Programmentwurf
Advanced Software Engineering

Visualisierung von Sortieralgorithmen

von:

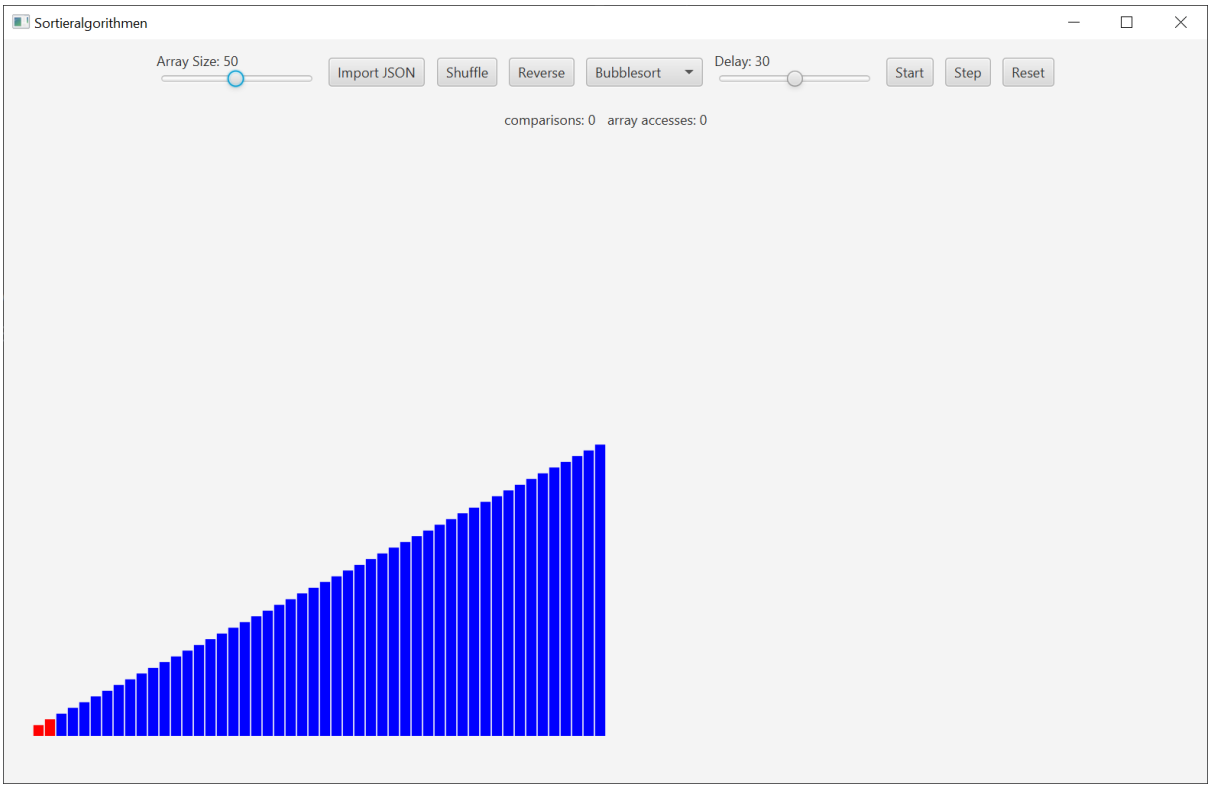
Tim Lincks und Dennis Meier

Matrikelnummern: 5249102, 7509190

Inhaltsverzeichnis

Projektbeschreibung

Die Anwendung, welche die Basis der folgenden Arbeit ist, bietet die Möglichkeit die Funktionsweise von Sortieralgorithmen grafisch darzustellen. Dabei kann zwischen neun verschiedenen Sortieralgorithmen ausgewählt werden. Die Darstellung der Zahlenwerte erfolgt anhand eines Balkendiagramms. Das Sortieren kann schrittweise oder automatisch mit wählbarer Zeitverzögerung ausgeführt werden. Es ist außerdem möglich, mittels einer JSON-Datei selbst Werte anzugeben, die sortiert werden sollen. Auf folgender Abbildung ist die Benutzeroberfläche erkennbar, welche mit JavaFX erstellt wurde.



SOLID

Single Responsible Principle

Das SRP wird zunächst auf die Klasse *Main* angewendet. Diese beinhaltet die Methode *start*, welche die gesamte Benutzeroberfläche initialisiert. Die Benutzeroberfläche wird in die Klassen *UserInputs*, *Results* und *BarGraph* eingeteilt. Auf folgender Abbildung ist zu erkennen, dass der Konstruktor von *UserInputs* die anderen GUI-Elemente als Parameter entgegennimmt und anschließend die Kommunikation zwischen diesen übernimmt. Die Sortieralgorithmen werden ebenfalls übergeben, obwohl diese sonst nicht mehr in der Klasse *Main* verwendet werden.

```java
public class Main extends Application {
    private UserInputs userInputs;

    @Override
    public void start(Stage primaryStage) throws Exception {

        SortingAlgorithm[] sortingAlgorithms = {
                new Bubblesort(),
                new Insertionsort(),
                new Heapsort(),
                new Quicksort(),
                new Shellsort(),
                new Mergesort(),
                new Selectionsort(),
                new Gnomesort()};
        Results results = new Results();
        BarGraph barGraph = new BarGraph();
        UserInputs userInputs = new UserInputs(barGraph, results, sortingAlgorithms);


        VBox vBox = new VBox(userInputs.getElements(), results.getElements(), barGraph.getElement());

        primaryStage.setOnCloseRequest(new EventHandler<WindowEvent>() {
            @Override
            public void handle(WindowEvent e) {
                userInputs.stopThread();
                Platform.exit();
                System.exit( status: 0);
            }
        });

        Scene scene = new Scene(vBox,  width: 1050,  height: 650);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Hello World");
        primaryStage.show();
    }
}
```

Um diesen Sachverhalt zu optimieren, wird die neue Klasse *GuiController* verwendet, welche die Kommunikation zwischen den drei GUI-Elementen übernimmt.

Die Methode *setOnCloseRequest* wird auf die *primaryStage* angewendet und sorgt für die Beendigung des Threads beim schließen des Fensters. Da dies nichts mit der eigentlichen Aufgabe der Methode start zu tun hat, wird die Funktion ausgelagert. Das Resultat dieser Verbesserungen ist auf folgender Abbildung erkennbar.
Commit: `8a86aed79026ebfc2bae1ed3a5572a91e93b2513`

```java
public class Main extends Application {
    GuiController guiController;

    @Override
    public void start(Stage primaryStage) throws Exception{

        guiController = new GuiController();
        VBox vBox = new VBox(guiController.getElements());
        stopThreadOnCloseRequest(primaryStage);
        Scene scene = new Scene(vBox,  width: 1050,  height: 650);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Hello World");
        primaryStage.show();
    }
```

Die Klasse *UserInputs* generiert im Konstruktor alle Eingabefelder und initialisiert die Klassenvariablen. Da dies gegen das SRP spricht, da zu viele Aufgaben in einer Methode erledigt werden, werden alle Initialisierungen von Eingabefeldern in eigene Methoden ausgelagert.
Commit: `6870a330b11743f5d0cdb56a148480c3e2b7c58a`

Die Initialisierung der Eingabefelder beinhaltet auch das Eventhandling. Dabei sind teilweise verschachtelte If-Abfragen enthalten (folgende Abbildung), welche nicht dem eigentlichen Zweck der Methode entsprechen und deshalb ausgelagert werden müssen.

```java
private void initStepButton(BarGraph barGraph, Results results) {
    stepButton = new Button( text: "Step");
    stepButton.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent event) {
            if(sortingAlgorithm.sort()){
                if(autoSort.isRunning()){
                    autoSort.stop();
                }
                stepButton.setDisable(true);
                startButton.setDisable(true);
            }

            barGraph.drawGraph(numberArray);
            results.update(sortingAlgorithm.getComparisons(), sortingAlgorithm.getArrayAccesses());
        }
    });
}
```

Generell wird in der Klasse *UserInputs* wie oben bereits beschrieben die Kommunikation zwischen den GUI-Elementen übernommen, weshalb diese ausgelagert wird. Auch die Klasse *AutoSort*, welche einen Thread für das automatische Sortieren verwaltet, wird dementsprechend angepasst.
Commit: `74d7f0f88926899d98b05b70cbfabcea4229cfdb`

Mergesort:
Das SRP wird auf die Methode *setInitialStates* der Klasse Mergesort angewendet. Der Zweck dieser Klasse ist, die Balken zu den Array-Einträgen zu markieren, die zuerst verglichen werden. Außerdem wird hier aktuell bestimmt, wie viele Durchläufe von Mergesort durchgeführt werden müssen, bis das Array sortiert ist.

```java
@Override
public void setInitialStates() {
    int[] numbers = numberArray.getNumbers();
    int arrayLength = numbers.length;
    NumberState[] numberStates = numberArray.getNumberStates();
    if (numbers[0] < numbers[1]) {
        numberStates[0] = NumberState.NEXTCOMPARISON;
    } else {
        numberStates[1] = NumberState.NEXTCOMPARISON;
    }

    if (arrayLength / 2 == 1 | arrayLength == 4) {
        iterations = 1;
    } else if (arrayLength / 4 == 1 | arrayLength == 8) {
        iterations = 2;
    } else if (arrayLength / 8 == 1 | arrayLength == 16) {
        iterations = 3;
    } else if (arrayLength / 16 == 1 | arrayLength == 32) {
        iterations = 4;
    } else if (arrayLength / 32 == 1 | arrayLength == 64) {
        iterations = 5;
    } else if (arrayLength / 64 == 1 | arrayLength == 128) {
        iterations = 6;
    } else if (arrayLength / 128 == 1) {
        iterations = 7;
    } else {
        System.out.println("Fehler!");
    }
}
```

Das spricht gegen das SRP, weshalb diese Funktion in eine eigene Methode *setMaxIteration* ausgelagert wird. Diese wird dann beim ersten Start der *sort* Methode aufgerufen.

```java
@Override
public boolean sort() {
    if(starter==0){
        setMaxIterations();
        starter++;
    }
    NumberState[] numberStates = numberArray.getNumberStates();
    if (iterationCounter <= iterations) {
        Arrays.fill(numberStates, NumberState.UNDEFINED);
        if (iterationCounter % 2 == 0) {
            forwardSorting();
        } else {
            backwardSorting();
        }
        return false;
    } else {
        Arrays.fill(numberStates, NumberState.FIXED);
        return true;
    }
}

public void setMaxIterations(){
    int[] numbers = numberArray.getNumbers();
    int arrayLength = numbers.length;
    NumberState[] numberStates = numberArray.getNumberStates();

    if (arrayLength / 2 == 1 | arrayLength == 4) {
        iterations = 1;
    } else if (arrayLength / 4 == 1 | arrayLength == 8) {
        iterations = 2;
    } else if (arrayLength / 8 == 1 | arrayLength == 16) {
        iterations = 3;
    } else if (arrayLength / 16 == 1 | arrayLength == 32) {
        iterations = 4;
    } else if (arrayLength / 32 == 1 | arrayLength == 64) {
        iterations = 5;
    } else if (arrayLength / 64 == 1 | arrayLength == 128) {
        iterations = 6;
    } else if (arrayLength / 128 == 1) {
        iterations = 7;
    } else {
        System.out.println("Fehler!");
        Arrays.fill(numberStates, NumberState.NEXTCOMPARISON);
    }
}
```

Commit: `88c34286b2b642a968e3dd6b79d1947b5d467010`

Open Closed Principle

Um die Modularität zu erhöhen wird für die verschiedenen Sortieralgorithmen das Interface *SortingAlgorithm* verwendet. So können neue Sortieralgorithmen implementiert werden, ohne an anderer Stelle den Quellcode anpassen zu müssen (abgesehen von der Initialisierung). Das OCP wird bereits eingehalten.

Liskov Substitution Principle

Wird aufgrund der fehlenden Klassenvererbung nicht angewendet.

Interface Segregation Principle

Kein passender Anwendungsfall vorhanden.

Dependency Inversion Principle

Zur Visualisierung der Sortieralgorithmen wird aktuell ein Balkendiagramm verwendet, welches durch die Klasse *BarGraph* repräsentiert wird. Dadurch hängt die Klasse *GuiController* direkt von *BarGraph* ab, wie in der Abbildung zu sehen ist. Benötigt wird eine Methode um das Diagramm zu zeichnen und eine weitere welche das JavaFX-Knotenelement zurück gibt. Zum aktuellen Stand müssten bei Änderungen der Implementierung des Graphen auch die Klasse *GuiController* mit angepasst werden.

```
public class GuiController {

    private final Results results;
    private final BarGraph barGraph;
    private NumberArray numberArray;
    private SortingAlgorithm chosenSortingAlgorithm;
    private final AutoSort autoSort = new AutoSort( guiController: this);
    private UserInputs userInputs;

    public GuiController(Results results, BarGraph barGraph){
        this.results = results;
        this.barGraph = barGraph;
    }
```

Die Lösung dafür ist das Interface *Graph*, welches die Methoden *drawGraph* und *getNode* beinhaltet. Dadurch können neben einem Balkendiagramm auch weitere Darstellungsarten modular implementiert werden. Im bisherigen Code existieren weitere Klassen die ebenfalls Methoden besitzen, die JavaFX-Knotenelemente zurückgeben. Diese wurden bisher als *getElements* bezeichnet. Da dies nicht eindeutig auf dessen Funktion hindeutet wurden diese im Zuge der Umsetzung des DIP ebenfalls auf angepasst.
Commit: `efa982acbdcfa443e7e372c2837865fdb3862b97`

Refactoring

Im Folgenden werden Code Smells identifiziert aufgezeigt wie diese behoben wurden.

Anonyme innere Klasse:
Bei der Implementierung der Benutzeroberfläche werden viele Eventlistener benötigt, wodurch viele Anonyme Klassen erzeugt wurden (siehe Abbildung).

```
numberSlider.valueProperty().addListener(new ChangeListener<Number>() {
    public void changed(ObservableValue<? extends Number> ov,
                        Number old_val, Number new_val) {
        numberSliderLabel.setText("Array Size: " + new_val.intValue());
        guiController.changeArraySize(new_val.intValue());
    }
});
```

Dadurch entsteht eine unklare Syntax, weshalb es sinnvoll ist dafür Lambda-Ausdrücke zu verwenden. Daher wurden alle anonymen Klassen auf diese Weise ersetzt, wodurch der Code Smell beseitigt werden konnte.
Commit: `d3a3e24a7fbfe78f2997d88a3c198dd188aba449`

```
numberSlider.valueProperty().addListener((ov, old_val, new_val) -> {
    numberSliderLabel.setText("Array Size: " + new_val.intValue());
    guiController.changeArraySize(new_val.intValue());
});
```

Ungenutzte Imports und Variablen:
Zur Steigerung der Übersichtlichkeit und Verständlichkeit wurden alle ungenutzten Imports und Variablen entfernt.
Commit :`a74b4f61baf9a75aadb530243716880ebacd8893`

Verwendung von Raw-Typen:
In der Klasse *UserInputs* wird das JavaFx-Element ChoiceBox verwendet. Dabei handelt es sich um ein GUI-Element welches eine Liste von Variablen beinhaltet. Diese können unterschiedliche Typen annehmen, weshalb von außen unklar ist, was sich darin befindet. Um diesen Code Smell zu beheben wird bei der Variablendeklaration noch mit angegeben, dass in der ChoiceBox nur Strings enthalten sein können.
Commit: `726dfdcfb6c8b3a6fdbc9397df94df5849d8c66f`

Duplicated Code:
In den Methoden *forwardSorting* und *backwardSorting* der Klasse *Mergesort* gab es einige Stellen, die die gleiche Funktion erfüllt haben (finden des kleinsten Elementes in einem Teilarray und das Verschieben dessen). Diese Funktionalitäten wurden in eigene Methoden ausgelagert, auf die bei Bedarf zugegriffen wird.

```
public void findLowestForward(int startValue, int endValue){...}

public void moveLowest(int position, int startOfChunk){...}

public void forwardSorting() {
    int[] numbers = numberArray.getNumbers();
    int arrayLength = numbers.length;
    NumberState[] numberStates = numberArray.getNumberStates();

    if (startOfChunk + sizeOfChunks - counter < arrayLength) {
        findLowestForward(startOfChunk, (startOfChunk+sizeOfChunks-counter));
        moveLowest(position, startOfChunk);

        numberStates[startOfChunk] = NumberState.NEXTCOMPARISON;
        startOfChunk++;
        counter++;
        lowest = maxInt;
```

Commit: `036ecc1ffd132fcd2daa8a6a721dda87a2cbafb2`

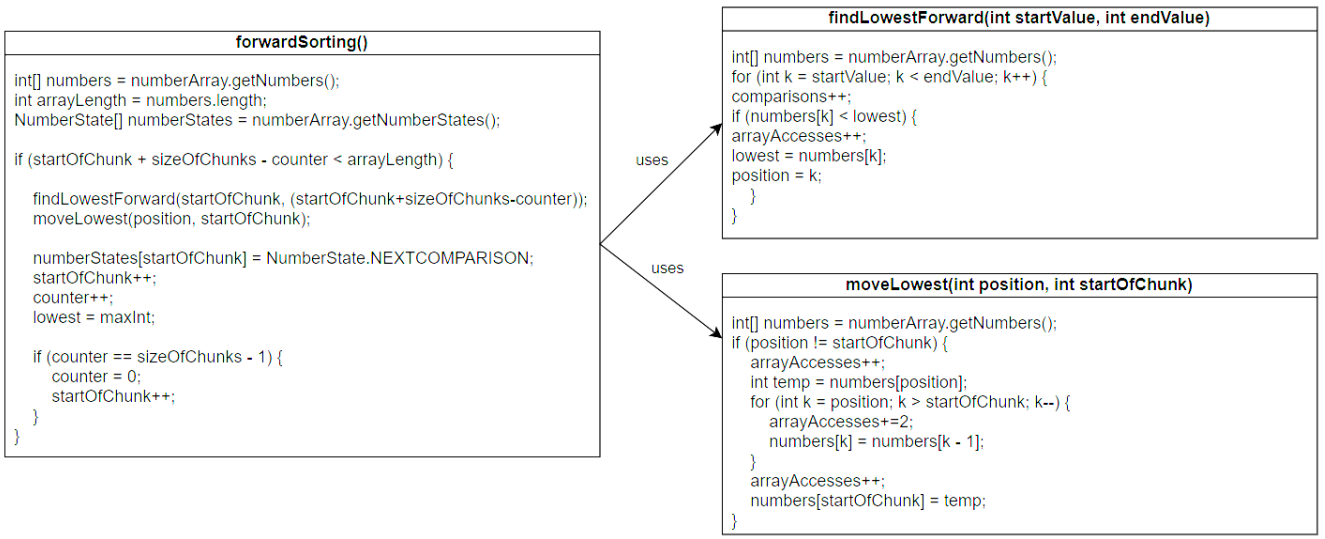
Im nachfolgenden UML-Diagramm ist dargestellt, wie der duplicated Code durch die Verwendung neuer Methoden behoben wurde. Die neuen Methoden werden an mehreren Stellen in der Klasse *Mergesort* verwendet, eine komplette Darstellung als UML wäre an dieser Stelle zu groß, weshalb exemplarisch nur eine Stelle gezeigt wird.

UML vorher:

forwardSorting()

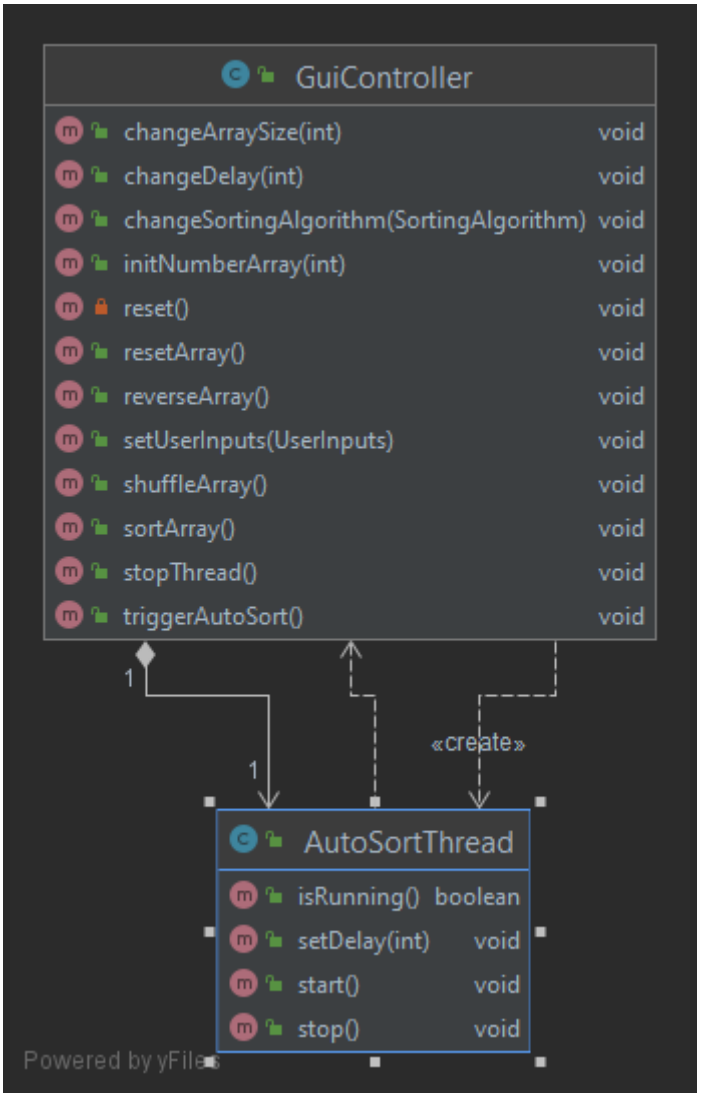
```
int[] numbers = numberArray.getNumbers();
int arrayLength = numbers.length;
NumberState[] numberStates = numberArray.getNumberStates();

if (startOfChunk + sizeOfChunks - counter < arrayLength) {
for (int k = startValue; k < endValue; k++) {
        comparisons++;
        if (numbers[k] < lowest) {
            arrayAccesses++;
            lowest = numbers[k];
            position = k;
        }
    }
    if (position != startOfChunk) {
        arrayAccesses++;
        int temp = numbers[position];
        for (int k = position; k > startOfChunk; k--) {
            arrayAccesses+=2;
            numbers[k] = numbers[k - 1];
        }
        arrayAccesses++;
        numbers[startOfChunk] = temp;
    }
    numberStates[startOfChunk] = NumberState.NEXTCOMPARISON;
    startOfChunk++;
    counter++;
    lowest = maxInt;

    if (counter == sizeOfChunks - 1) {
        counter = 0;
        startOfChunk++;
}
}
```

UML nachher:

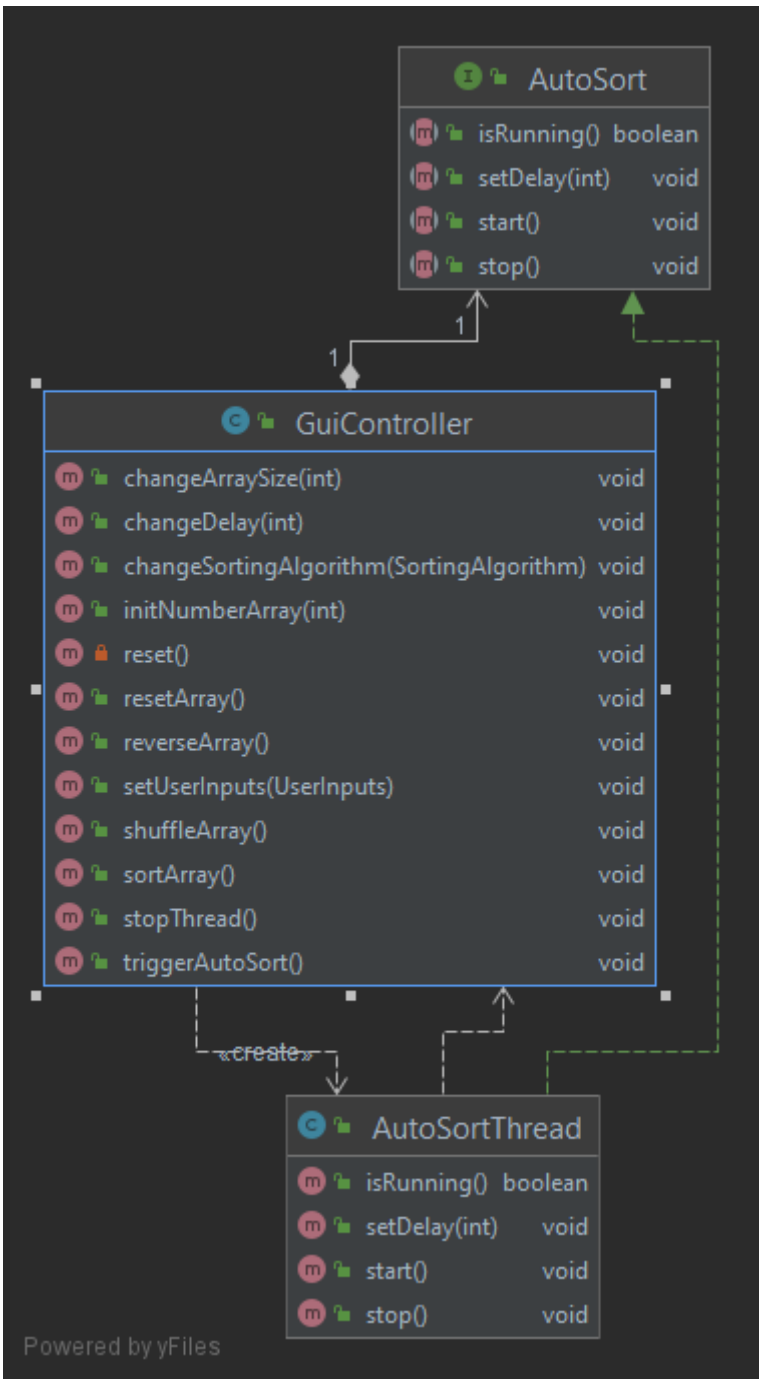


Legacy Code

Die Klasse *GuiController* ruft Methoden der Klasse *AutoSortThread* auf welche das automatische sortieren durch einen Thread ermöglicht. Das Testen von Methoden des *GuiControllers* ist schwierig weil die Klasse *AutosortThread* direkt in den Methoden verwendet wird. Im folgendenen is das zugehörige Klassendiagramm zu sehen.

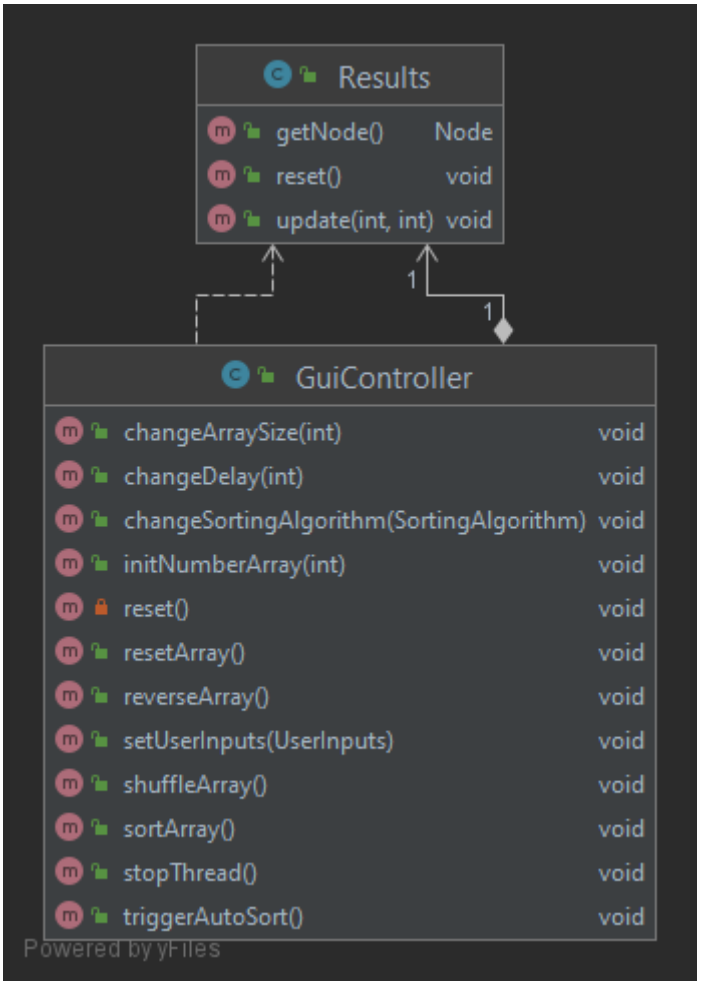


Um diese Abhängigkeit zu brechen, soll die Klasse *AutoSortThread* von außen reingegeben werden. Dafür wird ein Interface verwendet und anstelle der Klasse *AutoSortThread* in den *GuiController* gegeben. Im Folgenden ist das neue Klassendiagramm abgebildet.

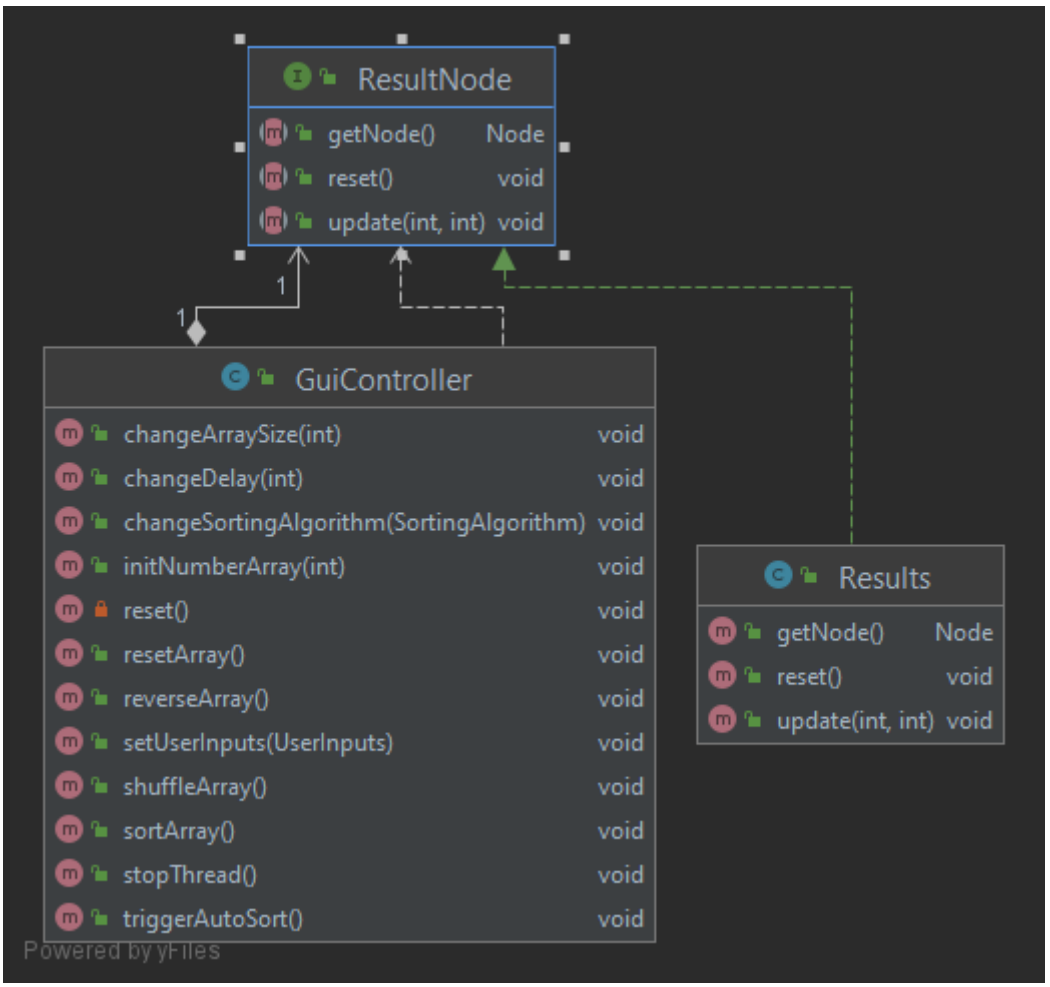


Durch die Verwendung des Interface *AutoSort* wurde die Abhängigkeit gebrochen. Dadurch können Fake- und Mockobjekte zum Testen der Methoden verwendet werden.
Commit: `7b604888a119252ab43c0052484377c9895ceab1`

Die Klasse *GuiController* ruft Methoden der Klasse *Results* auf, welche die Werte für die Anzahl der Vergleich und Array Zugriffe auf der Benutzeroberfläche anzeigt. *Results* bietet keine möglichkeit informationen zurück zu erhalten, weshalb das Testen von Methoden der Klasse *GuiController* erschwert ist. Im Folgenden Klassendiagramm sind die Abhängigkeiten erkennbar.



Zum Brechen der Abhängigkeit wird das Interface *ResultNode* verwendet, welches anstelle der eigentlichen Klasse *Result* im *GuiController* verwendet wird. Im Folgenden ist das neue Klassendiagramm dargestellt.



Es ist klar erkennbar, dass die Abhängigkeit gebrochen wurde, was im weiteren Verlauf das Testen der Methoden erleichtert.
Commit: `ee192006d7810d7883415899d0f462d714579060`

Domain Driven Design

Beispiele für die Ubiquitous Language:

- Comparisons = Anzahl der Vergleiche, die der Suchalgorithmus durchführt
- arrayAccesses = Anzahl der Zugriffe des Suchalgorithmus auf das Array
- gap = Spalt zwischen zwei zu vergleichenden Elementen beim Shellsort
- sizeOfChunks = Größe der Teilarrays, die im aktuellen Schritt untersucht werden
- startOfchunk = Anfangspunkt des aktuellen Teilarrays in Bezug auf das Gesamtarray

Analyse der Value Objects, Entities, Aggregates und Respositories:

Das vorliegende Programm kann als SMART UI angesehen werden, da es im Prinzip nur verschiedene Sortieralgorithmen graphisch darstellt.

Es wurden keine Value Objects verwendet, da es keine unveränderlichen Objekte gibt, die ein Wertekonzept vermitteln.

Es gibt auch keine Entities, da generell keine IDs vergeben werden und keine Repräsentation von Objekte erfolgt.

Da keine Value Objects und Entities vorhanden sind, gibt es auch keine Aggregates und entsprechend keine Repositories.

Unit Tests

Für die Umsetzung der Unit Tests wird die Bibliothek JUnit verwendet. Zusammengefasst konnte eine CodeCoverage von 50% erreicht werden. Viele Methoden, welche Änderungen auf der Benutzeroberfläche auslösen, können nicht getestet werden. Des Weiteren werden Methoden wie in der folgenden Abbildung zu sehen nicht getestet, da die darin enthaltenen Methoden bereits an anderer Stelle getestet werden.

```
public void changeArraySize(int arraySize){
    numberArray.init(arraySize);
    reset();
}
```

Wie bereits im Kapitel “Legacy Code” beschrieben, wurden dort Abhängigkeiten gebrochen um das Testen mit Fake Objekten zu ermöglichen. Diese werden auch verwendet um Methoden der Klasse *GuiController* zu testen. In diesem Zuge wurde auch noch die Abhängigkeit zur Klasse *UserInputs* gebrochen, weshalb nun das Interface *InputNode* hinein gegeben wird.
Das Beispiel in Folgender Abbildung zeigt, wie durch den Einsatz mehrer Fake Objekte die Methode *Reset* der Klasse *GuiController* getestet werden kann.

```
@Test
void testReset(){
    FakeInputNode fakeInputNode = new FakeInputNode();
    FakeResultNode fakeResultNode = new FakeResultNode();
    GuiController guiController = new GuiController(fakeResultNode, new FakeGraph(), new FakeAutoSort());
    guiController.changeSortingAlgorithm(new FakeSortingAlgorithm());
    guiController.setUserInputs(fakeInputNode);
    guiController.reset();
    Assertions.assertTrue(fakeResultNode.isReseted());
    Assertions.assertEquals( expected: "Start", fakeInputNode.getStartButtonText());
    Assertions.assertFalse(fakeInputNode.isStartButtonDisabled());
    Assertions.assertFalse(fakeInputNode.isStepButtonDisabled());
}
```

Commit: `2dde9894b44ad15e9f95fb24fce35493a42383c3`

Entwurfsmuster

Bei dem vorliegenden Projekt wurde das Strategie-Pattern verwendet. *SortingAlgorithm* dient als Inferface für die einzelnen Sortieralgorithmen. Eine abstrakte Klasse, die als Basis für weitere Unterklassen dient, wird nicht benötigt, da die komplette Logik des Programms in den einzelnen Algorithmen steckt.

UML: