet. Der *JSONFileProcessor* weist eine CodeCoverage von 82% auf.



Klassen die noch keine Implementierung haben wurden nicht getestet.

Im frontend package wurden die Hilfsklassen getestet, jedoch keine GUI Elemente, wie *VisualizerHeader* oder *VisualizerField*. Dies erklärt die geringe CC in diesem package.



Um die Code-Coverage in zukünftigen Projekten höher zu bekommen, kann eventuell auf Test-Driven-Design zurückgegriffen werden.

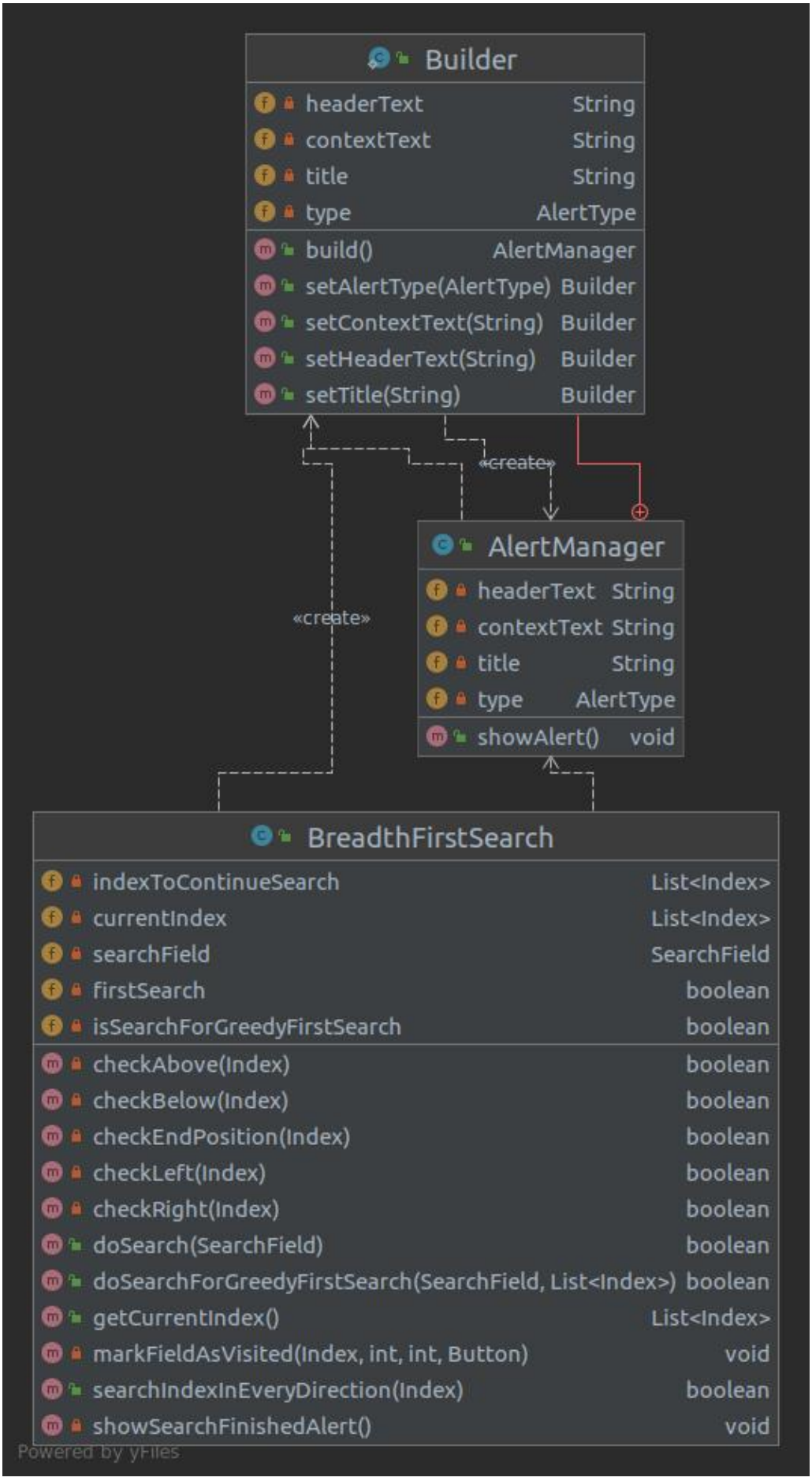
8. Entwurfsmuster (Builder-Pattern)

Innerhalb des Projektes wurde das Entwurfsmuster des Erbauers verwendet. Im Dokument unter SRP wurde bereits erwähnt als dieses entstand. Deshalb ist leider nicht mehr möglich ein vorher UML-Diagramm zu betrachten. Es wird jedoch ein „Jetziges-UML“ in diesem Kapitel dargestellt.

Das Erbauer-Pattern wurde verwendet, um einen Alert zu erzeugen, welcher im Frontend angezeigt werden kann. Der Alert kann unterschiedliche Nachrichten enthalten. Dem Alert musste man jedoch 3 verschiedene String Parameter (Title, HeaderText, ContextText) und einen Type Parameter mitgeben. Es kam vor, dass die Parameter vertauscht wurden und der ContextText dann im Title zum Beispiel stand. Also wurde sich für das Erbauer-Pattern entschieden. In nächster Abbildung ist gut zu erkennen wie der Alertmanager über den Builder zusammengebaut wird.

```
private void showSearchFinishedAlert() {
    AlertManager alertManager = new AlertManager.Builder()
            .setTitle("Information")
            .setHeaderText("Target could not be found")
            .setContextText("Either there is no target or the target couldn't be reached")
            .setAlertType(Alert.AlertType.INFORMATION)
            .build();
    alertManager.showAlert();
}
```

Es ist nun deutlich schwieriger die Parameter zu vertauschen, da man die passende Methode aufrufen muss. Es ist nun auch möglich manche Parameter wegzulassen, falls man beispielsweise keinen ContextText benötigt. Dies wurde alles durch das Pattern ermöglicht.


Builder
headerText String
contextText String
title String
type AlertType
build() AlertManager
setAlertType(AlertType) Builder
setContextText(String) Builder
setHeaderText(String) Builder
setTitle(String) Builder
«create»
AlertManager
headerText String
contextText String
title String
type AlertType
showAlert() void
«create»
BreadthFirstSearch
indexToContinueSearch List<Index>
currentIndex List<Index>
searchField SearchField
firstSearch boolean
isSearchForGreedyFirstSearch boolean
checkAbove(Index) boolean
checkBelow(Index) boolean
checkEndPosition(Index) boolean
checkLeft(Index) boolean
checkRight(Index) boolean
doSearch(SearchField) boolean
doSearchForGreedyFirstSearch(SearchField, List<Index>) boolean
getCurrentIndex() List<Index>
markFieldAsVisited(Index, int, int, Button) void
searchIndexInEveryDirection(Index) boolean
showSearchFinishedAlert() void
Powered by yFiles

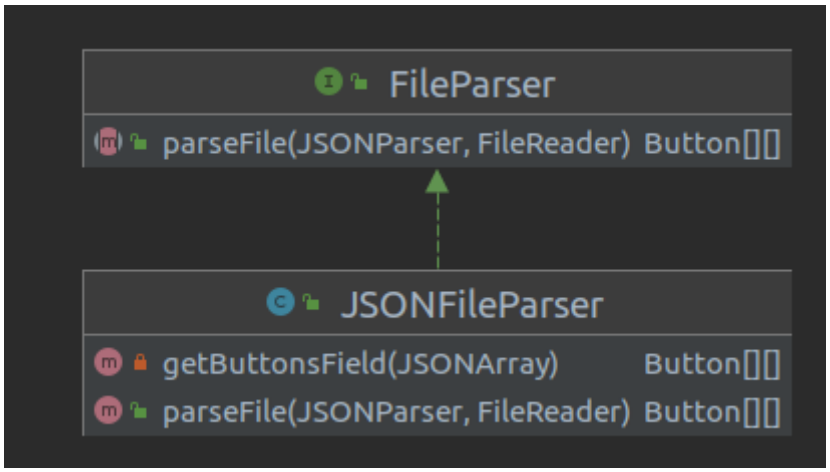
Oben ist das UML-Klassendiagramm mit dem Builder zu erkennen. Jede weitere Klasse, welche den *AlertManager* benötigt, baut sich diesen über den *Builder*. Weitere Klassen die zum Beispiel den *Alertmanager* benötigen sind alle anderen Suchalgorithmen. (Nicht im UML-Diagramm enthalten. Es wäre einfach der jeweilige Suchalgorithmus, welcher dann einen „create“-Pfeil auf den Builder hat).

9. Refactoring

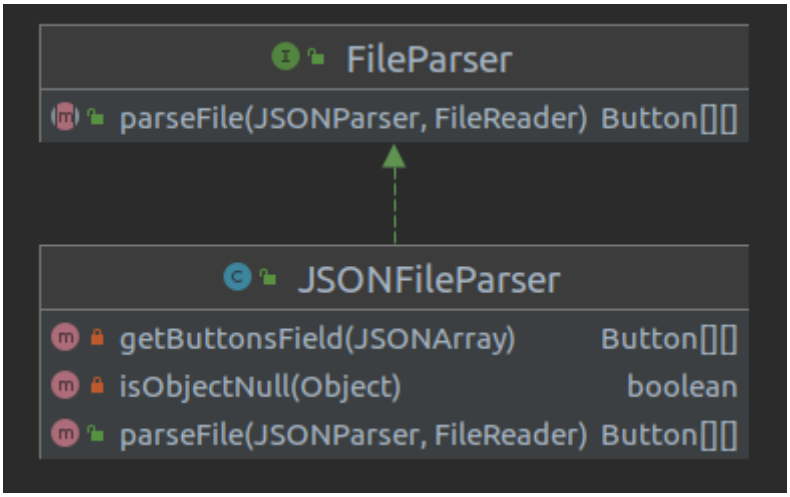
Es gilt zu sagen, dass schon viel Refactoring im SOLID Kapitel angewendet und aufgezeigt wurde. In diesem Kapitel können nur noch (kleinere) Codesmells beseitigt werden und keine riesige Refactorings durchgeführt werden, da keine mehr wirklich möglich sind und, vorallem im SRP, bereits sehr viel refactored wurde.

9.1. Code Smells identifizieren

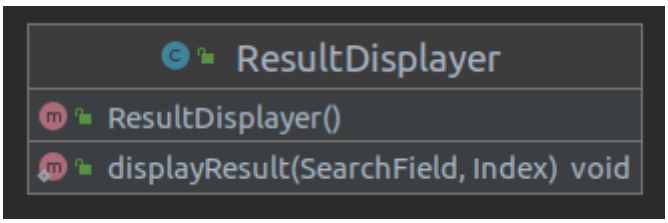
In der Klasse *JSONFileParser* wurde ein Code Smell identifiziert. In der Methode *parseFile* könnte eine NullPointerException auftreten, da in Zeile 25 *getButtonsField* aufgerufen wird und *obj* übergeben wird. *Obj* ist jedoch null, wenn es im try-Catch-Block nicht zu einem Fehler kommt. Diese Stelle muss angepasst werden. Das UML-Diagramm vorher wird in folgender Darstellung aufgezeigt. Es wird nur auf die Klasse *JSONFileParser* eingegangen in dem Diagramm, da für dieses Refactoring keine externen Klassen / Abhängigkeiten wichtig sind.



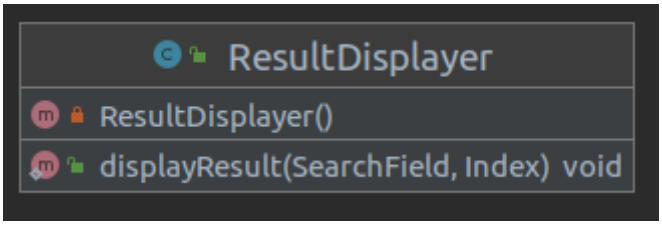
Es wurde eine neue Überprüfung eingefügt, zum Checken, ob das Objekt null ist. Es wurde eine neue Methode angelegt, damit auch ein kleiner Ausgabetext / Exception ausgegeben werden kann. Die Beseitigung des Code Smells ist unter der Commit-ID „20ffda7c8c25a16471ceed4cf00dbc43d7853402“ einsehbar. Das neue UML-Klassendiagramm sieht wie folgt aus:



Als nächster, kleiner Codesmell, wurde entdeckt, dass in der Klasse *ResultDisplayer* nur eine statische Methode vorhanden ist.



displayResult stellt hierbei die statische Methode dar. Es gibt jedoch auch noch den impliziten public Construktor. Es kann also aktuell ein Object von *ResultDisplayer* erzeugt werden, was nicht Sinn der Klasse ist, da man mit dem Objekt nichts machen kann. Deshalb wird der Konstruktor privat gemacht. Des Weiteren wurde noch ein unbenutzter Import entfernt. Die Änderungen können unter der Commit-ID „5afc0b1ed99ca8a030eea08b0dcd8a883b8fc074“ sowie in dem „Nachher-UML“ Diagramm betrachtet werden. Das Nachher-UML Diagramm sieht wie folgt aus:



Innerhalb der Klasse *VisualizerField* wurde auskommentierter / unbenutzter Code sowie unbenutzte Imports entfernt. Für diese Änderungen macht ein UML-Klassendiagramm kein Sinn. Die Änderungen können aber unter der Commit-ID „8dc4d0b7ce6db61b17bfdae8c26536f17e1fc8cc“ betrachtet werden.

10. Legacy Code

Es wurden bereits im Kapitel des Testings beschrieben wie ein Interface extrahiert wurde und ein Fakeobjekt dadurch erstellt werden konnte. Nachstehend werden zwei, bereits in der Entwicklung, gebrochene Abhängigkeiten beschrieben.

```
23      public VisualizerField(Path pathToConfig, FileProcessor fileProcessor) {
24          this.fileProcessor = fileProcessor;
25          this.pathToConfig=pathToConfig;
26      }
```

Der Konstruktor erhielt früher einen *JSONFileParser* anstelle des *fileProcessor*. Direkt in der Entwicklung wurde klar, dass auch mal eine XML-Datei in Zukunft verarbeiten werden konnte. Also entstanden mehrere Methoden wie *parseJSONFile, parseXMLFile.* Dies führte zu sehr viel Code und war auch aufwendig zu testen. Deshalb wurde ein Interface *FileProcessor* abstrahiert und als Übergabeparameter definiert. *XMLFileProcessor* und *JSONFileProcessor* stellen eine Implementierung des Interfaces dar und *VisualizerField* kann dann die Methoden des Interfaces aufrufen und es werden keine neuen Methoden benötigt. Das gleiche Verhalten war auch der Fall mit den Suchalgorithmen. *VisualizerField* hatte Klassenvariablen die wie folgt aussahen:

- Private BreadthFirstSearch bfs = new BreadthFirstSearch();
- Private GreedyFirstSearch gfs = new GreedyFirstSearch();
- …

Dazu gab es die passenden Methoden wie *doBreadthFirstSearch(), doGreedyFirstSearch() …* . Dieses Problem wurde gelöst, in dem ein Interface *SearchAlgorithm* abstrahiert wurde.

```
57      public void setSearchAlgorithm(SearchAlgorithm searchAlgorithm){
58          this.searchAlgorithm = searchAlgorithm;
59      }
```

Innerhalb der Klasse *VisualizerField* wurde also der Suchalgorithmus gesetzt und kann dann wie folgt verwendet werden:

```
38      public boolean nextSearchStep(){
39          if (searchAlgorithm !=null){
40            return searchAlgorithm.doSearch(searchField);
41          } else{
42              showAlgorithmError();
43              throw new NullPointerException("Search Algorithm isn't set");
44          }
45      }
```

Es ist nun nicht mehr wichtig, welcher Suchalgorithmus gesetzt wurde, da die Methoden auf dem Interface aufgerufen werden.

Wenn man im Test dann die Funktionalität überprüfen möchte, kann unteranderem auch ein Fake-Suchalgorithmus sowie ein Fake-Fileparser angelegt werden. Die Abhängigkeiten wurden also erfolgreich gebrochen.

Weitere Vorgänge, die durchgeführt wurden, sind unteranderem die Auslagerung von GUI-Logik, wie es in den Legacy-Codes-Slides beschrieben ist. Hierfür wurde unteranderem das Package „frontend“ angelegt, welche den größten Teil der GUI-Logik beinhaltet.

Als weitere Refactoring-Maßnahme wurde unbenutzter Code gelöscht. Diese wurde jedoch schon in den Kapiteln SOLID und Tests identifiziert und gelöscht.

Das Vorgehen der Exposed Static Method wurde in der Klasse *ResultDisplayer* verwendet. Die Klasse benötigt keine Instanzvariablen und somit wurde die Methode *displayResult* statisch gemacht.

11. Clean Architecture

Die Clean Architecture wurde nach dem in den Vorlesungen beschriebenen Schichtenmodell versucht zu implementieren. Hierfür könnte der Code in die einzelne Schichte aufgeteilt werden:

Schicht	Klassen / Interfaces
Domain Code	Index, Field, SearchField
Application Code	BreadthFirstSearch, DepthFirstSearch, GreedyFirstSearch, BidirectionalBreadthFirstSearch, BasisSearchAlgorithm, HeuristicSearchAlgorithm, SearchAlgorithm
Adapters	JSONFileParser, JSONFileProcessor, RestRequestProcessor, XMLFileProcessor, FileParser, FileProcessor, RemoteConnectionEtablisher

Die Aufgabe umfasst jedoch nur die Umsetzung einer Klasse in der Adapterschicht. Deshalb wird nur auf diesen Teil genauer eingegangen.

Zunächst wird ein neues Package mit den einzelnen Klassen aus der Adapterschicht angelegt.

Innerhalb einer Adapterschicht werden unteranderem Konvertierungen durchgeführt. Hierfür wurde eine neue Klasse *FileToButtonArrayMap* erstellt. Sie stellt eine eigene Schicht dar, die mit der Application Code Schicht interagiert und eine Konvertierung von einer beliebigen Datei (passender Parser muss da sein) zu einem Button[][] durchführt. Die neue Klasse *FileToButtonArrayMap* kann dann verwendet werden um ein Button[][] aus einer File und dem jeweiligen Parser generieren zu lassen. Der Code wird somit auch einfacher lesbarer. Die Änderungen sind unter der Commit-ID „8d8632ee0aac14d95ab4a500cff66bb398394a9b“ einsehbar. Es ist auch erkennbar, wie die neue Klasse in *VisualizerField* aufgerufen wird.

12. API-Design / Begründung

Innerhalb des Projektes wurde eine neue API auf Programmiersprachenebene angelegt. Diese kann verwendet werden, um allgemein Infos für das aktuell aktive Suchfeld zu erhalten oder das Suchfeld komplett zu manipulieren. Dies war zuvor, sogar nur beschränkt, über das Frontend durch den User möglich. Programmatisch gab es keine Möglichkeit das Feld einfach zu manipulieren oder allgemein Infos über das Feld zu bekommen. Die neue API *FieldHandler* wurde unter der Commit-ID „8dc4d0b7ce6db61b17bfdae8c26536f17e1fc8cc“ gepusht.

In nachstehender Tabelle werden die verfügbaren Methoden sowie eine zugehörige Beschreibung aufgezeigt.

Methode	Beschreibung
FieldHandler(SearchField searchField)	Einziger Konstruktor. Speichert übergebenes Suchfeld in Instanzvariable.
isIndexInField(Index key)	Überprüft ob ein gegebener Index in dem aktuellen Feld vorhanden ist.
getTotalFieldCount()	Gibt die Anzahl der aktuellen Felder im Suchfeld zurück. Hierbei ist egal, ob Felder Start-, Ende- oder Wandfelder sind.
changeAllEmptyFieldsWithWalls()	Ändert alle Felder, welche vom Algorithmus begehbar sind („0“) in eine Wand um („X“).
changeAllWallsWithEmptyFields()	Ändert alle Felder, welche vom Algorithmus nicht betreten werden können („X“) in ein begehbares Feld („0“) um.
getSearchField()	Liefert (manipuliertes) Suchfeld zurück
reInitFieldWithRandomWalls()	Erstellt ein komplett neues Feld und initialisiert zufällig Wände. Größe bleibt gleich zu dem eigentlichen Suchfeld.
getAmountOfWalls()	Liefert die Anzahl an Wände im Suchfeld zurück.
getLocationOfStartFieldOrNull()	Liefert Startfeld als Index zurück. Rückgabe ist ein Optional<Index>, falls Startfeld null ist.

getLocationOfEndFieldOrNull()	Liefert Zielfeld als Index zurück. Rückgabe ist ein Optional<Index>, falls Startfeld null ist.

Des Weiteren gibt es eine weitere nested Class *FieldChanger*. Dieser erlaubt es einzelne Felder zu ändern. Der *FieldChanger* hat eine Methode *changeFieldTo(Index indexToChange, FieldType newType*). Es wurde ein neues Interface *FieldType* angelegt, welches beschreibt um welchen FieldType (Wand, Begehbares Feld) es sich handelt. Das Interface beinhaltet eine Methode *getText*, welche den Text („0“ oder „X“) des jeweiligen Feldes zurückliefert. Mit Hilfe des *FieldChangers* lassen sich einfache Anfragen an die API erstellen. Es wurde ein Fluent Interface erstellt, da dies innerhalb von API-Design als gut angesehen wird.

Nachstehend ist ein Aufruf des Fluent Interfaces zu sehen.

```
Index index1 = new Index( row: 1, column: 2);
Index index2 = new Index( row: 2, column: 3);
fieldChanger.changeFieldTo(index1, new Wall()).changeFieldTo(index2, new EmptyField()).commitChanges();
```

Es können beliebig viele Felder mit ihrem neuen Typ aneinandergehängt werden. Es muss jedoch *commitChanges* aufgerufen werden, bevor die Änderungen dann wirklich durchgeführt werden. Dieses Fluent Interface könnte erweitert werden, durch weitere Methode, die innerhalb der Aufrufe aufgerufen werden können.

12.1. Analyse

Es wurden verschiedene Ansätze aus der Vorlesung übernommen. Die übernommenen Ansätze werden (mit Begründung) in diesem Kapitel aufgezeigt:

Minimal

Die API wurde minimal und sehr schlank gehalten. Die benötigten Methoden sind public und beziehen sich nur auf den eigentlichen Nutzen der API.

Einfach erweiterbar

Es können einfach neue Methoden hinzugefügt werden. Sei es im *FieldHandler* oder *FieldChanger.* Im *FieldChanger* kann auch das Fluent Interface erweitert werden. Hierfür muss nur die jeweilig neue Methode implementiert werden. Die Daten des vorhandenen Interfaces werden bereits in der Instanzvariable gespeichert.

Bennenung

Die Bennenung folgt einem roten Faden und ist einheitlich.

Typisierung

Die Typisierung wurde, wenn möglich durch ein Interface erzielt. Unter „Interfaces“ wird darauf genauer eingegangen.

Optional

Die Methoden *getLocationOfStartFieldOrNull* und *getLocationOfEndFieldOrNull* liefern ein Optional zurück. Der Name der Methode sowie der Rückgabetyp schließen schon darauf, dass der Wert null sein kann. Ein NullObject hätte in diesem Fall nicht viel Sinn ergeben und hätte die Lesbarkeit verringert. Der Nachteil, dass ein Optional nicht gut zu verketten ist, ist zu vernachlässigen, da hier keine Verkettung vorgesehen ist.

Exceptions

Es wurde eine eigene Exception *IndexIsNotInFieldException* angelegt. Diese stellt eine UncheckedException dar. Die Exception wird „geworfen“, wenn ein Index überprüft / geändert werden soll, der aber nicht im eigentlichen Feld ist.

Interfaces

Wie bereits beschrieben wurde ein Interface *FieldType* angelegt, welcher ein Feldtyp beschreibt. Das Interface ist wichtig, damit das **Fluent Interface** funktioniert. Das Fluent Interface stellte keinen deutlich höheren Programmieraufwand dar, aber erhöht deutlich die Lesbarkeit. Der Aufruf ist um einiges einfacher, wie wenn der Benutzer erstmal eine Map befüllen muss um die Felder, welche geändert werden sollen zu übergeben. Der Benutzer kann die einzelnen Felder einzeln mithilfe des Fluent Interfaces übergeben.

Threadsicherheit

Wurde nicht implementiert, aber sollte erwähnt werden, da das Programm keine Parallelität derzeit unterstützt. Es muss also nicht auf Threadsicherheit geachtet werden.

Abschließend gilt zu sagen, dass viele Best-Practices in Bezug auf generelles, wie Benennung, aber auch in Bezug auf technische Aspekte, wie die Verwendung von Optionals zum API-Design eingehalten wurde. Die API bietet einen großen Mehrwert für zukünftige Entwicklung und für den Benutzer.