Weiterentwicklung der labAlive Anwendung

Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Engineering (B.Eng.)

Bernd Röckert

Betreuer:
Prof. Dr.-Ing. Erwin Riederer

Tag der Abgabe: 30.06.2023

eingereicht bei
Universität der Bundeswehr München
Fakultät für Elektrotechnik und Technische Informatik

Neubiberg, Juni 2023

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, insbesondere keine anderen als die angegebenen Informationen.

Der Speicherung meiner Bachelorarbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.

Neubiberg, den 30.06.2023

Bernd Röckert

Zusammenfassung

Diese Bachelorarbeit befasst sich mit der Weiterentwicklung der Online-Laborumgebung labAlive für kommunikationstechnische Experimente. Im Konkreten soll die Nutzererfahrung durch die Implementierung von zwei neuen Features verbessert werden. Zum einem wird die Umsetzung eines Systems zum automatischen Abrufen und Speichern von Layouts für Simulationen, wie auch die Parametrisierung der Fenster-Positionen zur Speicherung und Wiederherstellung dieser beschrieben.

Inhaltsverzeichnis

Erklärung **III**

1 Einleitung **1**
1.1 Ziel der Arbeit . . . 1
1.2 Aufbau der Arbeit . . . 1

2 Software und Programmiersprachen **3**
2.1 Software . . . 3
2.1.1 Gitea . . . 3
2.1.2 IntelliJ . . . 3
2.1.3 Eclipse . . . 3
2.1.4 Microsoft Access . . . 4
2.2 Programmiersprachen . . . 4
2.2.1 Java . . . 4
2.2.2 HTML . . . 4
2.2.3 SQL . . . 4

3 labAlive **5**
3.1 Website . . . 5
3.2 Anwendung . . . 6
3.3 Layouts . . . 6
3.3.1 Definition . . . 6
3.3.2 Typen . . . 8
3.3.3 Full . . . 8
3.3.4 Compressed . . . 8
3.3.5 Collapsed . . . 9

4 Automatisierte Layout Verwaltung **11**
4.1 Einführung . . . 11
4.2 wiringLayoutProcessor . . . 11
4.2.1 Einstiegspunkt und Parameter . . . 11
4.2.2 Ablauf . . . 12
4.2.3 Layout-Erkennung . . . 13
4.2.4 Transformation . . . 16
4.2.5 Umrechnung . . . 19
4.2.6 Speicherung . . . 20

4.2.7 Layout-Suche . . . 21
4.2.8 User Interaktion . . . 22

5 Fensterpositions-Parametrisierung **25**
5.1 Einführung . . . 25
5.2 Einführung Parameter . . . 25
5.3 Positions-Erkennung und Speicherung . . . 26
5.4 Laden von Positionen . . . 27

6 Fazit **29**
6.1 Ergebnisse . . . 29
6.2 Ausblick . . . 29
6.3 Schlussfolgerung . . . 30

Anhang **I**

Abbildungsverzeichnis **IX**

Tabellenverzeichnis **XI**

Quellcodeverzeichnis **XIII**

Stichwortverzeichnis **XV**

Literaturverzeichnis **XVII**

1 Einleitung

In Zeiten der fortschreitenden Digitalisierung hat sich die einzigartige Möglichkeit eröffnet, eine immense Menge an Wissen, für jedermann zugänglich zu machen. Trotz des Zugangs zu theoretischem Wissen bleibt vielen Menschen jedoch nach wie vor die praktische Weiterbildung verwehrt. Die Beseitigung dieses Mangels stellt eine Herausforderung dar, die durch digitale Labore teilweise bewältigt werden kann. Um die Handhabung dieser Labore auch ohne Ausbildung möglichst zugänglich zu gestalten, ist eine intuitive Bedienung unerlässlich.

1.1 Ziel der Arbeit

Ziel dieser Arbeit ist es, die Benutzererfahrung mit der Anwendung der digitalen Experimentierumgebung *labAlive* [1] zu verbessern. Hierzu soll die Nutzung von Layouts für User zugänglicher gemacht werden. Dies soll durch die Implementierung eines Systems zur automatischen Speicherung sowie zur User-basierten Auswahl von bereits existierenden Layouts erreicht werden. Darüber hinaus soll die Positionierung von Fenstern während der Simulation parametrisiert und auf der serverseitigen Datenbank gespeichert werden. So kann beim erneuten Start der Simulation die Anordnung aller Fenster wiederhergestellt werden, wie sie beim letzten Schließen der Simulation vorlag.

1.2 Aufbau der Arbeit

Die nachfolgende Arbeit beschäftigt sich im Gesamtem mit dem *labAlive*-Projekt. Zunächst wird auf die zur Entwicklung verwendeten Umgebungen eingegangen. Im Anschluss werden Informationen und Grundlagen zu labAlive erläutert, gefolgt von detaillierten Erklärungen zu den behandelten Aufgaben. Abschließend wird alles Kapitel einmal zusammengefasst und ein Ausblick gegeben, gefolgt von einer Schlussfolgerung.

2 Software und Programmiersprachen

Das vorliegende *labAlive*-Projekt, inklusive des Quellcodes sowie aller zugehöriger Pakete und Informationen, wurde von Prof. Dr.-Ing. Erwin Riederer zur Verfügung gestellt. Die bestehende Code-Basis basiert auf der Programmiersprache Java, deren Language-Level auf Java 11 festgelegt ist.

2.1 Software

2.1.1 Gitea

Gitea [2] ist ein Open-Source-Projekt zur Versionsverwaltung über Git. Es bietet eine GitHub ähnliche Umgebung zur Bereitstellung und Verwaltung von Git-Repositories. Neben der Versionsverwaltung werden auch Features wie Bugtracking oder die Erstellung eines eigenen Wikis unterstützt.

Abbildung 2.1: Gitea-Logo [3]

2.1.2 IntelliJ

IntelliJ IDEA [4] ist eine integrierte Java-Entwicklungsumgebung (IDE) welche neben einer intuitiven Benutzeroberfläche Features wie Refactoring, integrierte Git-Unterstützung oder auch JUnit zum Testen bietet. Über Plugins kann zudem der im labAlive genutzte Apache Tomcat Server genutzt werden.
Verwendete Version: IntelliJ IDEA Ultimate 2023.1

Abbildung 2.2: IntelliJ-Logo [5]

2.1.3 Eclipse

Eclipse [6] ist eine weitere integrierte Java-Entwicklungsumgebung (IDE) in welcher das labAlive-Projekt ursprünglich erstellt wurde. Aus Gründen der Kompatibilität musste daher zum Teil auf diese IDE zurückgegriffen werden.
Verwendete Version: Eclipse IDE for Enterprise Java Developers 4.17.0

Abbildung 2.3: Eclipse-Logo [7]

2.1.4 Microsoft Access

Microsoft Access [8] ist ein Datenbankmanagementsystem (DBMS) für die einfache Verwaltung von relationalen Datenbanken. Es bietet Features für die einfache Erstellung von Tabellen und deren Spalten wie auch für entsprechende Abfragen für diese.
Verwendete DB-Version: Microsoft Access Database - Access 2000 Dateiformat

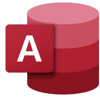

Abbildung 2.4: Access-Logo [9]

2.2 Programmiersprachen

2.2.1 Java

Java [10] ist eine objektorientierte Programmiersprache, welche durch ihre Virtualisierung eine hohe Plattformunabhängigkeit aufweist. Durch ihre breite Unterstützung eignet sie sich gut für die Verwendung als zentrale Programmiersprache in einem Projekt. Sie zeichnet sich zudem durch ihre gute Lesbarkeit und der relativ leichten Syntax aus.
Verwendete SDK: Oracle OpenJDK 17.0.1

Abbildung 2.5: Java-Logo [11]

2.2.2 HTML

HTML (Hypertext Markup Language) [12] ist eine Sprache zur Strukturierung von textbasierten Inhalten in einem Webdokument. Diese HTML-Dokumente werden zur Darstellung von Webseiten im Internet verwendet.

2.2.3 SQL

SQL (Structured Query Language) [13] ist eine Datenbanksprache für die Definition von relationalen Datenbanken. Mit ihr können Anfragen an Datenbanken gestellt werden, sowie Veränderungen an der Struktur und dem Inhalt durchgeführt werden.

3 labAlive

Das Projekt, welches diese Bachelorarbeit behandelt, genannt *labAlive*, ist eine digitale Experimentierumgebung für kommunikationstechnische Experimente welche Versuchsaufbauten ähnlich einer echten Laborumgebung nachbildet. Es bietet eine breite Auswahl an bereits fertigen Experimenten, die direkt gestartet werden können. Zudem können individuelle Experimente mit verschiedenen Systemen gestaltet und Parameter eingestellt werden. Sobald ein Experiment gestartet wird, kann es durch eine Vielzahl von Messgeräten analysiert werden.

3.1 Website

Die Website des *labAlive*-Projekts ist der Einstiegspunkt für jede Simulation. Hier finden sich nicht nur Dokumentationen zu verschiedenen Themen der Kommunikationstechnik, sondern auch die eigentlichen Experimente. Diese können mithilfe von Java Web Start direkt von der Website aus gestartet und lokal mit der entsprechenden Software ausgeführt werden.

Falls kein passendes Experiment gefunden werden kann, oder der Wunsch besteht, ein eigenes Experiment zu erstellen, kann das Tool *myApps* (siehe Abb. [3.1]) verwendet werden. Mit diesem ist es Nutzern möglich, Simulationen in Textform über eine eigene einfache Sprache zu erstellen und somit eine lauffähige Simulation zu erzeugen.

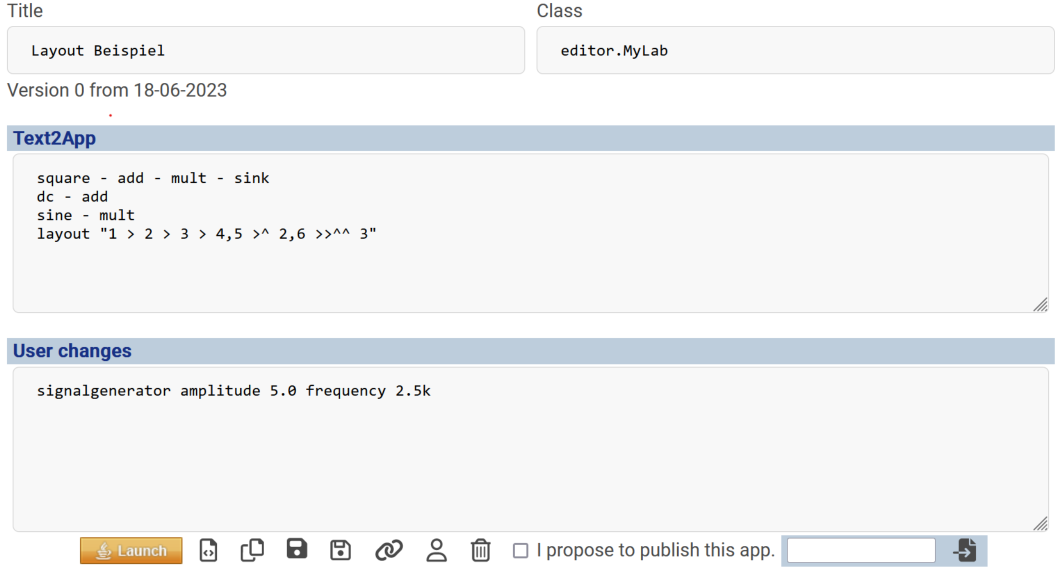

Abbildung 3.1: Eingabemaske myApps

3.2 Anwendung

Wenn eine Simulation mit Hilfe von Java Web Start gestartet wird, öffnet sich nach dem Download einer .jnlp-Datei auf dem lokalen Computer ein Java-Programm, das das entsprechende Experiment darstellt. Eine Simulation besteht aus mehreren Teilen, darunter das Hauptfenster, das die Anordnung der einzelnen Systeme zeigt, sowie verschiedene Messgeräte, die die entsprechenden Messdaten darstellen. Durch einen Rechtsklick auf einen der Übergange zwischen zwei Systemen können neben den initial dargestellten Messgeräten weitere angezeigt werden. Durch einen weiteren Rechtsklick auf ein System können die Parameter in Echtzeit eingestellt werden und von den Messgeräten erkannt und entsprechend angezeigt werden.

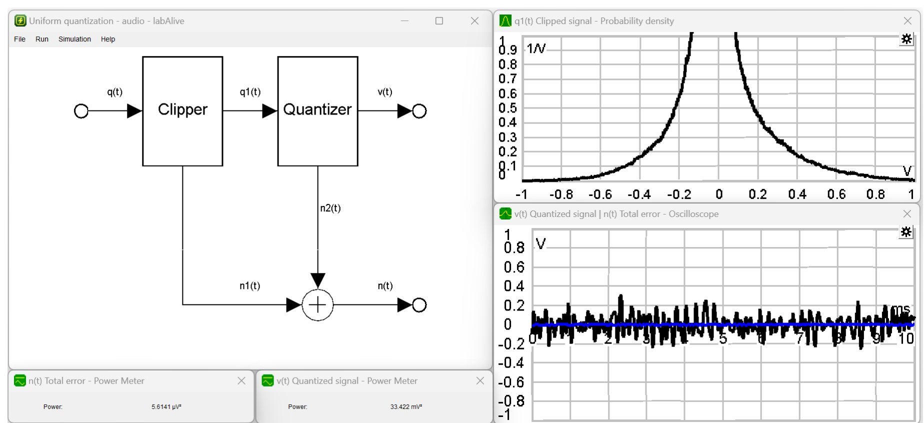

Abbildung 3.2: Beispiel gestartete Simulation

3.3 Layouts

3.3.1 Definition

Ein bereits umgesetztes Feature des Tools *myApps* ist das Erstellen individueller Layouts. Diese erlauben es, Systeme in einer Simulation individuell anzuordnen. Das Layout wird durch zwei Teile bestimmt: den Key und die Richtungsangaben. Der Key (siehe Quellcode [3.1]) gibt die Reihenfolge der Systeme in der Simulation an, die durch ein Minus getrennt werden.

```
1  [Key]:                1 - 2 - 3 - 4 - 5
```

Quellcode 3.1: Beispiel Key

Die Richtungsangaben (siehe Tabelle [3.1]) geben die Ausrichtung der Komponenten zueinander wieder und verbinden dabei zwei oder mehr Systeme miteinander. Angegeben werden sie über die Angabe von Halb- oder Vollschritten. Diese können beliebig miteinander kombiniert werden und geben Richtungsangaben im zweidimensionalen Raum wieder. Die Schritte können dabei jeweils nach oben, unten, links und rechts gehen.

Symbol	Richtung
^	Oben Vollschritt
w	Oben Halbschritt
v	Unten Vollschritt
s	Unten Halbschritt
>	Rechts Vollschritt
d	Rechts Halbschritt
<	Links Vollschritt
a	Links Halbschritt

Tabelle 3.1: Richtungsangaben Layout

Um ein Layout als Java-Objekt zu erstellen (siehe Quellcode [3.2]), müssen Key und Richtungsangaben fusioniert werden. Das erste Element des Objekts ist dabei der Key, welcher die Reihenfolge und die Anzahl der involvierten Systeme angibt. Das zweite Element ist die Kombination aus Key und Richtungsangaben, also das konkrete Layout. Im Verbund können diese Elemente alle Arten von Layouts eindeutig darstellen.

```
1 new Layout("1 - 2","1 > 2")
```

Quellcode 3.2: Beispiel Layout-Objekt

Layouts können auf zwei verschiedene Arten beschrieben werden, die inhaltlich identisch sind (siehe Quellcode [3.3]). In der impliziten Schreibweise werden Richtungsangaben direkt zwischen den Systemen eingefügt, während in der expliziten Schreibweise eine eigene Codezeile benötigt wird, um das Layout zu definieren.

```
1 [Implizit]:          sine > split > sink
2
3 [Explizit]:          sine - split - sink
4                      layout "1 > 2 > 3"
```

Quellcode 3.3: Vergleich Implizite und Explizite Definition

Im Allgemeinen lassen sich einige Regeln zur Definition von Layouts beschreiben:

- Systeme und Richtungsangaben/Minusse müssen durch Leerzeichen voneinander getrennt werden.
- Systeme und Richtungsangaben/Minusse müssen abwechselnd verwendet werden und mit einem System begonnen und beendet werden.
- Systeme dürfen ausschließlich aus alphanumerischen Zeichen bestehen.
- Systeme dürfen nicht allein aus den Symbolen der Richtungsangaben/Minusse bestehen

- Richtungsangaben können beliebig oft kombiniert werden, dürfen jedoch nicht durch Leerzeichen getrennt werden.

3.3.2 Typen

Die Logik hinter den Layouts ermöglicht es, sie in drei verschiedenen Typen zu kategorisieren: *Full*, *Compressed* und *Collapsed*. Dies wird erreicht, indem Teile einer Schaltung komprimiert oder erweitert werden können. Obwohl sich diese Typen inhaltlich nicht unterscheiden, variiert ihre Verwendbarkeit für unterschiedliche Simulationen.

Die Unterscheidung der Typen ermöglicht es, Layouts sehr allgemein zu gestalten und auf eine Vielzahl von Simulationen anzuwenden. Für komplexere Simulationen, auf die kein allgemeineres Layout zutrifft, können auch sehr genaue Layouts erstellt werden.

In den folgenden Kapiteln werden die Unterschiede und Besonderheiten der drei Typen erläutert.

3.3.3 Full

Der *Full*-Typ beschreibt Layouts ohne Komprimierung. Er zeichnet sich dadurch aus, dass die Anzahl an Systemen im Key der Anzahl an Systemen im Layout-Teil entsprechen. Das einfachste *Full*-Layout ist in dem Quellcode [3.2] dargestellt.

Ein komplexeres Beispiel, in welchem die Gleichheit der Anzahl an Systemen im ersten und zweiten Teil zu sehen ist, wird nachfolgend dargestellt (siehe Quellcode [3.4]).

```
1 new Layout("1 - 2 - 3 - 4,5 - 2,6 - 3","1 > 2 > 3 > 4,5 >^ 2,6 >>^^ 3")
```

Quellcode 3.4: Beispiel *Full*-Layout

3.3.4 Compressed

Der *Compressed*-Typ beschreibt einen Typen, welcher die Komprimierung von Systemen zulässt. Es werden hierbei nur Pfade, Quellen, Sinken und Knoten beschrieben. Wie auch beim *Full*-Typ muss die Anzahl an Systemen im Key und im Layout-Teil identisch sein. Der Unterschied zu diesem ist, dass größere Simulationen mit mehreren Systemen durch ein Layout dargestellt werden können. Die Anzahl an Systemen im Layout muss dabei nicht der Anzahl in der Simulation entsprechen.

Nachfolgend ist ein Beispiel aufgeführt (siehe Quellcode [3.5]), welches ein komprimiertes *Compressed*-Layout mit einem unkomprimierten *Full*-Layout vergleicht. Inhaltlich sind beide Layouts identisch, das *Compressed* kann aber für eine beliebig lange Kette von Single Input Single Output Systemen verwendet werden, während das *Full*-Layout nur für einen Key herangezogen werden kann.

```
1 [Full]:          new Layout("1 - 2 - 3 - 4 - 5","1 > 2 > 3 > 4 > 5")
2
3 [Compressed]:    new Layout("1 - 2","1 > 2")
```

Quellcode 3.5: Vergleich *Full*- und *Compressed*-Layout

3.3.5 Collapsed

Der Collapsed-Typ beschreibt einen weiteren Typen, welcher die Komprimierung von Systemen zulässt. Anders als der *Compressed*-Typ können Systeme vor Multiple Inputs oder nach Multiple Outputs eingefügt werden. Zu beachten ist, dass ein System ergänzt werden muss wenn zwei Multiple Inputs oder zwei Multiple Outputs miteinander verknüpft sind. Resultierend daraus muss die Anzahl an Systemen im Key nicht der Anzahl an Systemen im Layout-Teil entsprechen. Die neu hinzugefügten Systeme dürfen allerdings keine bereits vergebenen Nummern aus dem Key verwenden, weshalb neue verwendet werden.

Folgendes Beispiel (siehe Quellcode [3.6]) stellt ein universales Layout für Schaltungen dar, welche zwei Pfade kombinieren. Die Systeme fünf und sechs sind hierbei diese, die bei einem *Collapsed*-Layout eingefügt werden müssen.

```
1 [Collapsed]:     new Layout("1 - 2 - 3,4 - 2","1 > 5 > 2 > 3,4 ^ 6 ^ 2")
```

Quellcode 3.6: Beispiel Collapsed-Layout

4 Automatisierte Layout Verwaltung

4.1 Einführung

Der Hauptteil der vorliegenden Arbeit beschäftigte sich mit der Entwicklung eines Systems, welches die Nutzer-erstellten Layouts erkennen im diese, sofern noch nicht vorhanden, in einer Datenbank zu speichern. Zusätzlich sollte das System mit einem weiteren verknüpft werden, welches dem Nutzer eine Auswahl an passenden Layouts für seine Simulation präsentieren kann. Diese Auswahl basiert auf den zuvor abgespeicherten Layouts in der Datenbank.

Neben den Klassen und Methoden, die für die Umsetzung dieser Systeme benötigt wurden, wurden weitere allgemeine Methoden entwickelt, die auch anderweitig im Rahmen des *labAlive*-Projekts genutzt werden können. Diese werden allerdings in der vorliegenden Arbeit nicht thematisiert, da sie nicht zielführend für die konkrete Fragestellung sind.

4.2 wiringLayoutProcessor

Der *wiringLayoutProcessor* ist die Methode innerhalb der Klasse *LayoutLoader*, die den gesamten Prozess der Speicherung und des Ladens von Layouts startet. Mit dieser Methode werden alle erforderlichen weiteren Methoden aufgerufen, die für die weitere Verarbeitung notwendig sind.

4.2.1 Einstiegspunkt und Parameter

Zur automatischen Abfrage von Layouts wird das System in der Klasse *DispatcherServletWebStart* innerhalb der *doGet*-Methode aufgerufen. Hierbei wird die benötigte Simulationsbeschreibung, genannt *Wiring*, im *HTTP-GET-Request* übermittelt, welcher vom *labAlive*-Webserver empfangen wird, sobald ein Nutzer eine Simulation über das *myApps*-Tool startet. In dieser Methode, welche Requests verarbeitet, können eingehende Requests kopiert und als *HttpServletRequest*-Parameter an die *wiringLayoutProcessor*-Methode übergeben werden, welche ihn weiterverarbeitet.

Ein weiterer möglicher Übergabeparameter kann von einem Nutzer in Form einer Auswahl eines Layouts getroffen werden. Dieses wird dann als *LayoutRessource* an die entsprechende Methode übergeben, welche die Auswahl weiterbearbeitet.

4.2.2 Ablauf

Der Ablauf der Methode (siehe Abb. [4.1]) sieht vor, dass der übergebene *HttpServletRequest* zuerst disassembliert und der Parameter *w* extrahiert wird. Die restlichen Parameter, welche beispielsweise die Nutzerinformationen enthalten, werden nicht benötigt, da in den nächsten Schritten kein Personenbezug zu den Daten besteht. Der Parameter *w* steht hierbei für die Kurzform von *Wiring*. Das *Wiring* enthält hierbei den in dem *myApps*-Tool geschrieben Code zum Starten einer Simulation.

Nachdem das *Wiring* extrahiert wurde, muss dieses vereinheitlicht werden. Zu diesem Zweck wird die *unifyString*-Methode aufgerufen, welche mit Hilfe eines Regex-Patterns bzw. Muster alle Mehrfachvorkommen von Leerzeichen erkennt und jeweils durch ein einfaches Leerzeichen ersetzt. Anschließend werden, falls vorhanden, Leerzeichen, die das *Wiring* beginnen bzw. beenden, entfernt. Zweck dieser Vereinheitlichung ist, dass jedes *Wiring* unter denselben Voraussetzungen analysiert, und ein Layout immer im selben Format abgespeichert werden kann. Würde dies nicht geschehen, besteht die Gefahr, dass ein vermeintlich neues Layout wegen einem zusätzlichen Zeichen abgespeichert wird, obwohl dieses Layout bereits in der Datenbank vorliegt.

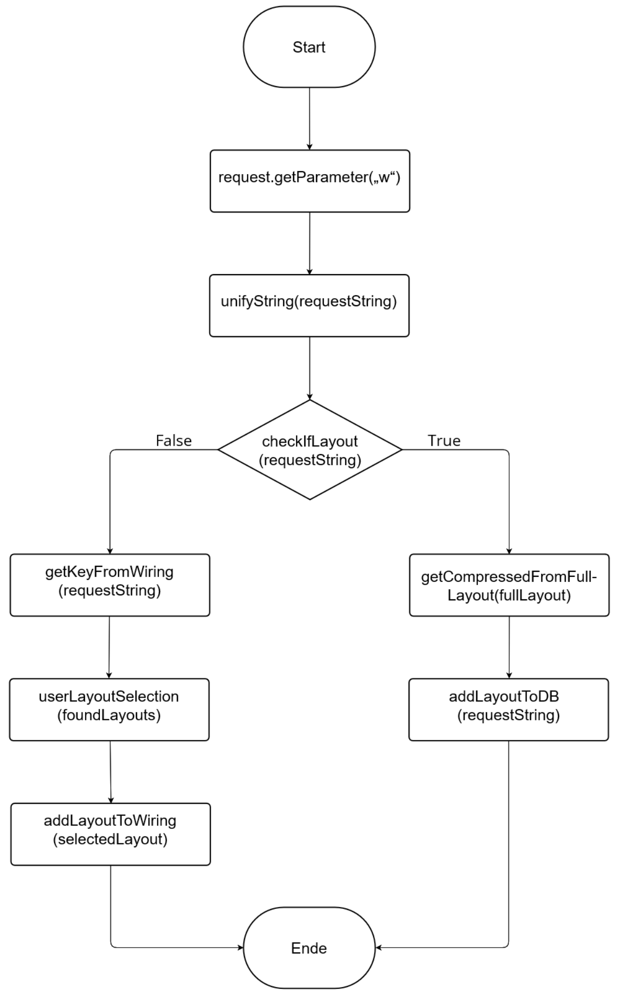

Abbildung 4.1: Flussdiagramm *wiringLayoutProcessor*

Um den weiteren Ablauf festzulegen, wird das *Wiring* nach einer Layout-Definition mithilfe von Regex-Pattern durchsucht. Sollte eine solche gefunden werden, wird das gefundene Layout in die Form eines Layouts-Objekts gebracht. Anschließend wird dieses Layout, sofern es als *Full*-Typ vorliegt, versucht in ein *Compressed*-Typen umzurechnen. Nach der Umrechnung werden beide Varianten in der Datenbank abgespeichert.

Sollte keine Layout-Definition gefunden worden sein, wird versucht den Key des *Wirings* zu generieren. Dieser Key wird anschließend dazu verwendet, alle passenden Layouts für diesen aus der Datenbank zu erhalten. Die Auswahl an Layouts wird dem Nutzer zur Auswahl gestellt. Nachdem durch eine Interaktion ein solches ausgewählt wurde, wird dieses dem ursprünglichen *Wiring* hinzugefügt. Dieses modifizierte *Wiring* wird dann von der *wiringLayoutProcessor*-Methode zurückgegeben, wodurch die Generierung des eigentlichen *.jnlp*-Files fortgesetzt werden kann.

4.2.3 Layout-Erkennung

Die Erkennung eines Layouts lässt sich mithilfe von Regex-Patterns realisieren. Regex basiert auf regulären Ausdrücken, welche eine definierte Zeichenkette, in einer eigenen Syntax beschrieben, erkennen können. Mit solchen regulären Ausdrücken lassen sich also Pattern beschreiben, welche eine Layout-Definition darstellen, mit denen anschließend gesucht werden können. Bei der Erkennung muss beachtet werden, dass es unterschiedliche Schreibweisen gibt, die es zu erkennen gilt. Konkret muss neben der impliziten Schreibweise, die explizite Schreibweise des *Full*-, *Compressed*- und *Collapsed*-Typ, gekennzeichnet durch unterschiedliche Schlüsselwörter (siehe Quellcode [4.1]), erkannt werden.

```
1  [Full]:             sine - sink
2                      layout "1 > 2"
3
4  [Compressed]:       sine - sink
5                      layoutCompressed "1 > 2"
6
7  [Collapsed]:        sine - sink
8                      layoutCollapsed "1 > 2"
```

Quellcode 4.1: Syntax der Layout-Typen

Es wäre theoretisch möglich, ein großes Pattern zu erstellen, welches alle möglichen Definitionen abdeckt. Allerdings wäre dies sehr unübersichtlich. Stattdessen wurden mehrere Patterns erstellt, die jeweils einen Teil der Definitionen erkennen können. Dabei gibt es ein Pattern für die implizite Schreibweise (siehe Quellcode [4.3]), da hier keine Unterscheidung zwischen Typen notwendig ist. Für die explizite Schreibweise, bei der sich die Syntax der Typen unterscheidet, wurde für jeden Typen ein eigenes Pattern erstellt (siehe Quellcode [4.4]).

```
1 [Full-Configured]:  sine 5kHz > lowpass > sink
```

Quellcode 4.2: *Full*-Layout mit Einstellungen

Die Erkennung eines impliziten Layouts (siehe Abb. [4.2]), erfordert nur zwei Systeme, die durch eine Richtung verbunden sind. Anschließend können unendlich viele weitere Systeme durch eine Richtung angeschlossen werden. Ebenfalls ist es möglich, nach einer Systemdefinition in der impliziten Schreibweise, diese mit Werten zu konfigurieren (siehe Quellcode [4.2]). Obwohl diese Werte erkannt werden müssen, sind sie für das eigentliche Layout irrelevant. Zusätzlich kann sich eine implizite Definition über mehrere Zeilen erstrecken, was für die Prüfung unerheblich ist, da jede Zeile einzeln geprüft werden kann, was die Komplexität reduziert.

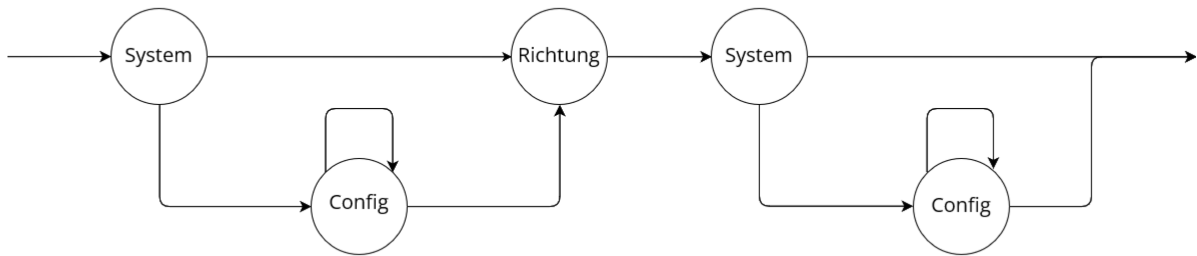

Abbildung 4.2: Syntaxgraph implizites Layout (ohne Leerzeichen)

Ein erstes Problem ergibt sich in der praktischen Umsetzung bei der Erkennung der einzelnen Ausdrücke. So erkennt man den Ausdruck System bzw. Einstellung über eine wiederholte Aneinanderreihung von alphanummerischen Zeichen. Durch die Erweiterung der Einstellungen, können solche Ausdrücke beliebig oft hintereinander gehängt werden. Möchte man allerdings wiederholt diesen Ausdruck, welcher Leerzeichen-getrennt auftreten darf, erkennen, führt die *greedy*-Eigenschaft zur Herausforderung. Diese Eigenschaft bewirkt, dass das Muster versucht, einen Ausdruck so oft wie möglich hintereinander zu erkennen. Dies wiederrum kann dazu führen, dass eine Richtungsangabe, welche auch nur aus Buchstaben bestehen kann, als System bzw. Einstellung erkannt wird.

Um sicherzustellen, dass eine solche Verwechslung nicht auftritt, kann sich weiterer Mittel von Regex bedient werden: den *Look-around assertions*. Unter *Look-around assertions* versteht man kontextabhängige Bedingungen, welche dazu verwendet werden können, einen Ausdruck unter der Bedingung eines voraus- oder nachfolgenden Ausdrucks zu erkennen. Diese können zur Erkennung einer Richtungsangabe nach einem System eingesetzt werden. Ist dies der Fall wird dieser Ausdruck nicht weiter erkannt und der nächste Ausdruck wird getestet. Der darauffolgende Ausdruck stellt dann das letzte System/ die letzte Einstellung vor der Richtungsangabe dar.

Ein weiteres Problem besteht darin, das Richtungsangaben als Teil-String in Systemen/Einstellungen erkannt werden könnten. Um zu vermeiden das Teilstrings erkannt werden, werden Richtungsangaben nur erkannt, wenn sie auf Leerzeichen enden. Wie auch bei dem vorherigen Problem lässt sich dies über eine *Look-around assertion* lösen, welche prüft, ob die Richtungsangabe auf ein oder mehr Leerzeichen

endet.

Das vollständige Regex-Pattern (siehe Quellcode [4.3]) lässt sich wie folgt, unter Verzicht auf die Leerzeichen, beschreiben: Ein System/Einstellung kann optional unendlich oft erkannt werden, solange danach keine Richtungsangabe steht. Wenn eine Richtungsangabe erkannt wird, wird noch einmalig versucht ein System/Richtung zu erkennen die auf eine Richtung endet. Anschließend die Richtung selbst. Dieser Ablauf wird versucht möglichst oft hintereinander erkannt zu werden. Sollte dies nicht mehr der Fall sein, wird optional versucht Systeme/Einstellungen zu finden die, auf das letzte System/ die letzte Einstellung endet.

```
1 Pattern implicitLayoutPattern = Pattern.compile("\\h*(([a-zA-Z0-9]+\\h↩
      +?(?![wasdv^<>]+))*([a-zA-Z0-9]+\\h+?(?=[wasdv^<>]+))+[wasdv↩
      ^<>]+(?=\\h)\\h+)+([a-zA-Z0-9]+\\h+?(?![wasdv^<>]+))*[a-zA-Z0-9]+\\h↩
      *");
```

Quellcode 4.3: Pattern zur Erkennung eines impliziten Layouts

Ausdruck	Regex
System/Einstellung	[a-zA-Z0-9]+
Horizontale Leerzeichen/Whitespaces	\\h
Richtung	[wasdvˆ<>]
Richtungsangaben	[wasdvˆ<>]+(?=\\h)
Nicht auf Richtung endendes System/Einstellung	[a-zA-Z0-9]+\\h+?(?![wasdvˆ<>]+)
Auf Richtung endendes System/Einstellung	[a-zA-Z0-9]+\\h+?(?=[wasdvˆ<>]+)

Tabelle 4.1: Regex-Pattern zur Layout-Erkennung

Die Erkennung eines Layouts in der expliziten Schreibweise (siehe Abb. [4.3]) ist, aufgrund der anderen Syntax, weniger komplex in der Umsetzung. Um die einzelnen Typen eindeutig voneinander unterscheiden zu können, wurde für jeden Typen ein eigenes Pattern erstellt. Diese unterscheiden sich allerdings nur in den initialen Schlüsselwörtern, sowie darin, dass Systeme keine nur noch numerische und keine alphanumerischen Namen besitzen. Über diese Unterschiede hinaus sind sie sonst identisch.

Der Ablauf der Erkennung beginnt mit einem der drei Schlüsselwörter gefolgt von einem Anführungszeichen. Danach beginnt die Beschreibung des eigentlichen Layouts, welches minimal aus einer Zahl, welche mit einer anderen Zahl über eine Richtungsangabe verbunden ist, besteht. Ein Unterschied zur impliziten Schreibweise besteht in der Möglichkeit, Kommata einzuführen. Diese können nach der ersten minimalen Kette angefügt werden, wonach wiederum mindestens eine minimale Kette folgen muss. Diese Erweiterung um weitere Ketten kann beliebig oft erfolgen. Beendet wird das Layout über ein Anführungszeichen.

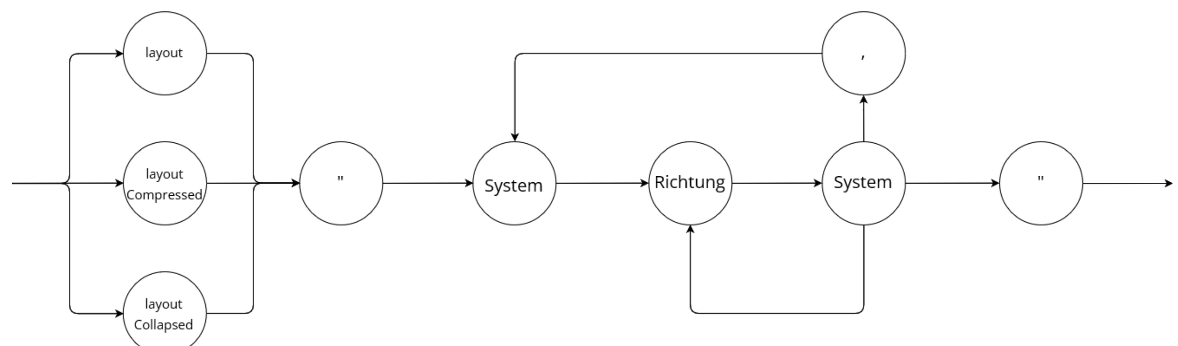

Abbildung 4.3: Syntaxgraph explizites Layout (ohne Leerzeichen)

Das vollständige Regex-Pattern (siehe Quellcode [4.4]), unter Verzicht der Leerzeichen, lässt sich wie folgt beschreiben: eines der drei Schlüsselwörter muss gefolgt von einem Leerzeichen einmalig erkannt werden. Anschließend muss ein System in Form einer Zahl folgen. Nach der initialen Zahl muss eine Richtungsangabe, sowie ein weiteres System folgen. Dieser Vorgang kann beliebig oft wiederholt werden. Diese Kette kann getrennt durch ein Komma beliebig oft wiederholt werden. Abschließend folgt ein Anführungszeichen.

```
1 Pattern explicitFullLayoutPattern          = Pattern.compile
2 ("(\\h)*layout(\\h)*(\")(\\h)*[0-9]+(((\\h)*[wasdv^<>]+(\\h)*[0-9]+)
    |((\\h)*(,)(\\h)*[0-9]+))+(\\h)*(\")(\\h)*");
3 Pattern explicitCompressedLayoutPattern    = Pattern.compile
4 ("(\\h)*layoutCompressed(\\h)*(\")(\\h)*[0-9]+(((\\h)*[wasdv^<>]+(\\h)*[
    0-9]+)|((\\h)*(,)(\\h)*[0-9]+))+(\\h)*(\")(\\h)*");
5 Pattern explicitCollapsedLayoutPattern     = Pattern.compile
6 ("(\\h)*layoutCollapsed(\\h)*(\")(\\h)*[0-9]+(((\\h)*[wasdv^<>]+(\\h)*[0
    -9]+)|((\\h)*(,)(\\h)*[0-9]+))+(\\h)*(\")(\\h)*");
```

Quellcode 4.4: Pattern zur Erkennung eines expliziten Layouts

4.2.4 Transformation

Eine Vereinfachung des Programmcodes wird dadurch erreicht, dass intern nur mit einer Layout-Schreibweise, der expliziten, gearbeitet wird. Diese hat den Vorteil, dass die Layout-Zeile am Ende des *Wiring* alle relevanten Informationen zur Erstellung eines Layout-Objektes enthält. Zur einheitlichen Nutzung der expliziten Darstellung muss daher die implizite in diese umgewandelt werden können.

Die Transformation von implizite in explizite Schreibweise geschieht in zwei Schritten: In einem ersten Schritt wird der Key aus der impliziten Schreibweise extrahiert, im zweiten Schritt die eigentliche Layout-definition. Zum Schluss werden die beiden Ergebnisse der Schritte kombiniert, um ein explizites Layout zu erhalten.

Im ersten Schritt, der der Berechnung des Key dient (siehe Quellcode [4.5]), wird der eingehende String in impliziter Schreibweise in Zeilen aufgeteilt. Dies kann genutzt werden, um mit niedrigerer Komplexität die Zeilen einzeln, anstatt alle als Ganzes, zu verarbeiten. Durch eine Iteration über alle Zeilen wird im Anschluss geprüft, ob es sich um eine Zeile handelt, die einen Teil eines impliziten Layouts enthält. Ist dies der Fall, werden in der entsprechenden Zeile alle Richtungsangaben über ein entsprechendes Regex-Muster (siehe Tabelle [4.1]) durch ein Minus ersetzt. Nachdem über alle Zeilen iteriert wurde, werden die Zeilen ohne Richtungsangaben wieder zusammengesetzt. Zurückgegeben wird schlussendlich der String, der die neuen Zeilen enthalten. Dieser stellt nun den Key des Layouts dar.

```
1  public static String transformImplicitToMinusSeperated(String ↩
       requestString) {
2      String[] lines = getLines(requestString);
3      String[] linesMinusSeperated = new String[lines.length];
4      for (int i = 0; i < lines.length; i++) {
5          if (isImplicit(lines[i])) {
6              linesMinusSeperated[i] = lines[i].replaceAll("[wasdv↩
                   <>^]+(?=\\h)", "-");
7              continue;
8          }
9          linesMinusSeperated[i] = lines[i];
10     }
11     return String.join("\n", linesMinusSeperated);
12 }
```

Quellcode 4.5: transformImplicitToMinusSeperated.java

Der zweite Schritt (siehe Quellcode [4.6]) besteht darin, Systeme zu erfassen und ihnen eine entsprechende nummerische Zahl zuzuordnen. Wie auch im ersten Schritt wird hier zeilenweise gearbeitet. Um die bereits erfassten Systeme über die Iteration hinaus zu erfassen, wird hierzu eine globale Hash-Map benötigt, welche als Key das System sowie als Value die neue System-Nummer enthält. Über dies hinaus wird eine globale Zählvariabel benötigt, sodass System-Nummern immer aufsteigend und nie doppelt vergeben werden.

```
1  public static String transformImplicitToSystemToNumbers(String ↩
       requestString) {
2      String[] lines = getLines(requestString);
3      List<String> systemAsNumbers = new ArrayList<>();
4      systemCounterForHashMap = 0;
5      for (String line : lines) {
6          if (isImplicit(line)) {
7              systemAsNumbers.add(systemToNumbers(line));
8          }
9      }
```

```
10      return systemAsNumbers.stream().map(Object::toString).collect(↩
            Collectors.joining(","));
11  }
```

Quellcode 4.6: transformImplicitToSystemToNumbers.java

Der eigentliche Ersetzungsprozess (siehe Quellcode [4.7]) beginnt damit, dass die übergebene Zeile in Worte aufgespalten wird. Bei jedem dieser Worte wird anschließend geprüft, ob es sich um ein System handelt. Wenn dies der Fall ist, wird überprüft, ob der entsprechende Key bereits in der Hash-Map vorhanden ist. Trifft dies zu, wird der Value des Keys abgefragt und in eine String-Liste gespeichert. Sollte er noch nicht in der Hash-Map vorhanden sein, wird er mit seinem Value abgelegt und der Wert in die String-Liste aufgenommen. Wenn es sich bei dem Wort nicht um ein System handelt, wird stattdessen überprüft, ob es sich um eine Richtung handelt. Handelt es sich dabei um eine Richtung, wird sie unverändert in die String-Liste eingefügt. Nachdem alle Worte auf diese Weise bearbeitet wurden, werden sie wieder zu einer Zeile zusammengefügt und als transformierte Zeile zurückgegeben. Wenn alle Zeilen transformiert wurden, werden diese durch Kommata getrennt wieder zusammengefügt. Zu diesem Zeitpunkt ist die Transformation abgeschlossen und der modifizierte Layout-String kann zurückgegeben werden.

```
1   public static String systemToNumbers(String line) {
2       String[] words = line.split("[!\\h+]");
3       List<String> systemAsNumbers = new ArrayList<>();
4       for (int i = 0; i < words.length; i++) {
5           if (isSystem(i, words)) {
6                   f (map.containsKey(words[i])) {
7                   systemAsNumbers.add(Integer.toString(map.get(words[i])))↩
                        ;
8                   continue;
9               }
10              map.put(words[i], ++systemCounterForHashMap);
11              systemAsNumbers.add(Integer.toString(map.get(words[i])));
12              continue;
13          }
14          if (isDirection(i, words)) {
15              systemAsNumbers.add(words[i]);
16          }
17      }
18      return systemAsNumbers.stream().map(Object::toString).collect(↩
            Collectors.joining(" "));
19  }
```

Quellcode 4.7: systemToNumbers.java

Um sicherzustellen, dass es sich bei einem Layout um eines in der expliziten Schreibweise handelt, muss abschließend nur noch die entsprechende Transformations-Methode aufgerufen werden, welche unabhängig vom übergebenen Typen einen Layout-String in expliziter Schreibweise zurückgibt.

```java
1 public static String transformToExplicitLayout(String requestString) {
2     if (isExplicit(requestString)) return requestString;
3     return transformImplicitToExplicitLayout(requestString);
4 }
```

Quellcode 4.8: transformToExplicitLayout.java

4.2.5 Umrechnung

Zur Verbesserung der Nutzererfahrung sollte eine große Auswahl an Layouts zur Verfügung stehen. Eine Möglichkeit, dies zu erreichen, besteht darin, Layouts in verschiedenen Typen zu speichern. Konkret können Layouts im *Full*-Typ genommen und in ein komprimiertes *Compressed*-Layout umgewandelt werden, bevor beide Varianten in der Datenbank gespeichert werden.

Um ein *Full*-Layout in ein *Compressed*-Layout umwandeln zu können, müssen die Systeme, die Eingänge (Signalquellen), Ausgänge (Signalsinken) sowie Knoten wie Multi Input Multi Output Systeme (MIMO) repräsentieren, identifiziert werden. Zudem müssen die Single Input Single Output Systeme (SISO) ermittelt werden. Da im *Compressed*-Layout nur Eingänge, Ausgänge und Knoten relevant sind, können nach der Ermittlung aller Systeme entsprechend alle SISO-Systeme und ihre damit verbundenen Richtungsangaben entfernt werden.

Die Vorgehensweise zur Systemerkennung erfolgt zunächst anhand der Position. Wenn sich ein System am Anfang oder Ende einer Zeile befindet, handelt es sich sicher um einen Eingang, Ausgang oder Knoten. Diese müssen in jedem Fall beibehalten werden. Auf diese Weise können alle Eingänge und Ausgänge bestimmt werden. Um sicherzustellen, dass alle Knoten erkannt werden (siehe Quellcode [4.9]), werden alle Zeichenfolgen einzeln untersucht und geprüft, ob sie im entsprechenden Key bzw. Layout zweimal vorkommen. Wenn dies der Fall ist, handelt es sich um einen Knoten, der ebenfalls nicht gelöscht werden darf.

```java
1 public static boolean systemIsMIMO(String system, String keyOrLayout) {
2     String[] words = keyOrLayout.split("[!\\h+]|,");
3     int count = 0;
4     for(String word: words) {
5         if(word.equals(system)) count++;
6     }
7     return count > 1;
8 }
```

Quellcode 4.9: systemIsMIMO.java

Sollte bei der Prüfung ein SISO gefunden werden, wird dieses sowie die Richtungsangabe, die sich zu seiner linken Seite befindet, gelöscht. Am Ende der Prüfung können dann alle Ein- und Ausgänge

sowie Knoten zum neuen *Compressed*-Layout zusammengesetzt werden. Mit der abschließenden Veränderung des Typen-Feldes des Layouts auf *Compressed* ist die Umrechnung abgeschlossen.

4.2.6 Speicherung

Um die erfassten Layouts zu speichern, konnte auf eine bereits passende Datenbank zurückgegriffen werden, welche eine eigene Tabelle nur für Layouts beinhaltet. Neben einer eindeutigen und automatisiert vergebenen *LayoutID* beinhaltet diese ein Feld für den Typ eines Layouts, also *Full*, *Compressed* oder *Collapsed*, eines für den Key, die Richtungsangaben, für einen individuellen Titel sowie für eine Beschreibung des Layouts.

In der festgelegten Datenhierarchie des *labAlive*-Projektes ist festgelegt, dass ein Layout zur Oberklasse der Ressourcen gehört. Dies heißt, dass jedes Layout eine Art von Ressource aber nicht jede Ressource, ist ein Layout.

Der Zugriff auf Ressourcen wird über die Klasse *ResourceProvider* geregelt. Diese Klasse wiederum bietet die Methode *addResource* an, mit welcher eine *Resource*, in diesem Fall ein Layout, in die Datenbank gespeichert werden kann. Um diese Methode entsprechend nutzen zu können, muss ein Resource-Objekt des Typen *LayoutResource* initialisiert werden. Hierzu wird die Methode *extractLayout* (siehe Quellcode [4.10]) der Klasse *LayoutToDB* aufgerufen. Zu Beginn wird in dieser ein neues Objekt vom Typ *LayoutResource* erstellt. Nach der Initialisierung kann anschließend mit den entsprechenden Befehlen das Objekt bearbeitet werden. Zu setzen ist die *LayoutID* auf -1, da die Layouts abgegriffen werden und nicht mit einem Nutzer verknüpft werden müssen. Die Version wird auf den Wert 100 gesetzt. Ferner ist der Key und das Layout mit den entsprechend erfassten Werten zu setzen. Der Titel wird immer auf den String *Auto generated layout from Text2App* gesetzt, die Beschreibung auf den String *Query-String:* kombiniert mit dem ursprünglich erfassten String aus dem als Compressed abgespeicherten Request.

```
1  public static LayoutResource extractLayout(String explicitLayout, String↩
       requestString) {
2      if (explicitLayout == null || requestString == null) return null;
3      LayoutResource layout2BeCreated = new LayoutResource();
4      requestString = requestString.replaceAll("\n", ",");
5      layout2BeCreated.setId(-1);
6      layout2BeCreated.setType(0);
7      layout2BeCreated.setKey(getKeyFromExplicitLayout(explicitLayout));
8      layout2BeCreated.setVersion(100);
9      layout2BeCreated.setLayout(getLayoutLineFromExplicitLayout(↩
          explicitLayout));
10     layout2BeCreated.setTitle("Auto generated layout from Text2App");
11     layout2BeCreated.setDescription("Query-String: " + requestString);
12     return layout2BeCreated;
```

```
13 }
```

Quellcode 4.10: extractLayout.java

Zur Erhöhung des Datenbestands an unterschiedlichen Layouts wird zusätzlich, falls es sich um ein Layout des *Full*-Typen handelt, dieser als *Compressed* abgespeichert.

4.2.7 Layout-Suche

Wenn ein Nutzer ein *Wiring* starten möchte, welches noch kein Layout enthält, wird der Methodenteil des Layout-Ladens gestartet. In diesem wird zuerst versucht den Key aus dem gegebenen *Wiring* zu extrahieren. Der Key wird benötigt, um darauf basierend die Datenbank nach passenden Layouts zu diesem spezifischen Key zu durchsuchen. Um diesen zu extrahiert wird eine modifizierte Variante der *transformImplicitToSystemToNumbers*-Methode (siehe Quellcode [4.6]) genutzt, genannt *getKeyFromWiring*. Diese ist eine Kopie der anderen Methode. Sie unterscheiden sich lediglich darin, dass in der for-Schleife, welche über die Zeilen iteriert die Prüfung entfällt, ob es sich um eine implizite Zeile handelt. Durch das Weglassen der Prüfung wird erreicht, dass die Systeme der Zeile immer zu einer Nummer umgewandelt werde. Zu diesem Zeitpunkt der Methode wurde bereits sichergestellt, dass kein Layout vorliegt. Das Ergebnis dieser Methode ist schlussendlich der Key des vorliegenden *Wirings*.

Der Key des *Wirings* ermöglicht es die Suche (siehe Quellcode [4.11]) nach einem passenden Layout zum vorliegenden *Wiring* zu starten. Wie auch bei der Speicherung von Layouts, ist auch bei der Suche der Zugriff auf die Datenbank über die *ResourceProvider*-Klasse geregelt. Über die Methode *getAllLayouts* lassen sich alle Layouts in eine Collection, bestehend aus *ResourceContainern*, ausgeben. Über diese Collection wird anschließend iteriert, wobei jeder Iterations-Schritt einem Layout entspricht. Jeder Key der Layouts wird dabei mit dem gesuchten Key verglichen. Sollte ein passendes Layout gefunden werden, wird dieses in eine ArrayList gespeichert und die Iteration wird fortgesetzt. Wenn über die gesamte Collection iteriert wurde, kann im Folgenden die ArrayList mit den Suchtreffern zurückgegeben werden.

```
1  public static ArrayList<LayoutResource> searchAlLayoutsByKey(String key)↩
       {
2      ArrayList<LayoutResource> foundLayouts = new ArrayList<>();
3      Collection<ResourceContainer> list = ResourceProvider.getAllLayouts↩
           ();
4      for (ResourceContainer app : list) {
5          LayoutResource layout = (LayoutResource) app.getResource();
6          if(key.equals(layout.getKey())) {
7              foundLayouts.add(layout);
8          }
9      }
10     return foundLayouts;
11 }
```

Quellcode 4.11: getAllLayoutsByKey.java

Da der berechnete Key immer einem *Full*-Layout entspricht, wird zuerst versucht, ein passendes *Full*-Layout in der Hash-Map zu finden. Sollte keines gefunden werden, wird zum nächsten Layout-Typ gewechselt und versucht ein *Compressed*-Layout zu finden, welches passt. Sollte weder ein *Full*- noch ein *Compressed*-Layout gefunden worden sein, wird abschließend versucht, ein *Collapsed-Layout* zu finden. Sollte auch ein solches nicht gefunden werden, wird eine leere ArrayList zurückgegeben, welche entsprechend signalisiert, dass kein passendes Layout in der Datenbank gefunden wurde.

4.2.8 User Interaktion

Ziel der Interaktion ist es, dem User die Freiheit zu geben, verschiedene passende Layouts im direkten Vergleich zu sehen und darauf basierend eine Entscheidung zu treffen. Die Auswahl wird in Tabellenform ausgegeben, zu welcher der User über die zugehörige Schaltfläche *Show suitable Layouts* gelangt (siehe Abb. [4.4]).

Abbildung 4.4: Button in *myApps* um sich passende Layouts anzuzeigen

In der Tabellen-Ansicht (siehe Abb. [4.5]) werden ihm, im Idealfall, mehrere Layouts präsentiert, welche zum Key, der auf Basis seiner bisher eingegeben *Wiring*-Definition berechnet wurde, passen. Diese Tabelle ist sortiert nach Typ und der Reihenfolge, in welcher die Layouts in der Datenbank abgelegt wurden.

ID	Key	Layout	Type	Options
10	1 - 2 - 3	1 > 2 > 3	FULL	Launch
22	1 - 2 - 3	1 > 2 v 3	FULL	Launch
84	1 - 2 - 3	1 s 2 > 3	FULL	Launch
117	1 - 2 - 3	1 > 2	COMPRESSED	Launch

Abbildung 4.5: Tabelle in der passende Layouts zur Auswahl stehen

Wenn sich ein User für ein Layout entschieden hat, kann er über eine entsprechende Schaltfläche auf der rechten Seite der Tabelle das Layout zu seinem *Wiring* hinzufügen. Hierzu wird das gewählte Layout in der expliziten Schreibweise, also mit einer Layout-Zeile, in die letzte Zeile seines *Wirings* eingefügt. Sobald der User die Schaltfläche zur Auswahl betätigt hat, wird er zurück zur *myApps*-Ansicht geleitet, wo die Änderung sichtbar ist.

5 Fensterpositions-Parametrisierung

5.1 Einführung

Der zweite Teil der vorliegenden Bachelorarbeit beschäftigt sich mit der Parametrisierung von Fensterpositionen. Ziel dieser ist es die Möglichkeit zu schaffen, die Fensterpositionen zurück zum Server zu übertragen und im betroffenen *Wiring* zu speichern. Bei einem erneuten Start dieses *Wirings* können die Fenster wieder so hergestellt werden, wie sie beim Schließen zuletzt positioniert waren.

Genutzt werden soll dies nur bei Messgeräten, nicht aber bei Einstellungsfenstern. Hintergrund ist die Speicherung in Parametern, welche zurück zum Server übertragen werden. Diese sind vom Aufbau des *labAlive*-Projektes nur für Messgeräte vorgesehen, weshalb die Umsetzung für Einstellungsfenster nicht möglich ist, ohne einen großen Teil der Programmlogik zu ändern. Dies ist ohnehin nicht notwendig, da Einstellungsfenster lediglich geöffnet werden, um Einstellungen zu ändern und anschließend wieder geschlossen werden.

5.2 Einführung Parameter

Um zu einem späteren Zeitpunkt Fensterpositionen speichern oder laden zu können, bedarf es zuerst einer Datengrundlage als Basis zur Übertragung dieser. Zu diesem Zweck wurde der Datentyp *LocationPoint* eingeführt, welcher auf der Java Klasse *Point* basiert. In diesem werden die X- und Y-Koordinaten in Integer-Genauigkeit gespeichert.

Dieser neu geschaffene Datentyp benötigt ein Parameter-Feld, über welches er später transportiert werden kann. Hierzu wird ein neues *LocationProperty* (siehe Quellcode [5.1]) erstellt, welches wiederum in seiner *createParameter*-Methode beim Start eines *Wirings* einen neuen *LocationParameter* mit dem Namen *Location* initialisiert. Dieser wird initial auf die Koordinaten (0|0) gesetzt.

```
1 public class LocationProperty extends SelectProperty4Measure<↩
      LocationPoint> {
2     public LocationProperty(Parameters parameters) {
3         super(parameters);
4     }
5     @Override
6     protected SelectParameter createParameter() {
7         SelectParameter drawConstellationDiagram = new LocationParameter↩
              ("Location", LocationPoint.NOT_SET_BY_USER);
```

```
 8          drawConstellationDiagram.detailLevel(ParameterDetailLevel.↩
               DETAIL_LEVEL3);
 9          return drawConstellationDiagram;
10      }
11  }
```

Quellcode 5.1: LocationProperty.java

5.3 Positions-Erkennung und Speicherung

Änderungen an den Fenster-Positionen können über das *ComponentListener*-Interface des Java Abstract Window Toolkit (AWT) erkannt werden. Dieses bietet eine *componentMoved*-Methode an, welche aufgerufen wird, sobald eine Komponente, hier ein Fenster, sich in seiner Position verändert. Bei der Verschiebung von Fenstern werden mit jedem Pixel, Events ausgelöst, die von der componentMoved erfasst werden.

An dieser Stelle kann die aktuelle Position genommen und gespeichert werden. Hierzu werden die Parameter des aktuellen Messgerätes aufgerufen und die Position mit X und Y Koordinate an die *setLocation*-Funktion (siehe Quellcode [5.2]) übergeben. In dieser wird ein *LocationPoint* mit den übergebenen Koordinaten erstellt. Um die Position speichern zu können, muss in dem *LocationParameter* ein Value gesetzt werden. Dieser Value besteht neben dem *LocationPoint*, der die eigentliche Position angibt, aus der Angabe einer *ChangePrivilege*. Diese gibt an, dass die Position durch eine User-Interaktion gesetzt wurde und ist für die Verarbeitung von nutzerbasierten Änderungen wichtig.

```
1  public void setLocation(int x, int y) {
2      LocationPoint locationPoint = new LocationPoint();
3      locationPoint.setLocation(x, y);
4      getLocationProperty().setValue(ChangePrivilege.CURRENT_USER_CHANGE, ↩
           locationPoint);
5  }
```

Quellcode 5.2: setLocation.java

Neben der Erkennung durch die Bewegung wird beim Schließen eines Fensters ebenfalls die Position dieses gespeichert. Der Ablauf ist hier praktisch identisch zum Speichern nach Bewegung und wird nur durch eine Methode, die durch das Schließen eines Fensters aufgerufen wird, ausgeführt.

Wenn das *Wiring* nun gespeichert wird und die Positionen hierdurch mit dem Server synchronisiert werden, sind diese anschließend in der *myApps*-Ansicht zu sehen und werden bei einem neuen Start über die Website beachtet.

5.4 Laden von Positionen

Ist ein *Wiring* zuvor bereits gestartet worden und die Position wurde verändert, ist diese Änderung in der Website sichtbar (siehe Abb. [5.1]).

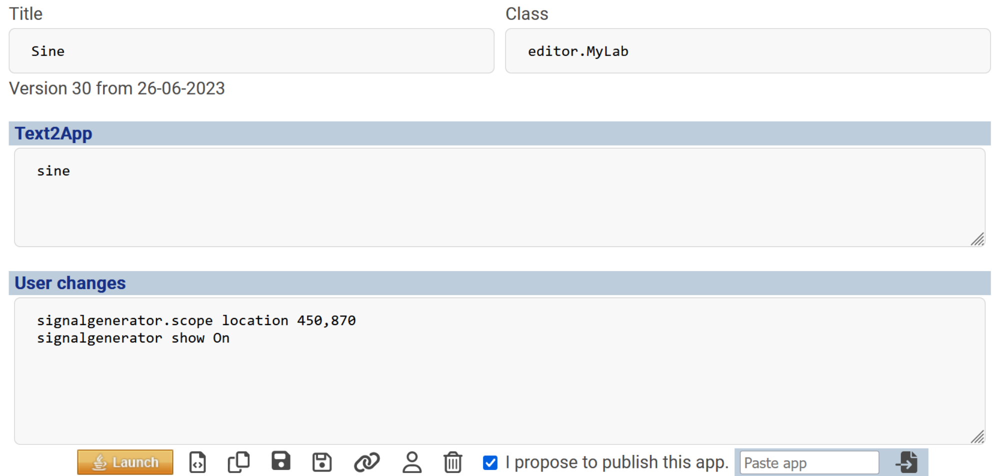

Abbildung 5.1: myApps - Messgerät mit gespeicherter Position

Wird dieses *Wiring* nun gestartet, werden beim Programmstart in der *StartSystemSignalingsSender*-Klasse die Einstellungen geladen. In dem Aufruf der *takeReference4UserChangesApplyUserChanges*-Methode der *ConfigInitializer*-Klasse werden zum Ende dieser alle übergebenen User-Änderungen angewandt. Gestartet wird das *Wiring* dann mit der veränderten Position für die entsprechenden Fenster.

6 Fazit

6.1 Ergebnisse

Im Rahmen dieser Bachelorarbeit konnte durch die Umsetzung der zwei Projekte die Benutzerfreundlichkeit des *labAlive*-Portal für User nachhaltig verbessert werden.

Durch die Implementierung des neuen Systems für die Verwaltung von Layouts konnte dieses deutlich User-freundlicher gestaltet werden. Nun können diese ohne große Vorerfahrung direkt genutzt werden, was nachhaltig dazu anregen soll, selbst Layouts zu erstellen. Darauf basierend wird das Feature umso mächtiger, je mehr User darauf zugreifen und so eine stetig wachsende Anzahl an Layouts zu Verfügung gestellt werden kann. Diese können im Idealfall alle Simulationen abdecken. Durch die Erstellung von entsprechenden JUnit-Tests konnte die Funktion für verschiedene Varianten automatisiert getestet werden, um zu verifizieren, dass die Funktionalität für verschiedene positive und negative Testbeispiele lauffähig und fehlerfrei funktioniert.

Neben der Einführung der Layout-Verwaltung konnte über die Parametrisierung der Fensterpositionen die Wiederverwendbarkeit von bereits erstellten Simulationen deutlich gesteigert werden. Auch wenn schlussendlich die Funktionalität nicht bei allen Arten von Messgeräten umgesetzt werden konnte, ist der Grundstein für die vollständige Umsetzung gelegt, und wird so in naher Zukunft vervollständigt werden können. Da das korrekte Laden des Parameters bzw. das Speichern dieser nicht mit JUnit-Tests abgedeckt werden konnte, wurde die Funktionalität hauptsächlich über manuelles Testen verifiziert.

6.2 Ausblick

Durch die beschränkte Zeit, die für ein Projekt im Rahmen der Bachelorarbeit zur Verfügung steht, können dementsprechend nicht immer alle gewünschten Themen behandelt bzw. vollumfänglich bearbeitet werden. Allerdings ist es möglich durch die vorliegende Arbeit eine wichtige Grundlage für weitere Arbeiten des *labAlive*-Projekts zu stellen.

Folgend werden einige Ideen für weitere Funktionalitäten oder Themen aufgeführt die im Rahmen weiterer Arbeiten umgesetzt bzw. behandelt werden könnten:

- Vervollständigung der Fenster-Parametrisierung für alle Fenster-Typen
- Grafische Vorschau für Layouts vor Auswahl

- Verbesserung der Auto-Positionierung von Fenstern in gestarteten Simulationen
- Ausgabe von aussagekräftigeren Fehlermeldungen für Nutzer bei gescheitertem Simulations-Start
- System zur Berechnung von Collapsed-Layouts aus Full und/oder Compressed
- Verbesserung der Nutzeroberfläche in der Web-Oberfläche

6.3 Schlussfolgerung

Im Zuge der Bachelorarbeit und der vorrausgegangen Projektarbeit konnte ein tieferes Verständnis für die Arbeit in größeren Softwareprojekten und der damit verbunden Möglichkeiten, sowie Herausforderungen gewonnen werden. Durch die studieninhalts-nahen Themen konnte sich viel Gelerntes aus dem Studium praxisnah einsetzen und vertiefen lassen. Neben der Vertiefung konnten aber auch viele neue Einblicke in der Welt der Kommunikationstechnik, sowie der Programmierung von Client-Server-Anwendungen gewonnen werden. Die Programmier-Fähigkeiten waren stets gefragt und konnten stetig weiter verbessert werden.

Anhang

Im Anhang wird eine detallierteSchritt-für-Schritt-Anleitung für die Verwendung des neuen Features zum Abrufen von passenden Layouts dargestellt werden:

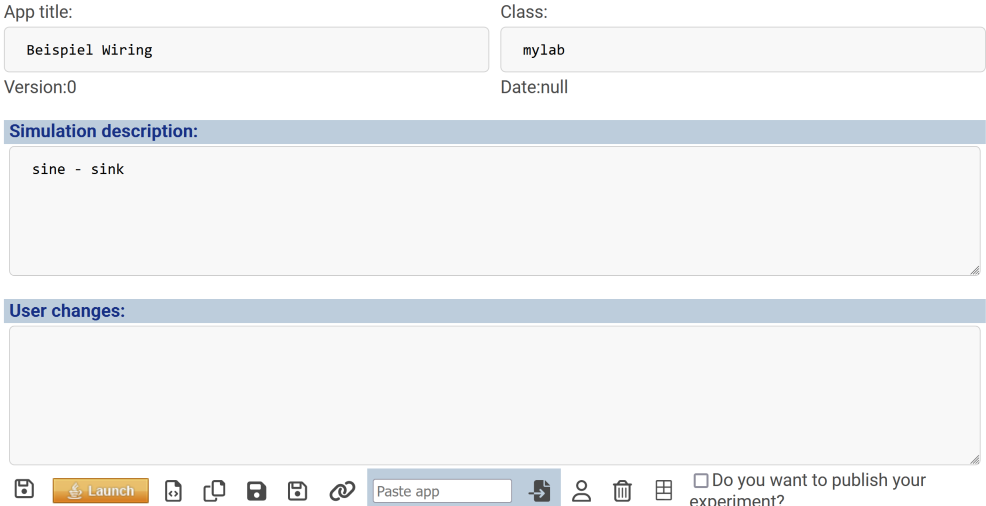

Abbildung 6.1: Schritt 1: Wiring ohne Layout-Angabe

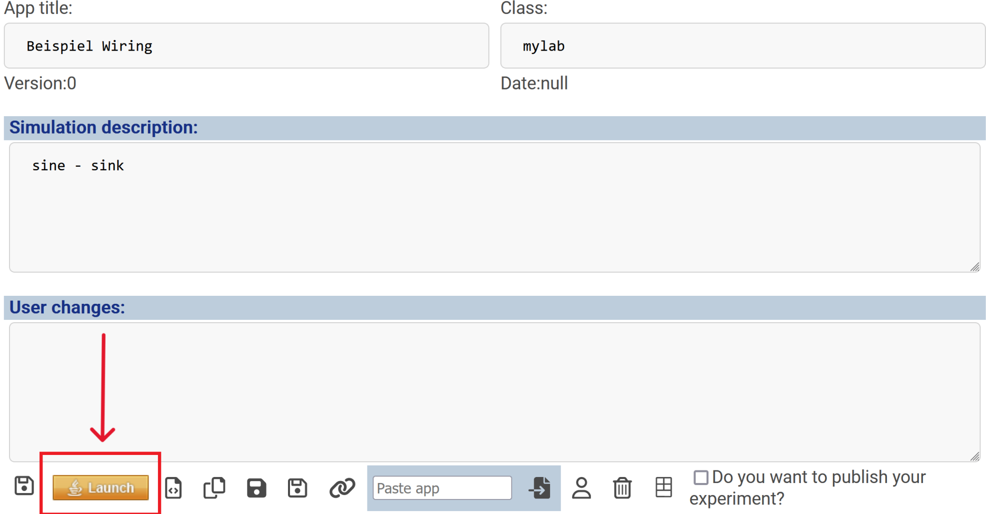

Abbildung 6.2: Schritt 2: Starten ohne Layout-Angabe

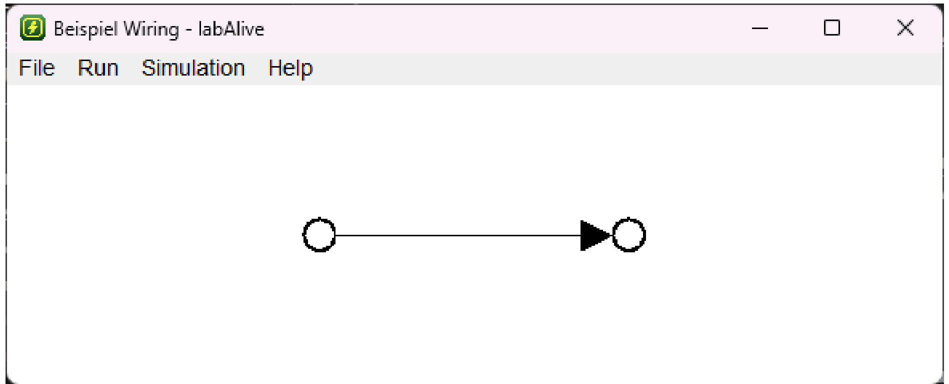

Abbildung 6.3: Resultat aus Schritt 2: Autopositionierung ohne Layout

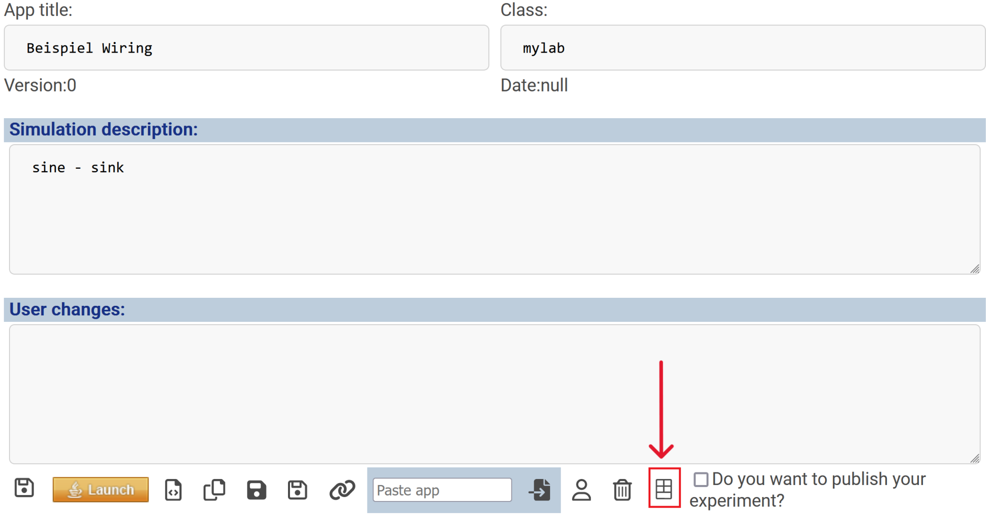

Abbildung 6.4: Schritt 3: Anklicken *Show suitable Layouts*

ID	Key	Layout	Type	Options
1	1 - 2	1 > 2	FULL	Launch
12	1 - 2	1 v 2	FULL	Launch
33	1 - 2 - 3	1 s 2 > 3	FULL	Launch
54	1 - 2	1 >vv>^^>>v< 2	FULL	Launch

Abbildung 6.5: Resultat aus Schritt 3: Layout-Tabelle

ID	Key	Layout	Type	Options
1	1 - 2	1 > 2	FULL	Launch
12	1 - 2	1 v 2	FULL	Launch
33	1 - 2 - 3	1 s 2 > 3	FULL	Launch
54	1 - 2	1 >vv>^^>>v< 2	~~FULL~~	Launch

Abbildung 6.6: Schritt 4: Layout in Tabelle auswählen

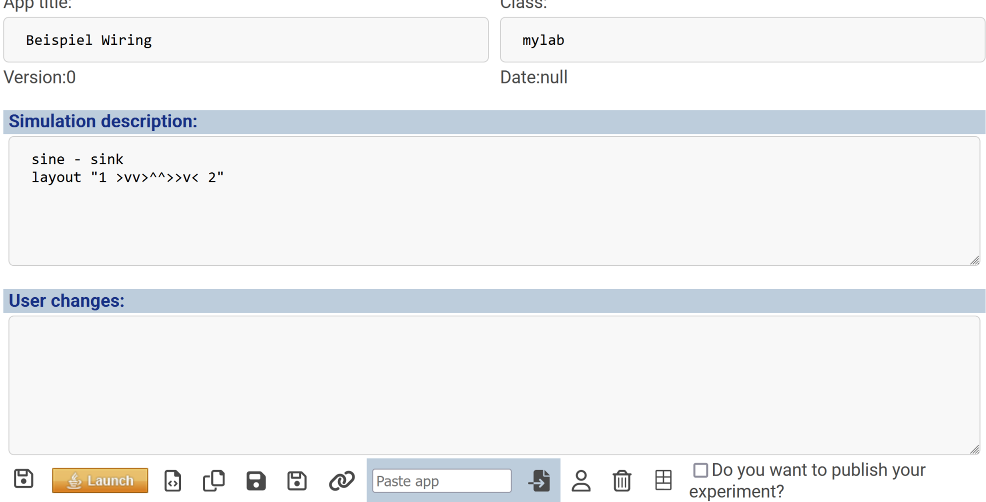

Abbildung 6.7: Resultat aus Schritt 4: Ergänztes Wiring

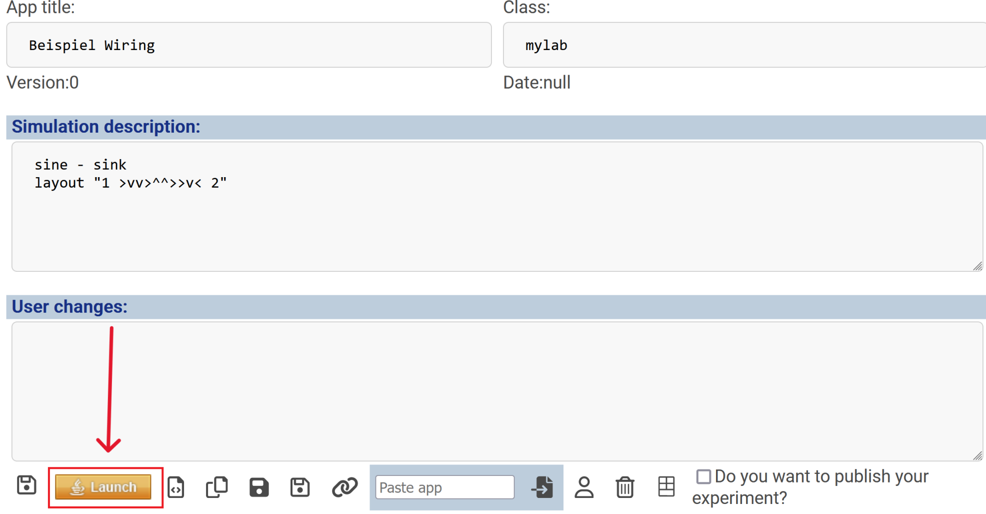

Abbildung 6.8: Schritt 5: Starten mit Layout-Angabe

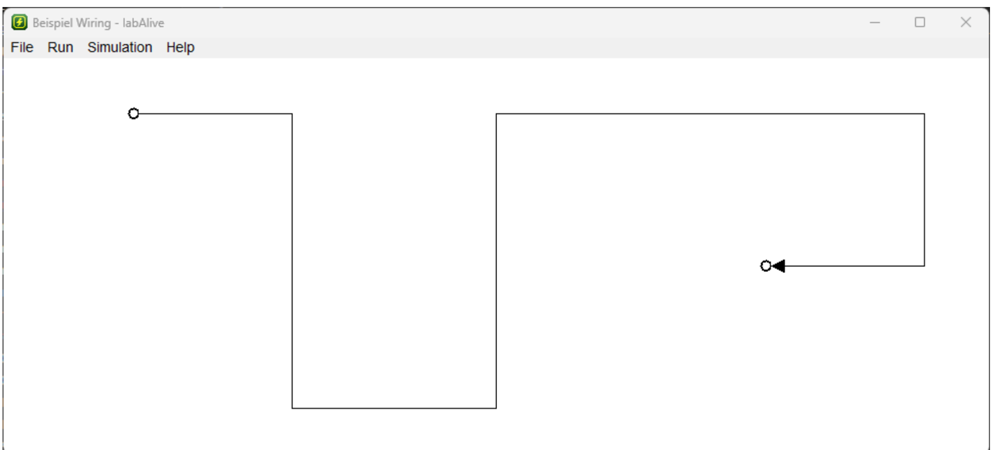

Abbildung 6.9: Resultat aus Schritt 5: Starten mit Layout-Angabe

Abbildungsverzeichnis

2.1 Gitea-Logo [3] 3
2.2 IntelliJ-Logo [5] 3
2.3 Eclipse-Logo [7] 3
2.4 Access-Logo [9] 4
2.5 Java-Logo [11] 4

3.1 Eingabemaske myApps 5
3.2 Beispiel gestartete Simulation 6

4.1 Flussdiagramm *wiringLayoutProcessor* 12
4.2 Syntaxgraph implizites Layout (ohne Leerzeichen) 14
4.3 Syntaxgraph explizites Layout (ohne Leerzeichen) 16
4.4 Button in *myApps* um sich passende Layouts anzuzeigen 22
4.5 Tabelle in der passende Layouts zur Auswahl stehen 22

5.1 myApps - Messgerät mit gespeicherter Position 27

6.1 Schritt 1: Wiring ohne Layout-Angabe I
6.2 Schritt 2: Starten ohne Layout-Angabe II
6.3 Resultat aus Schritt 2: Autopositionierung ohne Layout II
6.4 Schritt 3: Anklicken *Show suitable Layouts* III
6.5 Resultat aus Schritt 3: Layout-Tabelle III
6.6 Schritt 4: Layout in Tabelle auswählen IV
6.7 Resultat aus Schritt 4: Ergänztes Wiring IV
6.8 Schritt 5: Starten mit Layout-Angabe V
6.9 Resultat aus Schritt 5: Starten mit Layout-Angabe V

Tabellenverzeichnis

3.1 Richtungsangaben Layout 7

4.1 Regex-Pattern zur Layout-Erkennung 15

Quellcodeverzeichnis

3.1 Beispiel Key 6
3.2 Beispiel Layout-Objekt 7
3.3 Vergleich Implizite und Explizite Definition 7
3.4 Beispiel *Full*-Layout 8
3.5 Vergleich *Full*- und *Compressed*-Layout 9
3.6 Beispiel Collapsed-Layout 9

4.1 Syntax der Layout-Typen 13
4.2 *Full*-Layout mit Einstellungen 13
4.3 Pattern zur Erkennung eines impliziten Layouts 15
4.4 Pattern zur Erkennung eines expliziten Layouts 16
4.5 transformImplicitToMinusSeperated.java 17
4.6 transformImplicitToSystemToNumbers.java 17
4.7 systemToNumbers.java 18
4.8 transformToExplicitLayout.java 19
4.9 systemIsMIMO.java 19
4.10 extractLayout.java 20
4.11 getAllLayoutsByKey.java 21

5.1 LocationProperty.java 25
5.2 setLocation.java 26

Stichwortverzeichnis

Ablauf, 12
Anwendung, 6
Aufbau der Arbeit, 1
Ausblick, 29
Automatisierte Layout Verwaltung, 11

Collapsed, 9
Compressed, 8

Eclipse, 3
Einführung, 11, 25
Einführung Parameter, 25
Einleitung, 1
Einstiegspunkt und Parameter, 11
Ergebnisse, 29

Fazit, 29
Fensterpositions-Parametrisierung, 25
Full, 8

Gitea, 3

HTML, 4

IntelliJ, 3

Java, 4

labAlive, 5
Laden von Positionen, 27
Layout-Erkennung, 13
Layout-Suche, 21
Layouts, 6

Microsoft Access, 4

Positions-Erkennung und Speicherung, 26
Programmiersprachen, 4

Schlussfolgerung, 30
Software, 3
Speicherung, 20
SQL, 4

Transformation, 16
Typen, 8

Umrechnung, 19
User Interaktion, 22

Website, 5
wiringLayoutProcessor, 11

Ziel der Arbeit, 1

Literaturverzeichnis

[1] Prof. Dr.-Ing. Erwin Riederer. *labAlive Website*.
URL: https://www.etti.unibw.de/labalive/ (besucht am 20. 06. 2023) (siehe Seite 1).

[2] *Gitea*. URL: https://about.gitea.com/ (besucht am 22. 06. 2023) (siehe Seite 3).

[3] *Gitea Logo*. URL: https://about.gitea.com/images/gitea.svg (besucht am 22. 06. 2023) (siehe Seite 3).

[4] *IntelliJ*. URL: https://www.jetbrains.com/de-de/idea/ (besucht am 22. 06. 2023) (siehe Seite 3).

[5] *IntelliJ Logo*. URL:
https://resources.jetbrains.com/storage/products/company/brand/logos/IntelliJ_IDEA_icon.png (besucht am 22. 06. 2023) (siehe Seite 3).

[6] *Eclipse*. URL: https://eclipseide.org/ (besucht am 22. 06. 2023) (siehe Seite 3).

[7] *Eclipse Logo*.
URL: https://www.eclipse.org/org/artwork/zip_files/eclipse-ide-logo.zip (besucht am 22. 06. 2023) (siehe Seite 3).

[8] *Access*. URL: https://www.microsoft.com/access (besucht am 22. 06. 2023) (siehe Seite 4).

[9] *Access Logo*. URL:
https://upload.wikimedia.org/wikipedia/commons/thumb/f/f1/Microsoft_Office_Access_%282019-present%29.svg/1920px-Microsoft_Office_Access_%282019-present%29.svg.png (besucht am 22. 06. 2023) (siehe Seite 4).

[10] *Java*. URL: https://www.java.com/download/help/whatis_java.html (besucht am 22. 06. 2023) (siehe Seite 4).

[11] *Java Logo*. URL: https://upload.wikimedia.org/wikipedia/de/thumb/e/e1/Java-Logo.svg/1024px-Java-Logo.svg.png (besucht am 22. 06. 2023) (siehe Seite 4).

[12] *HTML*. URL: https://wiki.selfhtml.org/ (besucht am 22. 06. 2023) (siehe Seite 4).

[13] *SQL*. URL: https://aws.amazon.com/de/what-is/sql/ (besucht am 22. 06. 2023) (siehe Seite 4).