

Eine Entwicklungsumgebung für ein grafisches Programmierkonzept

Christian Colbach

Bachelor-Abschlussarbeit

Betreuer: Prof. Dr.-Ing. Christof Rezk-Salama

Trier 31.08.2017

---

## Kurzfassung

Viele grafische Programmierkonzepte haben entweder die Eigenschaft nur eine grafische Benutzeroberfläche um ein trotzdem eigentlich textbasiertes Konzept zu bauen oder aber nur ein sehr spezielles Anwendungsgebiet abzudecken. Bei meiner Arbeit beschreibe ich ein alternatives grafisches Programmierkonzept, welches versucht diese Eigenschaften nicht zu erfüllen. Daraufhin erkläre ich die Verwendung aller benötigten Werkzeuge um mit diesem Programmierkonzept Projekte umsetzen zu können und beschreibe wie ich diese implementiert habe. Zum Schluss gehe ich auf die Vor- und Nachteile meiner Umsetzung ein und komme zum Ergebnis, dass trotz bestehender Schwächen das Konzept verwendbar ist und eine Hilfe sein kann, Aufgaben schnell und einfach zu lösen.

---

# Inhaltsverzeichnis

<b>1 Einleitung</b>	1
1.1 Fragestellung / Problemstellung	1
1.2 Zielsetzung	2
1.3 Entwicklung des Konzeptes und Planung der Umsetzung	2
1.4 Einsatzbereiche	3
1.5 Programmcode	4
<b>2 Benutzerhandbuch</b>	5
2.1 Einrichtung	5
2.1.1 Systemvoraussetzungen	5
2.1.2 Starten des Hauptprogramms	5
2.2 Verwendung als Endnutzer	6
2.2.1 Funktionsweise	6
2.2.2 Beispiele	13
2.2.3 Benutzerinterface	20
2.2.4 Auswahlfenster für Elemente	31
2.2.5 Fenster für Einstellungen	32
2.2.6 Fenster für System Ein-/Ausgaben ( <i>Konsole</i> )	32
2.2.7 Berichte	33
2.2.8 Debugger	34
2.2.9 Direkte Eingaben	38
2.3 Verwendung als Programmierer (Erweiterung des bestehenden Frameworks)	40
2.3.1 Umsetzung von Element-Definitionen	40
<b>3 Technische Umsetzung</b>	47
3.1 Implementierung	47
3.1.1 Umfang	47
3.1.2 Aufbau	48

3.1.3 Hauptmodul . . . . .	49
3.1.4 Hilfs-Klassen . . . . .	49
3.1.5 Model . . . . .	49
3.1.6 Kommandozeile . . . . .	56
3.1.7 Grafische Benutzeroberfläche . . . . .	56
3.1.8 Persistente Einstellungen . . . . .	63
3.1.9 Protokollierung . . . . .	64
3.1.10 Über Reflexion geladene Klassen und geteilte Klassen/Schnittstellen . . . . .	64
3.1.11 Weitere Konzepte . . . . .	65
3.1.12 Verwendete Bibliotheken . . . . .	66
3.2 Verwendete und erstellte Grafiken . . . . .	66
 <b>4 Analyse</b> . . . . .	67
4.1 Vorteile des Programmierkonzeptes . . . . .	67
4.1.1 Parallelisierung . . . . .	67
4.1.2 Keine Programmierkenntnisse notwendig . . . . .	67
4.1.3 Schnelle Ergebnisse . . . . .	67
4.1.4 Einfache Erweiterung . . . . .	67
4.2 Nachteile des Programmierkonzeptes . . . . .	68
4.2.1 Begrenzter Baukasten . . . . .	68
4.2.2 Eingeschränkter Funktionsumfang des Benutzerinterface . . . . .	68
4.2.3 Hoher Ressourcenverbrauch . . . . .	68
4.2.4 Fehlen von Funktionalitäten konventioneller Programmierparadigmen . . . . .	68
4.3 Turing-Vollständigkeit . . . . .	69
4.3.1 Beweis Turing-Vollständigkeit . . . . .	69
4.3.2 Beispiel . . . . .	70
 <b>5 Fazit</b> . . . . .	72
 <b>6 Ausblick</b> . . . . .	73
6.1 Optimierung . . . . .	73
6.1.1 Verwendung von Threadpools . . . . .	73
6.1.2 Verbesserung der automatischen Kompatibilität . . . . .	73
6.2 Ausbau des Frameworks . . . . .	74
6.3 Erweiterung der grafischen Benutzeroberfläche . . . . .	74
6.4 Bündeln diverser Teilmodule als Library . . . . .	74
 <b>Erklärung des Kandidaten</b> . . . . .	75

---

## Abbildungsverzeichnis

1.1 EditorPanel . . . . .	1
1.2 Mockup (erstellt in Adobe Photoshop) . . . . .	3
2.1 Einfaches Element . . . . .	11
2.2 Spezielles Element . . . . .	12
2.3 Kontext erzeugendes Element . . . . .	13
2.4 Dialoge, die der Benutzer beim Primzahl-Quiz angezeigt bekommt . . . . .	15
2.5 Inhalt des Ordners “Überwachungskamera” . . . . .	16
2.6 Toolbar . . . . .	21
2.7 Arbeitsfläche . . . . .	26
2.8 Verschieben von Elementen . . . . .	26
2.9 Verbinden von Elementen . . . . .	27
2.10 Target-Position . . . . .	27
2.11 Verbindungs-Overlay . . . . .	28
2.12 Verbinden von einem Ausgang mit einem Eingang über das Verbindungs-Overlay . . . . .	28
2.13 Kontextmenü (auf leere Arbeitsfläche) . . . . .	29
2.14 Kontextmenü (auf Element) . . . . .	30
2.15 Zusätzliche Informationen . . . . .	30
2.16 Meldungen im Hauptfenster . . . . .	31
2.17 Auswahlfenster für Elemente . . . . .	31
2.18 Fenster für Einstellungen . . . . .	32
2.19 Konsole . . . . .	33
2.20 Bericht . . . . .	33
2.21 Haltemarke . . . . .	34
2.22 Auslöser Toolbar . . . . .	34
2.23 Debugger . . . . .	35
2.24 Farben Zustände . . . . .	35
2.25 Eingegangene Daten . . . . .	35

2.26	Inspect-Overlay .....	36
2.27	Kontext-Übersicht .....	37
2.28	Monitor-Fenster .....	38
2.29	Eingabe (Matrix) .....	39
2.30	Eingabe (Quellcode) .....	40
3.1	Aufbau Module .....	48
3.2	Aufbau Model .....	50
3.3	Grundaufbau Projektausführung .....	54
3.4	Storyboard .....	57
3.5	Zeichen-Methoden .....	59
3.6	ShowStates .....	60
3.7	Ladebalken .....	60
3.8	Toolbar Storyboard .....	61
3.9	Scrollbar .....	62
3.10	Templategenerator .....	65
4.1	Umsetzung Universelle Turingmaschine .....	69
4.2	Eingaben Universelle Turingmaschine .....	70
4.3	Ergebnis der Ausführung .....	71

# 1

## Einleitung

### 1.1 Fragestellung / Problemstellung

2016 habe ich unter dem Namen *EditorPanel*<sup>1</sup> (Abb. 1.1) eine Software für eine grafische „Blueprint“-artige Editierung von Bildern entwickelt.

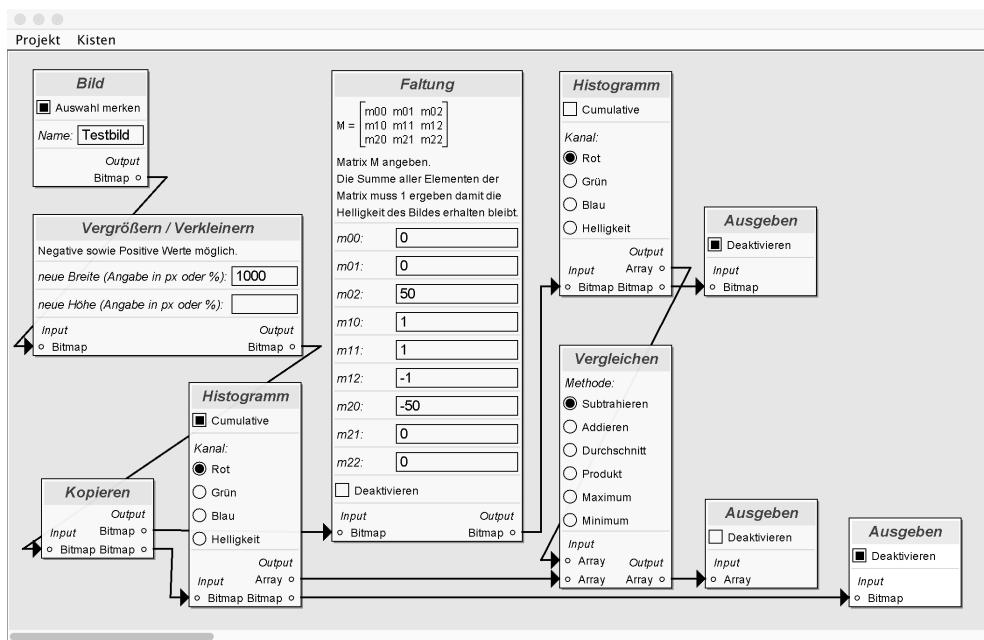


Abb. 1.1. EditorPanel

Prinzipiell war dieses Projekt ein Erfolg und hat mich vom Potential dieses Konzeptes überzeugt, jedoch hat es mir auch aufgezeigt, wo die Grenzen dieses

<sup>1</sup> <https://github.com/colbach/EditorPanel>

Konzeptes lagen und hat mich zu Überlegungen angeregt, ob und wie dieses verbessert werden kann.

Das offensichtliche Problem von EditorPanel besteht darin, dass es keinerlei Art von Kontrollstrukturen unterstützt. Um herauszufinden, wie sich solche am besten in mein Projekt integrieren lassen, habe ich mir hierzu verwandte Konzepte der grafischen Programmierung wie unter anderem *Unreal Blueprints*<sup>2</sup>, *Lego Mindstorms Software*<sup>3</sup>, *LabVIEW*<sup>4</sup> und *App Inventor*<sup>5</sup> angeschaut. Bei allen diesen grafischen Konzepten gibt es zwei Probleme, die störend sind. Einerseits haben diese Systeme meistens die Eigenschaft entweder ein sehr spezielles Anwendungsgebiet zu haben oder andererseits zu nah an der klassischen – textbasierten – Programmierung zu sein und eigentlich nur eine grafische Betrachtung und Editierung dieser bereitzustellen.

## 1.2 Zielsetzung

Ziel meiner Arbeit ist es, ein alternatives, einfach zu bedienendes, einfach zu erweiterndes und allgemein gehaltenes grafisches Programmierkonzept auszuarbeiten und die Entwicklungsumgebung, um in diesem zu arbeiten und dieses auszuführen, zu entwickeln.

## 1.3 Entwicklung des Konzeptes und Planung der Umsetzung

Bei der Entwicklung meines Konzeptes nehme ich als Ausgangsbasis die Funktionsweise von EditorPanel und versuche diese um Kontrollstrukturen und weitere Konzepte zu erweitern, welche es ermöglichen, komplexe Aufgaben zu lösen. Die Software, die hierbei entsteht, baut nicht auf EditorPanel auf und wird von Grund auf neu entwickelt.

Konkret stelle ich folgende Anforderungen an mein Programm:

- Grundlegende mathematische Operationen
- Wiederholende Tätigkeiten
- Bedingte Anweisungen
- Verschiedene Arten von Eingabe / Ausgabe
- Reaktion auf Ereignisse
- Leichte Erweiterbarkeit

---

<sup>2</sup> <https://docs.unrealengine.com/latest/INT/Engine/Blueprints/>

<sup>3</sup> <https://www.lego.com/de-de/mindstorms/downloads/download-software>

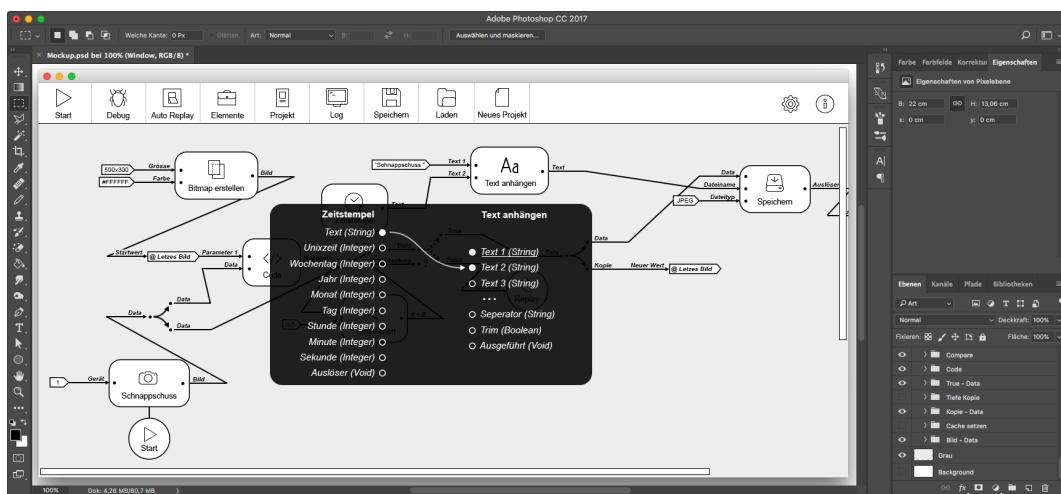
<sup>4</sup> <http://www.ni.com/de-de/shop/labview.html>

<sup>5</sup> <http://appinventor.mit.edu/>

- Einfach zu verwendendes Benutzerinterface

Die Erfüllung dieser Anforderungen ist mit einer Reihe von Problemen verbunden, denen ich mich einzeln widmen werde und versuchen werde, sie zu lösen.

Um eine Vorstellung davon zu bekommen wie die Software später aussehen soll, die ich entwickeln werde, baue ich mir ein Mockup (Abb. 1.2), das mich bei der Entwicklung begleiten soll und mir als Vorlage dient das Benutzerinterface zu implementieren. Hierbei ist zu beachten, dass es sich nur um einen ersten Entwurf handelt, der bereits vor der ersten Zeile Programmcode entstanden ist. Während der Entwicklung wurden viele Ideen verworfen und neue Ideen hinzugefügt.



**Abb. 1.2.** Mockup (erstellt in Adobe Photoshop)

## 1.4 Einsatzbereiche

Das Einsatzgebiet soll bewusst offen gehalten werden. Jedoch gibt es bestimmte Bereiche, bei denen ich mir die Verwendung gut vorstellen kann.

Diese Bereiche sind:

- Bildverarbeitung
- Mathematische Berechnungen
- Überwachungstechnik
- Programmierung von Geräten für das *Internet der Dinge*
- DIY Heimprojekte

## 1.5 Programmcode

Sämtlicher Programmcode, ergänzende Dokumentation und Beispiele, die im Rahmen dieser Arbeit entstanden sind, stehen auf GitHub unter folgendem Link zum Download bereit:

<https://github.com/colbach/Bachelor-Projekt>

## 2

---

# Benutzerhandbuch

## 2.1 Einrichtung

### 2.1.1 Systemvoraussetzungen

#### Mindestvoraussetzung

- Java 8 JRE<sup>1</sup>
- 64 MB freier Hauptspeicher für die JVM

**Empfohlene System-Eigenschaften, um sämtliche Features optimal nutzen zu können**

- Java 8 JDK<sup>2</sup> (Für Kompilierung zur Laufzeit von Javacode)
- OpenCV<sup>3</sup> (Für Camera-Features)
- 512 MB freier Hauptspeicher für die JVM (Dies wird erzwungen durch den VM-Parameter `-Xmx512m`)

### 2.1.2 Starten des Hauptprogramms

#### Unix-basierte Betriebssysteme (Kommandozeile)

Navigieren Sie im Terminal in das Programm-Verzeichnis und geben Sie folgenden Befehl ein:

```
java -cp ".:lib/*" main.MainClass
```

---

<sup>1</sup> <https://java.com/de/download/>

<sup>2</sup> <http://www.oracle.com/technetwork/java/javase/downloads/>

<sup>3</sup> <http://opencv.org/>

## Windows (Kommandozeile)

Navigieren Sie in der Windows-Shell in das Programm-Verzeichnis und geben Sie folgenden Befehl ein:

```
java -cp ".;lib/*" main.MainClass
```

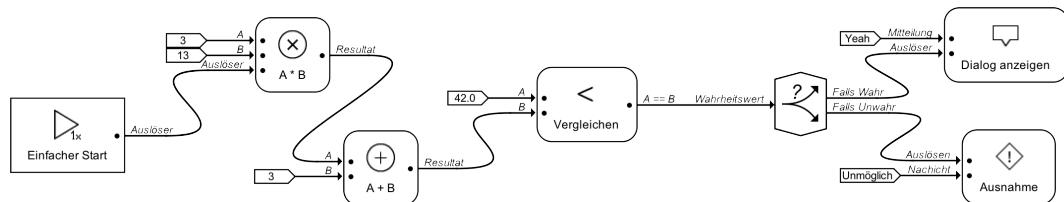
## Verwendung des Launchers

Alternativ zum Start über die Kommandozeile, kann die Anwendung auch durch ein *Doppelklick* auf die Datei “Launcher.jar” gestartet werden (Voraussetzung hierfür ist dass die Dateiendung “jar” mit dem Programm “javaw.exe” bzw. “Jar Launcher.app“ assoziiert ist).

## 2.2 Verwendung als Endnutzer

### 2.2.1 Funktionsweise

#### Grundsätzlicher Aufbau



#### Ein-/Ausgänge

Ein-/Ausgänge bestehen jeweils aus einem Namen und einem Datentyp.

#### Virtuelle Ein-/Ausgänge

Virtuelle Ein-/Ausgänge sind Ein-/Ausgänge, die von den Elementen selbst nicht vorgesehen sind und nur von der Umgebung unterstützend verwendet werden. Konkret gibt es drei Arten von virtuellen Eingängen:

- **copy** (nur Ausgang): Diese Ausgänge ermöglichen es, Daten an mehrere Elemente zu übergeben ohne diese neu berechnen zu müssen. Diese Ausgänge sind für alle nicht virtuellen Ausgänge verfügbar.

- `append` (nur Eingang): Diese Eingänge ermöglichen es, Daten, die an einem Eingang eingehen sollen, aus mehreren Ausgängen zusammenzusetzen. Diese Eingänge sind nur für nicht virtuelle Eingänge, die Listen akzeptieren, verfügbar.
- Auslöser (Ein- und Ausgang): Dieser Eingang oder Ausgang dient nur dazu ein vorhergehendes oder nachfolgendes Element anzustoßen. Es werden keine Daten angenommen oder weitergereicht.

### *Optionale Eingänge*

Eingänge sind *optional*, wenn sie nicht verbunden werden müssen damit ein Element funktionieren kann. Anstelle von Eingangsdaten greift das Element in diesem Fall auf Standardwerte oder dynamisch generierte Werte zurück, sodass es nicht zu einem Fehler kommt.

### *Dynamische Eingänge*

Dynamische Eingänge sind Eingänge, die sich dynamisch hinzuschalten, wenn die Eingänge davor verbunden sind. Anzumerken hierzu ist, dass es sich bei dynamischen Eingängen logischerweise immer auch um optionale Eingänge handelt, auch wenn in dem Benutzerinterface nicht gesondert darauf hingewiesen wird.

### *Datentypen*

Datentypen bestehen jeweils aus einer (Java) Basisklasse/Schnittstelle und der Angabe ob es sich um einen Listen-Datentypen handelt oder nicht.

Die Beschreibungen der Datentypen setzen sich aus folgenden zwei Teilen zusammen:

1. (Evtl. übersetzter) Klassenname
2. “...” falls es sich um einen Listen-Datentyp handelt oder “” (leer) falls nicht.

Beispiele für korrekte Bezeichnungen:

- Matrix (Instanz der eigenen Schnittstelle `reflection.customdatatype.math.Matrix`)
- Text... (Liste von Instanzen der Klasse `java.lang.String`)
- Vektor... (Liste von Instanzen der eigenen Schnittstelle `reflection.customdatatype.math.Vector`)
- Farbe (Instanz der Klasse `java.awt.Color`)
- 64 Bit Ganzzahl (Instanz der Klasse `java.lang.Long`)
- Zahl... (Liste von Instanzen der Schnittstelle `java.lang.Number`)

Folgende Basisklassen/Schnittstellen werden jeweils als Datentyp für einen einzelnen Wert sowie als Listen-Datentypen verwendet:

- “Wahrheitswert” (Basierend auf: `java.lang.Boolean`)
- “Text” (Basierend auf `java.lang.String`)
- “Zeichen” (Basierend auf `java.lang.Character`)
- “Zahl” (Basierend auf `java.lang.Number`)
- “8 Bit Ganzzahl” (Basierend auf `java.lang.Byte`)
- “16 Bit Ganzzahl” (Basierend auf `java.lang.Short`)
- “32 Bit Ganzzahl” (Basierend auf `java.lang.Integer`)
- “64 Bit Ganzzahl” (Basierend auf `java.lang.Long`)
- “32 Bit Gleitkommazahl” (Basierend auf `java.lang.Float`)
- “64 Bit Gleitkommazahl” (Basierend auf `java.lang.Double`)
- “Datum” (Basierend auf `java.util.Date`)
- “Farbe” (Basierend auf `java.awt.Color`)
- “Objekt” (Basierend auf `java.lang.Object`)
- “Funktion” (Basierend auf `reflection.customdatatypes.Function`)
- “Bild” (Basierend auf `java.awt.Image`)
- “Gepuffertes Bild” (Basierend auf `java.awt.BufferedImage`)
- “Rohdaten” (Basierend auf `reflection.customdatatypes.rawdata.RawData`)
- “Gelesene Rohdaten” (Basierend auf `reflection.customdatatypes.rawdata.RawDataFromFile`)
- “Empfangene Rohdaten” (Basierend auf `reflection.customdatatypes.rawdata.RawDataFromNetwork`)
- “Pfad” (Basierend auf `java.nio.file.Path`)
- “Dateipfad” (Basierend auf `java.nio.File`)
- “Kamera” (Basierend auf `reflection.customdatatypes.camera.Camera`)
- “Raster” (Basierend auf `reflection.customdatatypes.BooleanGrid`)
- “Matrix” (Basierend auf `reflection.customdatatypes.math.Matrix`)
- “Vektor” (Basierend auf `reflection.customdatatypes.math.Vector`)
- “Mathematisches Object” (Basierend auf `reflection.customdatatypes.math.MathObject`)
- “Quellcode” (Basierend auf `reflection.customdatatypes.SourceCode`)
- “SmartIdentifier” (Basierend auf `reflection.customdatatypes.SmartIdentifier`)

Im Folgenden werden die Datentypen Text, Zeichen, 8/16/32/64 Bit Ganzzahl sowie 32/64 Bit Gleitkommazahl als *Primitive Datentypen* bezeichnet.

Ein Weiterer jedoch nicht als Listen-Datentyp verwendbarer Datentyp ist der **Void-Datentyp**, der auf der Klasse `java.lang.Void` basiert. Dieser Datentyp wird im Allgemeinen nur verwendet, um Elemente anzustoßen, ohne diesen jedoch Daten zu übergeben.

## Kompatibilität von Ein-/Ausgängen

Ein Ausgang ist im einfachsten Fall zu einem Eingang kompatibel wenn sein Datentyp sich auf die gleiche Basisklasse bezieht wie der Datentyp des Eingangs. Hierbei spielt es in erster Instanz keine Rolle, ob es sich bei beiden Datentypen um Listen-Datentypen handelt oder nicht.

Auch wenn Ein- und Ausgänge den gleichen Datentyp haben sollen, können Ein-/Ausgänge mit unterschiedlichen Basisklassen jedoch unter bestimmten Situationen trotzdem zueinander kompatibel sein.

*Fall 1:*

Die Basisklasse des Ausgangs ist eine Ableitung von der Basisklasse des Eingangs. Dies ist der unproblematischste Fall da es in diesem Fall nie zu einem Laufzeit-Fehler kommen kann. Instanzen einer Klasse, die von einer anderen zweiten Klasse abgeleitet sind, sind de facto auch Instanzen dieser zweiten Klasse.

*Fall 2:*

Die Basisklasse des Eingangs ist eine Ableitung von der Basisklasse des Ausgangs. Dies ist der problematischste Fall. Je nach Situation können Daten, die von Ausgängen kommen, die sich auf eine Oberklasse einer zweiten Klasse beziehen, auch Instanzen dieser zweiten Klasse sein. Dies ist insbesondere dann der Fall wenn ein Ausgang den sehr allgemeinen Typ “Objekt” oder “Object...” besitzt. In diesem Fall ist es als Benutzer wichtig zu beachten, welche Daten von einem Ausgang erwartet werden.

*Fall 3:*

In bestimmten Fällen wird zur Erhöhung der Kompatibilität eine automatische Konvertierung durchgeführt. In diesen Fällen können Ein-/Ausgänge komplett unterschiedlicher Typen trotzdem kompatibel zueinander sein.

In folgenden Fällen ist dies der Fall:

- Der Eingang oder der Ausgang haben als Datentyp “Void”.
- Der Eingang und der Ausgang haben als Datentyp jeweils einen *Primiven Datentyp*.
- Der Eingang hat den Datentyp “Text” und der Ausgang hat den Datentyp “Objekt”.
- Der Eingang hat den Datentyp “Dateipfad” und der Ausgang hat den Datentyp “Text” oder umgekehrt.
- Der Eingang hat den Datentyp “Pfad” und der Ausgang hat den Datentyp “Text” oder umgekehrt.

- Der Eingang hat den Datentyp “Pfad” und der Ausgang hat den Datentyp “Dateipfad” oder umgekehrt.
- Der Eingang hat den Datentyp “Mathematisches Objekt” bzw. leitet von dessen Basisklasse ab und der Ausgang hat den Datentyp Zahl bzw. leitet von dessen Basisklasse ab oder umgekehrt.

## Kontexte

Ein Kontext stellt eine Umgebung dar, in welcher Elemente ausgeführt werden und Daten weiter gereicht werden können. In einem Kontext wird ein Element nie oder genau einmal ausgeführt. Dies bedeutet, dass wenn ein Ablauf wiederholt werden soll, mehrere Kontexte erzeugt werden müssen. Kontexte können terminieren, müssen dies jedoch nicht. Ein Kontext gilt als terminiert, wenn sich keine aktiven Elemente mehr in diesem befinden.

Innerhalb eines Kontextes können Elemente *angestoßen* werden. Angestoßene Elemente versuchen von ihrem Zustand 1 in den Zustand 5 zu kommen (siehe Abschnitt *Zustände von Elementen*). Elemente, die sich bereits in einem Zustand größer als 1 befinden, ignorieren diese Aufforderung.

Liefern Elemente Daten über ihre Ausgänge weiter, so werden diese innerhalb des Kontextes gespeichert und stehen zum Abruf durch ein mit diesem Ausgang verbundenen Eingang bereit. Treffen Daten von einem Ausgang ein, werden automatisch alle Elemente angestoßen, die mit ihren Eingängen an diesem Ausgang verbunden sind.

## Zustände von Elementen

Es gibt fünf Zustände, in denen sich Elemente befinden können:

- Zustand 1 *Nicht ausgeführt* (inaktiv)
- Zustand 2 *Warten*(aktiv)
- Zustand 3 *Arbeiten* (aktiv)
- Zustand 4 *Ausliefern*(aktiv)
- Zustand 5 *Terminiert* (inaktiv)

Der Zustand eines Elementes betrifft immer nur einen bestimmten Kontext. Ein Element kann sich beispielsweise in einem Kontext im Zustand 2 und gleichzeitig in einem anderen Kontext im Zustand 4 befinden. Zustände können immer nur von Zustand 1 nach Zustand 5 durchlaufen werden (Zustände dürfen übersprungen werden). Ein Element bewegt sich nie in einen vorherigen Zustand zurück. Ein Element gilt in einem Kontext als aktiv, wenn es sich in diesem Kontext im Zustand 2, 3 oder 4 befindet.

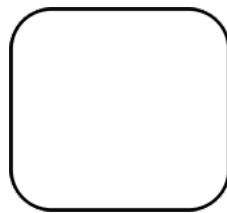
## Elemente

Elemente bestehen jeweils aus einem Namen, einem Icon, einer Beschreibung, einer Liste von Eingängen und einer Liste von Ausgängen.

Es gibt drei verschiedene Arten von Elementen: *Einfache Elemente*, *Spezielle Elemente* und *Kontext erzeugende Elemente*

Im Folgenden gehe ich nur auf bestimmte – wichtige – Elemente ein. Eine vollständige Liste aller implementierten Elemente steht auf GitHub im Verzeichnis „*Dokumentation Element Definitionen*“ zur Verfügung.

### *Einfache Elemente*



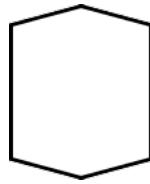
**Abb. 2.1.** Einfaches Element

Diese Elemente finden am häufigsten Verwendung. Sie repräsentieren Tätigkeiten, die abgearbeitet werden, sodass dann fortgefahrene werden kann. Alle diese Elemente terminieren (durch Erfolg oder durch Abbruch bei einem Fehler). Die Ausführung dieser Elemente ist jeweils einem bestehenden Kontext zugeordnet. Erkennbar sind diese Elemente an ihren abgerundeten Kanten (Abb. 2.1). Einfache Elemente gehen hierbei in 4 Schritten vor:

1. Anstoßen aller vorhergehenden Elemente (an Eingängen verbundene Elemente). (Zustand 2)
2. Warten bis die Daten aller Eingänge im Kontext abrufbar sind.
3. Abarbeiten ihrer spezifischen Tätigkeit mit den Daten an den Eingängen als Parameter. (Zustand 3)
4. Liefern der Daten an alle verbunden Elemente

### *Spezielle Elemente*

Diese Elemente verhalten sich prinzipiell wie *einfache Elemente*, haben jedoch immer jeweils mindestens eine Eigenschaft welche diese von *einfachen Elementen* unterscheidet, sodass diese von der Ausführungsumgebung gesondert behandelt werden. Erkennbar sind diese Elemente an ihren sechs Kanten (Abb. 2.2). Um die Funktionsweise dieser speziellen Elemente zu verstehen, ist es nötig diese im



**Abb. 2.2.** Spezielles Element

Einzelnen zu betrachten.

Element *Wenn-Vor*:

1. Anstoßen aller vorhergehenden Elemente (an Eingängen verbundene Elemente). (Zustand 2)
2. Warten bis die Daten aller Eingänge im Kontext abrufbar sind. (Zustand 2)
3. Liefern der Daten von Eingang “Objekt” an Ausgang “Falls Wahr” falls an Eingang “Wahrheitswert” **true** anliegt und an Ausgang “Falls Unwahr” sonst. (Zustand 4, Zustand 3 übersprungen)

Element *Wenn-Zurück*:

1. Anstoßen des vorhergehenden Element von Eingang “Wahrheitswert”. (Zustand 2)
2. Anstoßen des vorhergehenden Element von Eingang “Falls Wahr” falls an Eingang “Wahrheitswert” **true** anliegt und Anstoßen des vorhergehenden Element von Eingang “Falls Unwahr” sonst. (Zustand 2)
3. Liefern der Daten an verbundenen Ausgang “Objekt”. (Zustand 4, Zustand 3 übersprungen)

Element *Schnellster Wert*:

1. Anstoßen aller vorhergehenden Elemente (an Eingängen verbunden). (Zustand 2)
2. Warten bis die Daten **eines beliebigen Eingangs** im Kontext abrufbar sind. (Zustand 2)
3. Liefern der Daten dieses Eingangs an verbundenen Ausgang “Wert”. (Zustand 4, Zustand 3 übersprungen)

Element *Ein Wert*:

1. Warten bis die Daten **eines beliebigen Eingangs** im Kontext abrufbar sind (die vorhergehenden Elemente werden nicht angestoßen). (Zustand 2)
2. Liefern der Daten an verbundenen Ausgang “Wert”. (Zustand 4, Zustand 3 übersprungen)

*Element Für Alle und Zusammenführen:*

Diese Elemente gehören zusammen. Das *Für Alle*-Element teilt eine Liste in ihre einzelnen Elemente auf und erzeugt für jedes einen eigenen Kontext um dieses abzuarbeiten. Das Zusammenführen-Element wiederum führt diese Elemente wieder zusammen auf den ursprünglichen Kontext.

*Kontext erzeugende Elemente*



**Abb. 2.3.** Kontext erzeugendes Element

*Kontext erzeugende Elemente* dienen – wie der Name bereits sagt – dazu Kontexte zu erzeugen. Ihre Ausführung kann, muss jedoch nicht terminieren. Die Ausführung dieser Elemente ist jeweils einem eigenen Kontext zugeordnet, der nur dazu dient diese und evtl. vorhergehende Elemente am Laufen zu halten. Erkennbar sind diese Elemente an ihren vier nicht abgerundeten Kanten (Abb. 2.3).

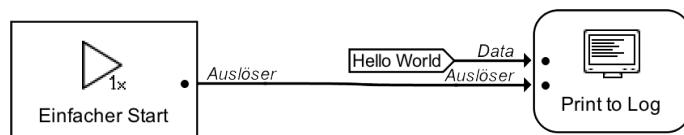
Kontext erzeugende Elemente gehen hierbei in folgenden Schritten vor:

1. Anstoßen aller vorhergehenden Elemente innerhalb von ihrem eigenen Kontext.  
(Zustand 2)
2. Warten bis die Daten aller Eingänge in ihrem eigenen Kontext abrufbar sind.  
(Zustand 2)
3. Starten der eigenen Tätigkeit.
4. Starten von Kind-Kontexten und Übergabe von Daten an die über die Ausgänge verbundenen Elemente innerhalb von den Kind-Kontexten.
5. Schritt 4 beliebig oft (nie, einmal oder mehrmals) wiederholen.

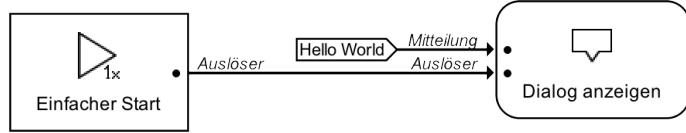
### 2.2.2 Beispiele

#### Hello World

Als Erstes widme ich mich dem simplesten und vermutlich ersten Programm eines jeden Programmierer.

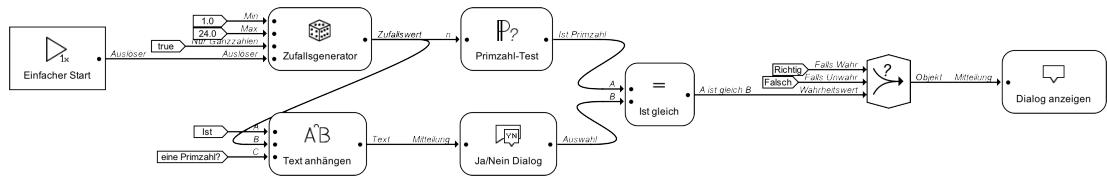


In einer anderen Variante des *Hello World Beispiels* verwenden wir keine Konsolen-Ausgabe, sondern einen Dialog:



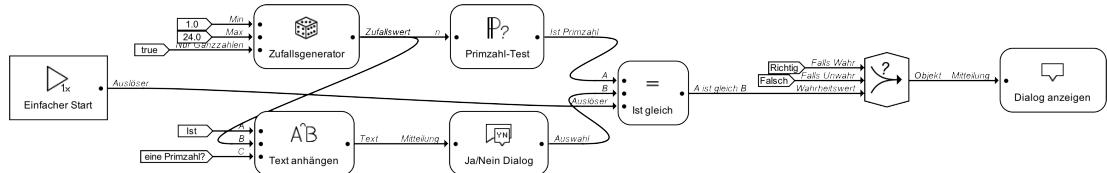
## Primzahl-Quiz

Beim nächsten Beispiel integrieren wir eine Interaktion vom Benutzer mit dem Programm. Bei unserem Programm wird eine zufällige Zahl gewürfelt und berechnet, ob es sich bei dieser Zahl um eine Primzahl handelt. Gleichzeitig wird der Benutzer gefragt ob es sich bei der berechneten Zahl um eine Primzahl handelt. Am Ende bekommt der Benutzer einen Dialog angezeigt, der zeigt, ob er richtig getippt hat (Abb. 2.4).

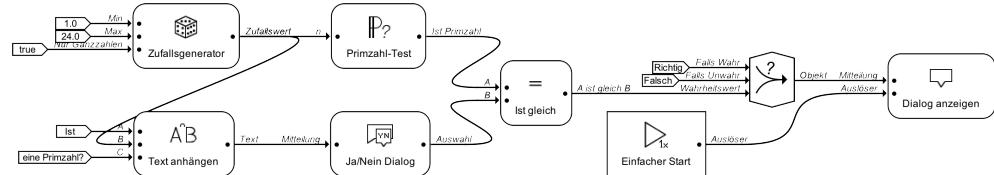


Interessant bei diesem Beispiel ist, dass es hier irrelevant ist mit welchem Element das Start-Element verbunden ist.

Start-Element ist mit “Ist gleich”-Element verbunden:

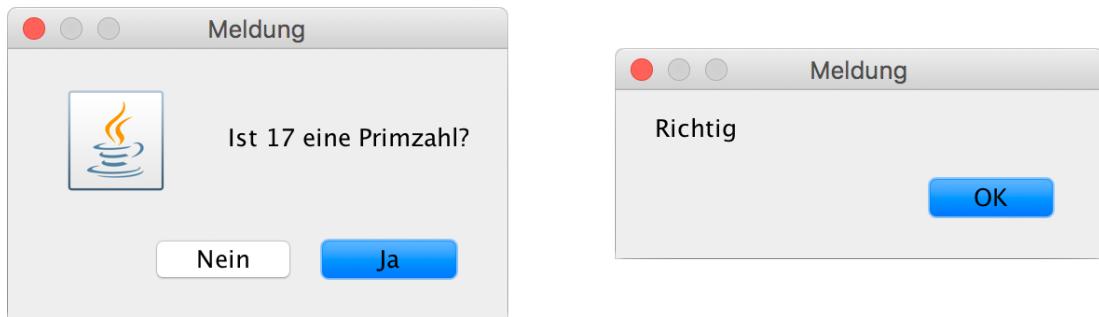


Start-Element ist mit “Dialog anzeigen”-Element verbunden:



Beim Versuch dieser drei Programme fällt auf, dass sich die Dialoge jeweils in der richtigen Reihenfolge öffnen, auch wenn das Start-Element mit dem letzten Element verbunden wird. Dies liegt daran dass Elemente immer bis zum Eingang der

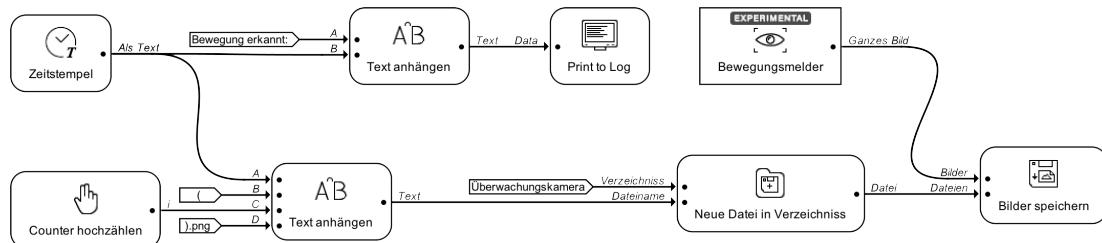
Daten aller Eingänge warten. Aus diesem Grund ist es in den meisten Situationen irrelevant, mit welchem Element der Auslöser des *Kontext Erzeugenden Element* verbunden ist.



**Abb. 2.4.** Dialoge, die der Benutzer beim Primzahl-Quiz angezeigt bekommt

### Bewegungsmelder

Das folgende Programm implementiert einen einfachen Bewegungsmelder. Immer wenn sich etwas stark vor der Webcam bewegt, wird eine Meldung mit dem aktuellen Zeitpunkt ausgegeben und ein Schnappschuss als Datei abgespeichert (Abb. 2.5).



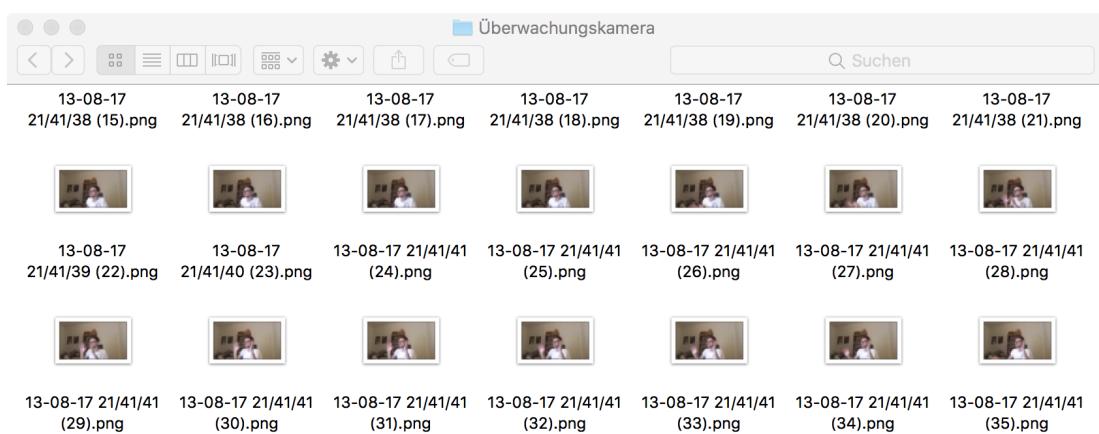
Ähnlich wie beim Primzahl-Quiz ist es hier wieder nicht entscheidend, welches Element vom Kontext-Erzeugenden-Element angestoßen wird. Interessant an diesem Beispiel ist es jedoch, zu sehen dass – obwohl nur ein Bild an das “Bild speichern”-Element übergeben wird – es trotzdem auch zu einer Ausgabe in der Konsole kommt. Dies liegt daran, dass der Zeitstempel Daten an das “Text anhängen”-Element des Bildspeichern-Zweigs abgibt auch automatisch seine Daten an das “Text anhängen”-Element des Konsolenausgaben-Zweigs abgibt und diesen damit anstößt.

Ausgabe (gekürzt):

```
Bewegung erkannt: 13-08-17 21:41:35
Bewegung erkannt: 13-08-17 21:41:36
Bewegung erkannt: 13-08-17 21:41:36
```

```
Bewegung erkannt: 13-08-17 21:41:36
Bewegung erkannt: 13-08-17 21:41:37
Bewegung erkannt: 13-08-17 21:41:37
Bewegung erkannt: 13-08-17 21:41:37
Bewegung erkannt: 13-08-17 21:41:38
...

```



**Abb. 2.5.** Inhalt des Ordners “Überwachungskamera”

### Fizz buzz

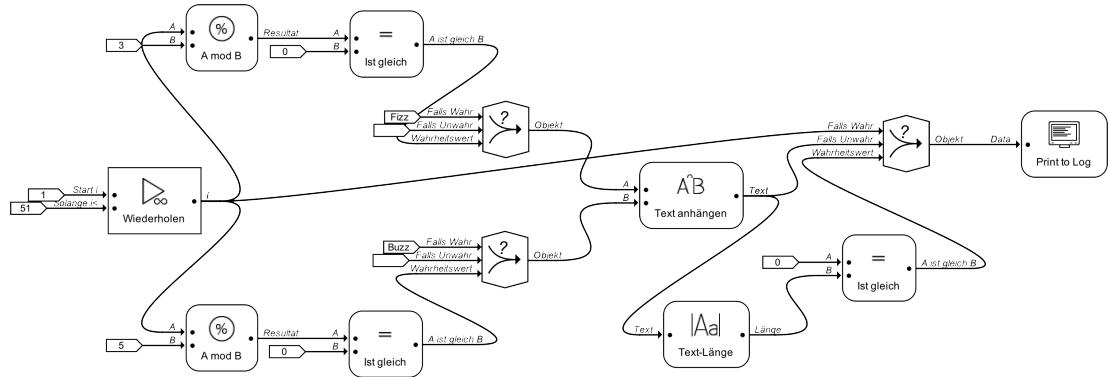
Fizz buzz ist ein Spiel, das erfunden wurde, um Kindern die Divisionsregeln beizubringen. Das Spiel hat jedoch einen gewissen Bekanntheitsgrad erreicht, da es dazu verwendet werden soll, die Programmierkenntnisse von Bewerbern im Vorstellungsgesprächen zu prüfen<sup>4</sup>

Die Spielregeln laut Wikipedia: *“Players generally sit in a circle. The player designated to go first says the number “1”, and each player counts one number in turn. However, any number divisible by three is replaced by the word fizz and any divisible by five by the word buzz. Numbers divisible by both become fizz buzz. A player who hesitates or makes a mistake is eliminated from the game. For example, a typical round of fizz buzz would start as follows: 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizz Buzz, 16, 17, Fizz, 19, Buzz, Fizz, 22, 23, Fizz, Buzz, 26, Fizz, 28, 29, Fizz Buzz, 31, 32, Fizz, 34, Buzz, Fizz...”*

<sup>4</sup> <https://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/>

Grundsätzlich bedeutet dies, dass eine Reihe von Zahlen beginnend mit der 1 aufgesagt wird und dabei jede Zahl, die durch 3 teilbar ist, durch Fizz und jede Zahl, die durch 5 teilbar ist, durch Buzz ersetzt wird. Ist die Zahl sowohl durch 3, als auch durch 5 teilbar, so wird diese durch FizzBuzz ersetzt. Das folgende Programm soll Fizz buzz von 1 bis 50 ausgeben.

*Erster Ansatz mit Hilfe eines Wiederholen-Elementes*

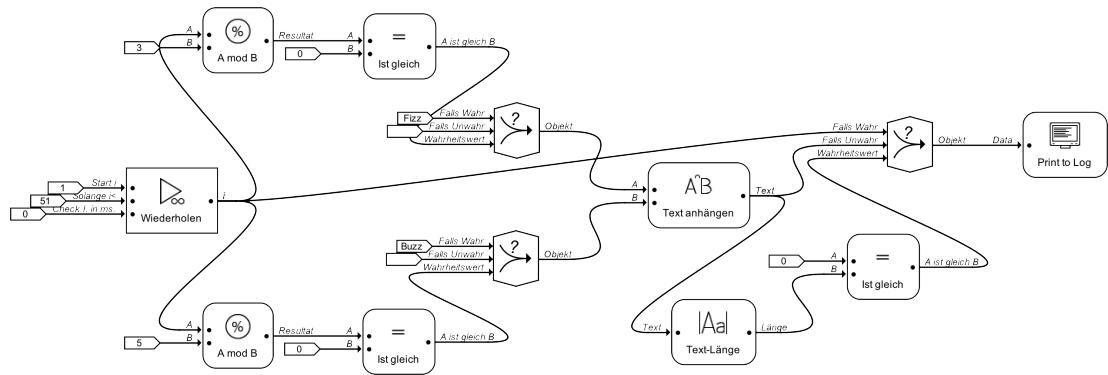


Ausgabe (gekürzt und Zeilenumbrüche wurde gegen Kommas getauscht):

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizz  
Buzz, 16, 17, Fizz, 19, Buzz, Fizz, ...

Ausführungszeit: **5.182 ms**

Die Ausgabe ist wie erwartet jedoch ist die Ausführungszeit für eine relativ leichte Arbeit sehr hoch. Dies liegt hauptsächlich an der Verwendung des Wiederholen-Elements. Das Wiederholen-Element kontrolliert jede 100 ms, ob der zuletzt erzeugte Kontext bereits fertig ist und falls ja erzeugt es den nächsten Kontext. Um das Programm zu optimieren, lässt sich die Zeit, die zwischen den Kontrollen liegt, manuell einstellen. Im folgenden Aufbau setzen wir die Zeit auf 0 ms, es wird also aktiv auf das Ende jedes Kontextes gewartet (dies ist zwar rechenaufwändiger, aber in diesem Fall ist dies zu vernachlässigen, da die Rechenzeit insgesamt kürzer wird).



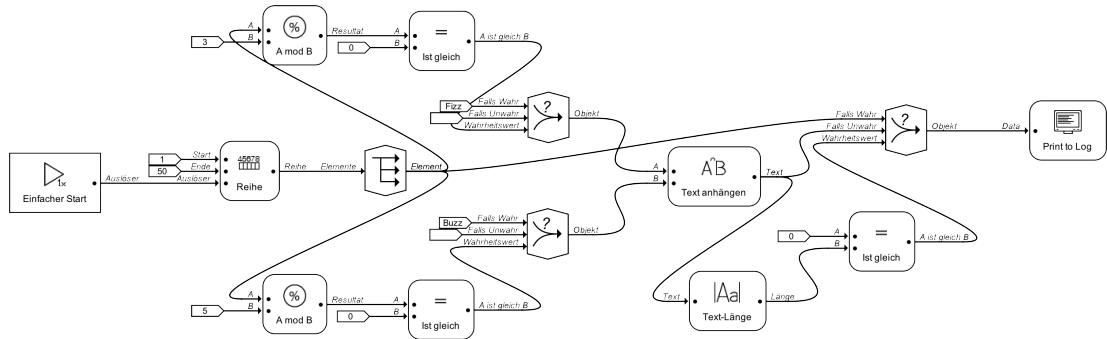
Ausgabe (gekürzt und Zeilenumbrüche wurde gegen Kommas getauscht):

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizz  
Buzz, 16, 17, Fizz, 19, Buzz, Fizz, ...

Ausführungszeit: **135 ms**

Zweiter Ansatz mit Hilfe einer Für-Alle Struktur

Eine alternative Möglichkeit das Fizz buzz Programm umzusetzen ohne ein Wiederholen-Element zu verwenden (dies ist teilweise mit Nachteilen verbunden oder bedingt durch die Struktur nicht möglich) ist mit Hilfe einer Für-Alle Struktur.

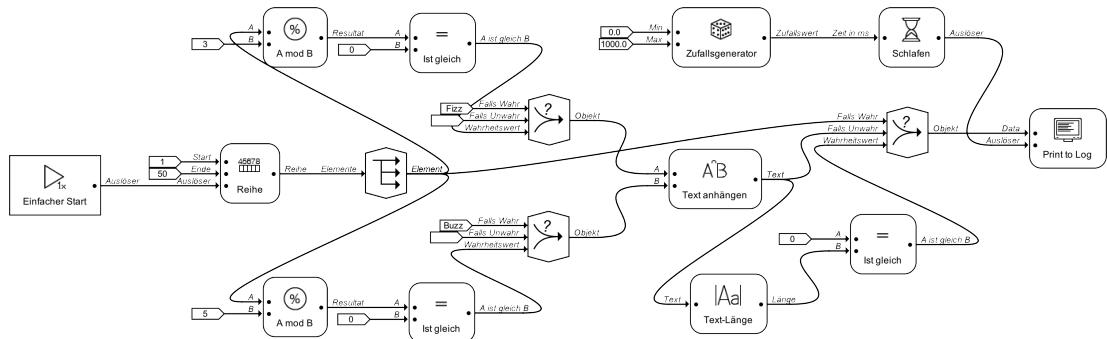


Ausgabe (gekürzt und Zeilenumbrüche wurde gegen Kommas getauscht):

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizz  
Buzz, 16, 17, Fizz, 19, Buzz, Fizz, ...

Ausführungszeit: **124 ms**

Das Ergebnis ist wieder das Gleiche und auch die Ausführungszeit ist vergleichbar, jedoch ist der Aufbau in dieser Form theoretisch **falsch**. Um dies zu verdeutlichen bauen wir zwei weitere Elemente in unser Programm ein.

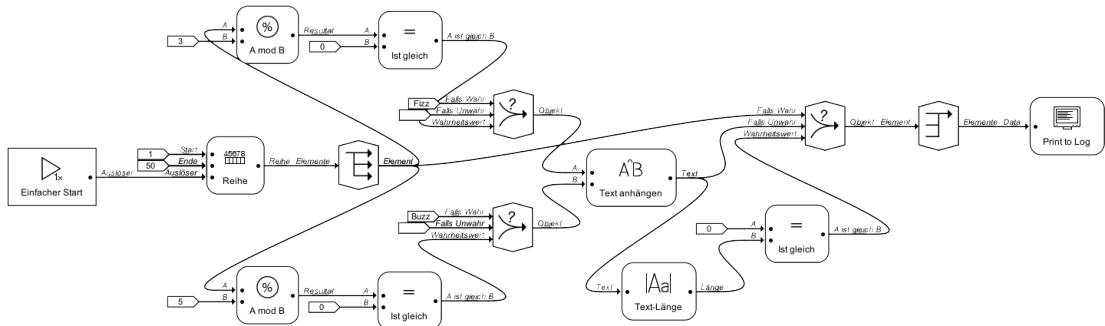


Ausgabe (gekürzt und Zeilenumbrüche wurde gegen Kommas getauscht):

29, Fizz, 37, Fizz, 38, 4, Buzz, 46, 43, 28, Fizz, 17, 1, 19, 26,  
8, Buzz, Fizz, 13, 32, Fizz, 31, ...

Ausführungszeit: **1430 ms** (durch künstliche Verzögerung)

Wie zu erkennen ist, stimmen die einzelnen Ausgaben zwar, befinden sich jedoch nicht in der richtigen Reihenfolge. Dies liegt daran, dass Für-Alle Strukturen keine Reihenfolgen kennen und alle Elemente parallel abgearbeitet werden. Da durch die künstliche Verzögerung die Kontexte unterschiedlich viel Zeit benötigen, wird das Ausgabe-Element in der falschen Reihenfolge angestoßen. Ist die Reihenfolge der Elemente entscheidend, empfiehlt es sich zusätzlich ein Zusammenführen-Element einzusetzen.



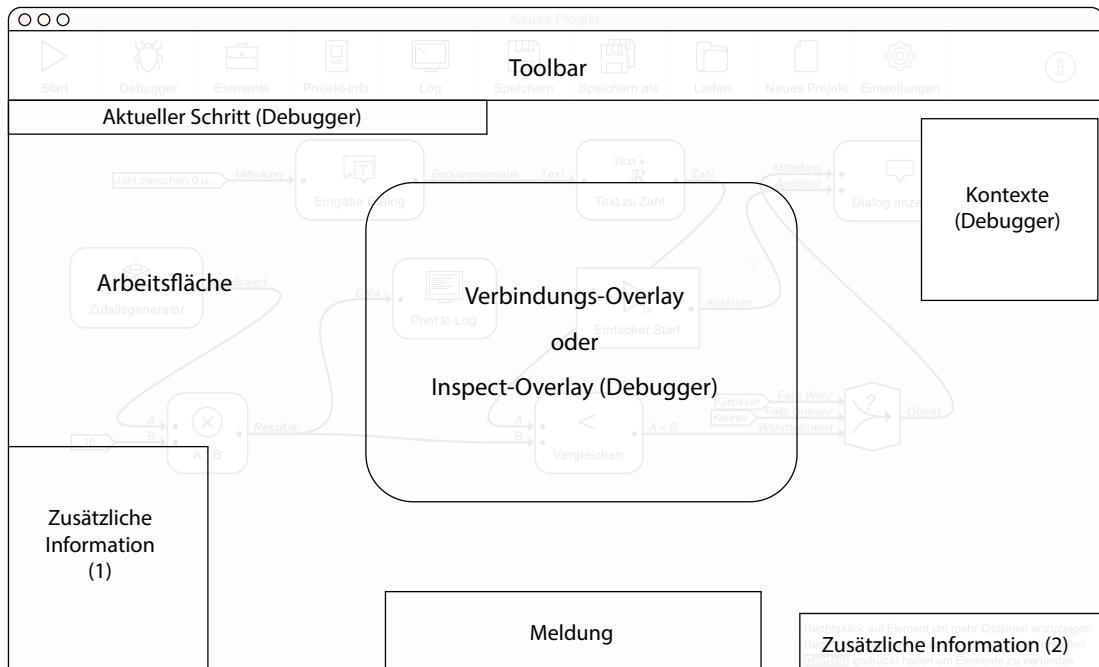
Ausgabe (gekürzt und Zeilenumbrüche wurde gegen Kommas getauscht):

1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, Fizz  
Buzz, 16, 17, Fizz, 19, Buzz, Fizz, ...

Ausführungszeit: **121 ms**

### 2.2.3 Benutzerinterface

#### Begriffe Hauptfenster



#### Zustände Hauptfenster

Grundlegend unterscheidet das Benutzerinterface des Hauptfensters zwischen folgenden Zuständen:

Zustand	Beschreibung	Kontextmenüs	Zusätzliche Informationen
1a	Projekt wird nicht ausgeführt	Ja	Ja
1b	Projekt wird nicht ausgeführt und es wird Verbindung zwischen Elementen gezogen	Nein	Ja
1c	Projekt wird nicht ausgeführt und Overlay um Elemente zu verbinden ist geöffnet	Nein	Ja
1d	Projekt wird nicht ausgeführt, Overlay um Elemente zu verbinden ist geöffnet und es wird Verbindung zwischen Ein-/Ausgängen gezogen	Nein	Ja <sup>5</sup>
2	Projekt wird ausgeführt	Nein	Nein

Zustand	Beschreibung	Kontextmenüs	Zusätzliche Informationen
2x	Projekt wird nicht mehr ausgeführt	Nein	Nein
3a	Projekt wird ausgeführt mit Debugger	Nein	Ja <sup>6</sup>
3b	Projekt wird ausgeführt mit Debugger und Element wird eingesehen	Nein	Ja <sup>6</sup>
3x	Projekt wird nicht mehr ausgeführt mit Debugger	Nein	Nein

Je nach Zustand stellt das Benutzerinterface andere Funktionen zur Verfügung und stellt andere Informationen dar. Zu beachten ist, dass die in der Tabelle beschriebenen Zustände **nur** das Hauptfenster betreffen und andere Fenster von diesen nicht beeinflusst werden.

### Toolbar Hauptfenster

Die Toolbar (Abb. 2.6) stellt dem Benutzer dynamisch Funktionalitäten zur Verfügung. Je nach Zustand der Benutzeroberfläche werden unterschiedliche Einträge dargestellt. Auch ist es je nach Eintrag möglich, dass Untereinträge dargestellt werden, diese können mittels einem *Verlassen*-Eintrag wieder geschlossen werden, womit der Benutzer wieder zur vorherigen Ansicht zurück gelangt.



Abb. 2.6. Toolbar

Welche Einträge in der Toolbar angezeigt werden, entscheidet sich hauptsächlich daran, in welchem Zustand das Benutzerinterface sich gerade befindet. Folgende Einträge werden in der Toolbar angezeigt:

<sup>5</sup> Entspricht Informationen von Zustand 1c.

<sup>6</sup> Anstelle von einer Information darüber, welche Möglichkeiten der Benutzer gerade hat, werden Informationen über die Zustände der einzelnen Elemente im Debugger angezeigt.

Icon	Name	Sichtbare Zustände	Beschreibung	Besonderheit
	Start	1a, 1b, 1c, 1d	Startet Ausführung von aktuellem Projekt.	
	Elemente	1a, 1b, 1c, 1d	Öffnet Auswahlfenster für neue Elemente.	
	Projekt-Info	1a, 1b, 1c, 1d	Zeigt Informationen zum aktuellen Projekt an.	
	Konsole/Log	1a, 1b, 1c, 1d, 2, 2x, 3a, 3b, 3x (Jeder Zustand)	Öffnet Fenster für System Ein-/Ausgaben.	Je nachdem ob der Prompt aktiviert ist wird hier Konsole (aktiver Prompt) oder Log (Kein Prompt) angezeigt
	Speichern	(1a) <sup>8</sup> , (1b) <sup>8</sup> , (1c) <sup>8</sup> , (1d) <sup>8</sup>	Speichert aktuelles Projekt unter dem aktuellen Pfad.	Wird nur angezeigt, wenn Projekt nicht bereits gespeichert ist.
	Speichern als	1a, 1b, 1c, 1d	Speichert aktuelles Projekt über Auswahl Dialog unter einem neuen Pfad.	
	Laden	1a, 1b, 1c, 1d	Öffnet bestehendes Projekt über ein Auswahl Dialog.	
	Neues Projekt	1a, 1b, 1c, 1d	Legt neues Projekt an. Falls aktuelles Projekt nicht gespeichert ist, wird ein Speicher-Dialog angezeigt.	

Icon	Name	Sichtbare Zustände	Beschreibung	Besonderheit
	Einstellungen	1a, 1b, 1c, 1d	Öffnet Fenster für Einstellungen.	
	Debugger	1a, 1b, 1c, 1d	Stellt Funktionen des Debuggers zur Verfügung und zeigt folgende Untereinträge an: Debug, Breakpoint Auslöser, Monitor, Log.	
	Debug	(1a) <sup>7</sup> , (1b) <sup>7</sup> , (1c) <sup>7</sup> , (1d) <sup>7</sup>	Startet Ausführung von aktuellem Projekt mit aktiviertem Debugger.	
	Monitor	3a, 3b	Öffnet Fenster für Monitoring.	
	Breakpoint Auslöser	(1a) <sup>7</sup> , (1b) <sup>7</sup> , (1c) <sup>7</sup> , (1d) <sup>7</sup> , 3a, 3b	Stellt eine Reihe von Toogle-Einträge zur Verfügung.	
	Breakpoint Auslöser: Neuer Prozess	1a, 1b, 1c, 1d, 3a, 3b	Entscheidet, ob Breakpoint bei neuem Prozess ausgelöst werden soll.	Toogle-Eintrag, Untereintrag von Breakpoint Auslöser
	Breakpoint Auslöser: Vorbereiten	(1a) <sup>7</sup> , (1b) <sup>7</sup> , (1c) <sup>7</sup> , (1d) <sup>7</sup> , (3a) <sup>7</sup> , (3b) <sup>7</sup>	Entscheidet, ob Breakpoint bei Zustandswechsel zu "Vorbereiten" ausgelöst werden soll.	Toogle-Eintrag, Untereintrag von Breakpoint Auslöser
	Breakpoint Auslöser: Sammeln	(1a) <sup>7</sup> , (1b) <sup>7</sup> , (1c) <sup>7</sup> , (1d) <sup>7</sup> , (3a) <sup>7</sup> , (3b) <sup>7</sup>	Entscheidet, ob Breakpoint bei Zustandswechsel zu "Sammeln" ausgelöst werden soll.	Toogle-Eintrag, Untereintrag von Breakpoint Auslöser

Icon	Name	Sichtbare Zustände	Beschreibung	Besonderheit
	Breakpoint Auslöser: Arbeiten	(1a) <sup>7</sup> , (1b) <sup>7</sup> , (1c) <sup>7</sup> , (1d) <sup>7</sup> , (3a) <sup>7</sup> , (3b) <sup>7</sup>	Entscheidet, ob Breakpoint bei Zustandswechsel zu "Arbeiten" ausgelöst werden soll.	Toogle-Eintrag, Untereintrag von Breakpoint Auslöser
	Breakpoint Auslöser: Ausliefern	(1a) <sup>7</sup> , (1b) <sup>7</sup> , (1c) <sup>7</sup> , (1d) <sup>7</sup> , (3a) <sup>7</sup> , (3b) <sup>7</sup>	Entscheidet, ob Breakpoint bei Zustandswechsel zu "Ausliefern" ausgelöst werden soll.	Toogle-Eintrag, Untereintrag von Breakpoint Auslöser
	Breakpoint Auslöser: Beendet	(1a) <sup>7</sup> , (1b) <sup>7</sup> , (1c) <sup>7</sup> , (1d) <sup>7</sup> , (3a) <sup>7</sup> , (3b) <sup>7</sup>	Entscheidet, ob Breakpoint bei Zustandswechsel zu "Beendet" ausgelöst werden soll.	Toogle-Eintrag, Untereintrag von Breakpoint Auslöser
	Stop	2, 3a, 3b	Bricht Ausführung von Projekt ab.	
	Schritt	3a, 3b	Schritt bis zum nächsten Auslöser (nächstes Element, unabhängig davon, ob dieses einen gesetzten Breakpoint hat).	Eine Besonderheit hierbei ist, dass Schritt auch ausgelöst werden kann, ohne dass gerade gehalten wird.
	Fortsetzen	(3a) <sup>8</sup> , (3b) <sup>8</sup>	Fortsetzen bis zum nächsten Breakpoint.	Nur sichtbar, wenn gerade an Breakpoint gehalten wird.

Icon	Name	Sichtbare Zustände	Beschreibung	Besonderheit
	Weiter ohne Debugger	(3a) <sup>8</sup> , (3b) <sup>8</sup>	Fortsetzen ohne am nächsten Breakpoint zu halten.	Nur sichtbar, wenn gerade an Breakpoint gehalten wird.
	Weiter mit Debugger	(3a) <sup>8</sup> , (3b) <sup>8</sup>	Deaktiviert Debugger temporär.	Nur sichtbar, wenn gerade Debugger deaktiviert ist.
	Bericht	2x, 3x	Öffnet Fenster mit Bericht über gerade beendete Ausführung.	
	Fertig	2x, 3x	Führt von Zustand 2x oder 3x wieder in Zustand 1	

<sup>7</sup> Als Untereintrag.<sup>8</sup> Situationsbedingt.

## Arbeitsfläche Hauptfenster

Auf der Arbeitsfläche (Abb. 2.7) befinden sich alle aktuell im Projekt platzierten Elemente und es wird dargestellt, wie diese miteinander verbunden sind.

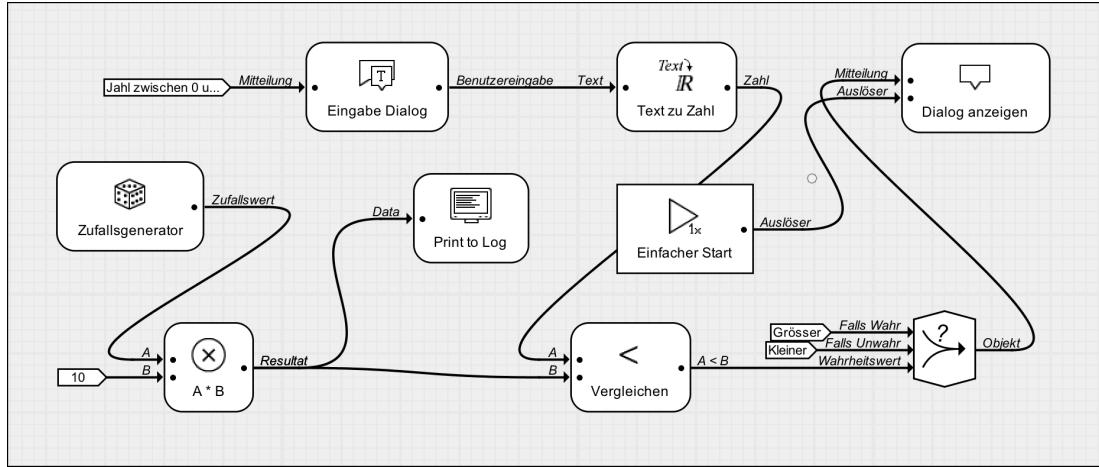


Abb. 2.7. Arbeitsfläche

Elemente auf der Arbeitsfläche können beliebig verschoben werden (Abb. 2.8). Bewegt sich ein Element rechts oder unten über die Arbeitsfläche hinaus, erweitert sich diese dynamisch.

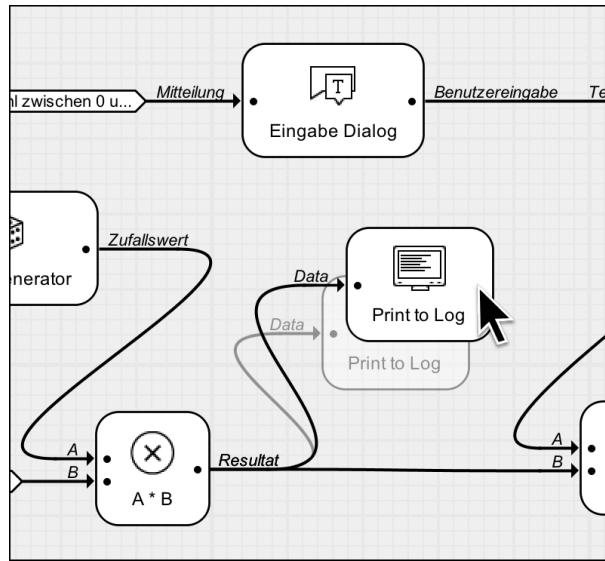
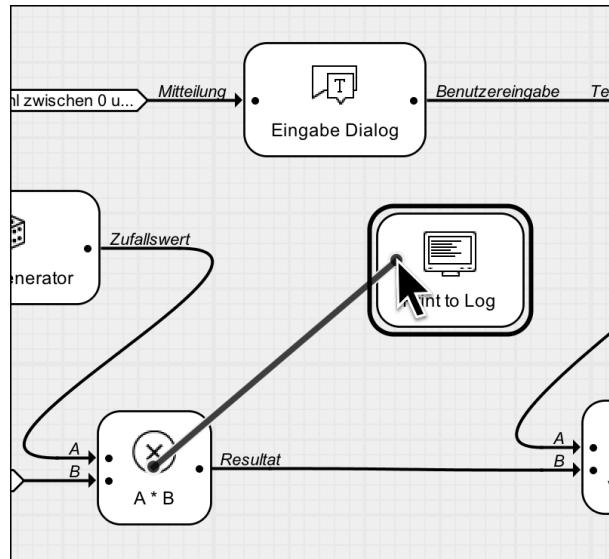


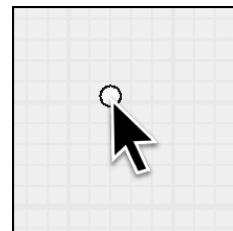
Abb. 2.8. Verschieben von Elementen

Um Elemente miteinander zu verbinden, muss ein Element mit gedrückter CTRL-Taste auf ein anderes Element gezogen werden (Abb. 2.9). Auf diese Aktion hin öffnet sich das Verbindungs-Overlay, um diese beiden Elemente miteinander zu verbinden.



**Abb. 2.9.** Verbinden von Elementen

Durch das Klicken auf einen freien Bereich auf der Arbeitsfläche wird diese Stelle als *Target-Position* festgelegt (Abb. 2.10). Die *Target-Position* definiert den Ort, an dem neue Elemente angelegt werden.



**Abb. 2.10.** Target-Position

### Verbindungs-Overlay Hauptfenster

Werden auf der Arbeitsfläche zwei Elemente mit gedrückter CTRL-Taste aufeinander gezogen, erscheint das Verbindungs-Overlay (Abb. 2.11). Auf dem Verbindungs-Overlay ist sichtbar, welche Ausgänge mit welchen Eingängen verbunden werden

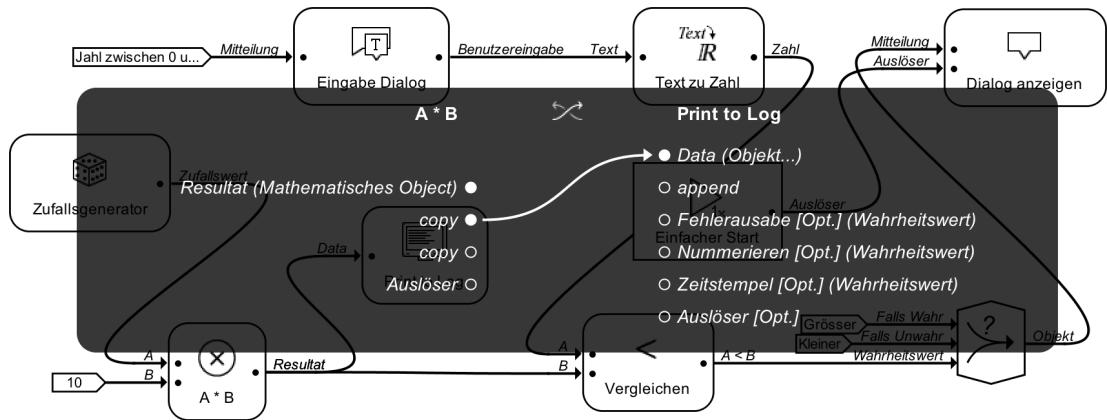


Abb. 2.11. Verbindungs-Overlay

können. Hierbei gibt die Kennzeichnung [Opt.] an, dass es sich bei diesem Eingang um einen optionalen Eingang handelt und die Kennzeichnung [Dyn.] gibt an, dass es sich um einen dynamisch hinzugefügten Eingang handelt. Außerdem ist hinter der Bezeichnung für den Ein-/Ausgang zusätzlich angegeben, welchen Datentyp jeder Ein-/Ausgang besitzt.

Auf der linken Seite befinden sich die Ausgänge des Elementes, von dem aus die Verbindungen gezogen werden (*Von-Element*) und auf der rechten Seite befinden sich die Eingänge des Elementes, auf das die Verbindung gezogen wird (*Zu-Element*). Um dies umzukehren, also das *Zu-Element* und *Von-Element* zu vertauschen, befindet sich ein Button  $\curvearrowleft$  in der Mitte zwischen den beiden Elementen. Ein-/Ausgänge, die mit einem ausgefüllten Kreis ● gekennzeichnet sind, sind bereits verbunden und Aus-/Eingänge, die mit einem nicht ausgefüllten Kreis ○ gekennzeichnet sind, sind nicht verbunden.

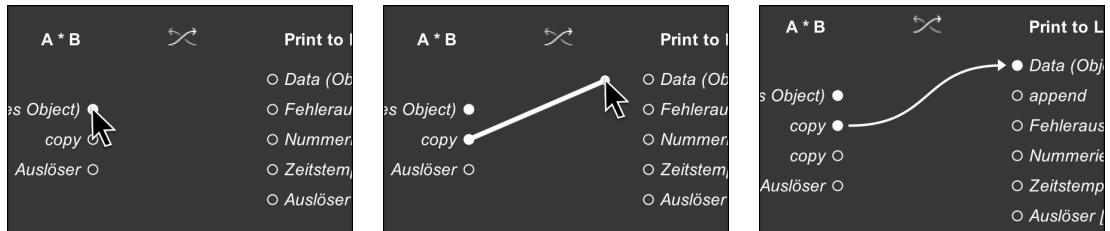


Abb. 2.12. Verbinden von einem Ausgang mit einem Eingang über das Verbindungs-Overlay

Um einen Ausgang mit einem Eingang zu verbinden, muss eins von den beiden auf das jeweils andere gezogen werden (Abb. 2.12). Falls ein Ein-/Ausgang bereits eine Verbindung besitzt, wird diese automatisch verworfen, wenn eine neue Verbin-

dung erstellt wird. Werden zwei Ein-/Ausgänge miteinander verbunden, die nicht kompatibel zueinander sind, wird eine Fehlermeldung angezeigt.

Um Verbindungen nur zu entfernen, reicht es, einen einfachen Klick auf den Namen des Ein-/Ausgänge auszuführen. Die Verbindung wird daraufhin verworfen und Direkte Eingabe Elemente gegebenenfalls automatisch gelöscht.

### Kontextmenüs Hauptfenster

Es gibt zwei verschiedene Arten von Kontextmenüs. Einträge in Kontextmenüs können Untermenüs haben.

#### *Auf leere Arbeitsfläche*

Dieses Kontextmenü zeigt eine Auswahl von Elementen, die erstellt werden können (Abb. 2.13). Hierbei ist das Erste immer das zuletzt erstellte Element. Der letzte Eintrag öffnet ein Auswahlfenster mit allen zur Verfügung stehenden Elementen.



Abb. 2.13. Kontextmenü (auf leere Arbeitsfläche)

### Auf Element

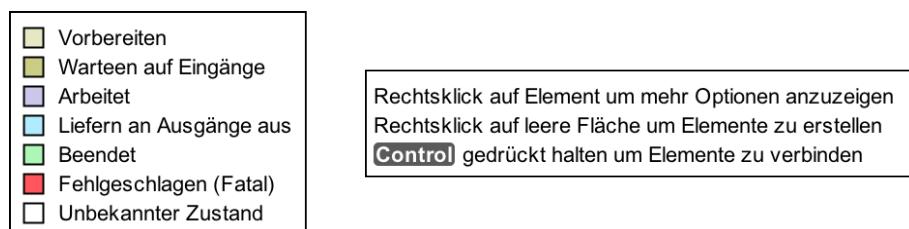
Dieses Kontextmenü zeigt eine Auswahl von Funktionen dieses Element betreffend (Abb. 2.14).



**Abb. 2.14.** Kontextmenü (auf Element)

### Anzeige von zusätzlichen Informationen im Hauptfenster

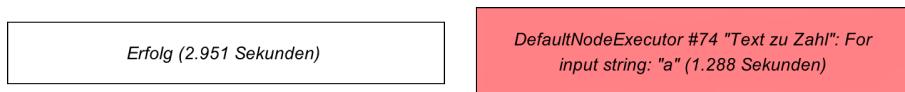
Situationsbedingt werden zusätzliche Informationen in der unteren rechten oder linken Ecke angezeigt (Abb. 2.15). Diese Informationen dienen in erster Linie dazu, auf Dinge hinzuweisen, die nicht intuitiv erkennbar sind (Bsp. Tasten, die zusätzlich gedrückt werden können).



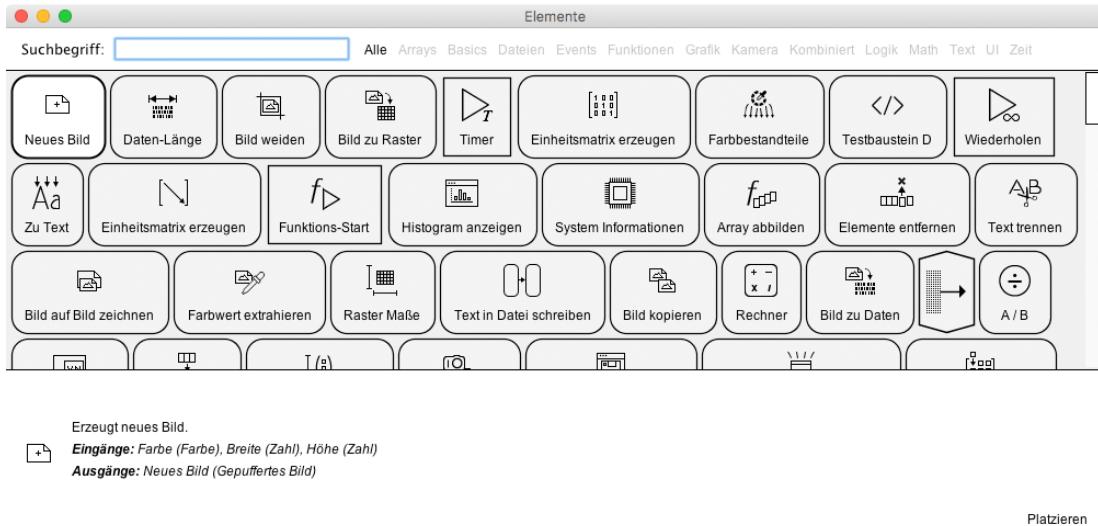
**Abb. 2.15.** Zusätzliche Informationen

### Anzeige von Meldungen im Hauptfenster

Meldungen über erfolgreiche oder fehlgeschlagene Operationen das Hauptfenster betreffend werden an der unteren Kante angezeigt (Abb. 2.16). Meldungen lassen sich durch das Drücken der ESC-Taste schließen.



**Abb. 2.16.** Meldungen im Hauptfenster



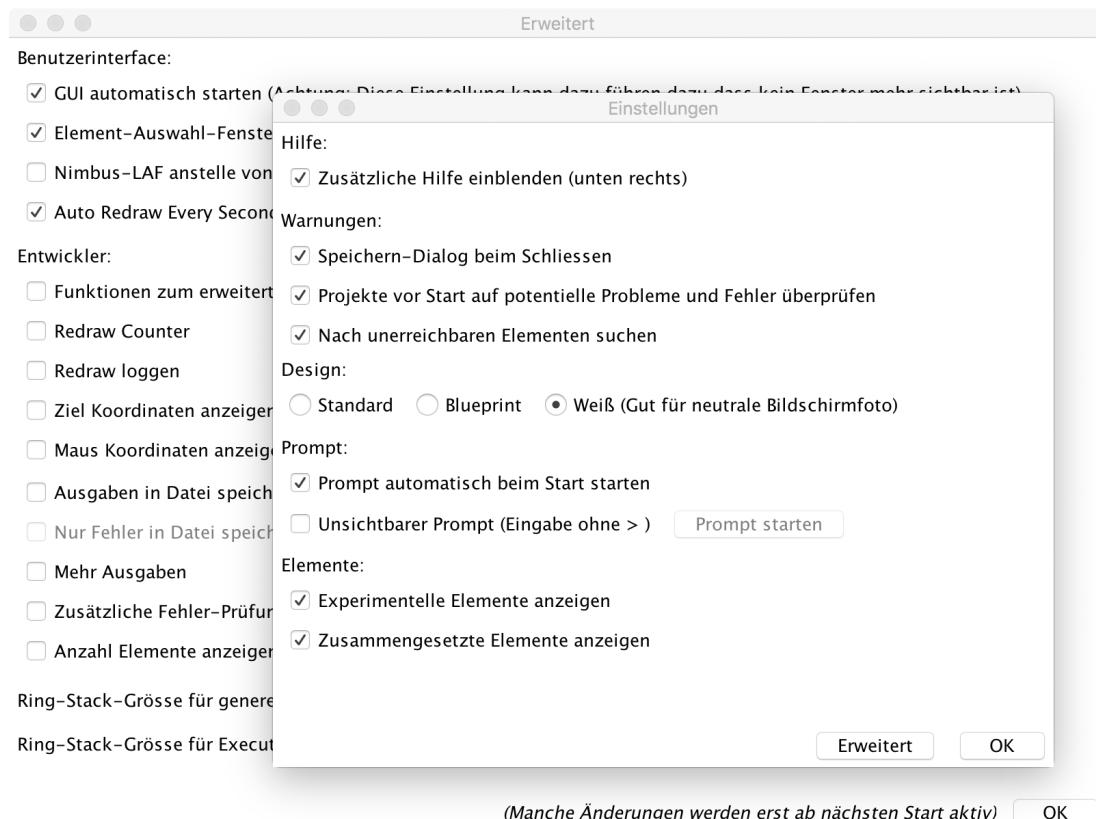
**Abb. 2.17.** Auswahlfenster für Elemente

#### 2.2.4 Auswahlfenster für Elemente

Über den Toolbar-Eintrag ‘‘Elemente’’ oder den Kontextmenü-Eintrag ‘‘Anderes Element erstellen’’ lässt sich das Auswahlfenster für neue Elemente öffnen (Abb. 2.17). In diesem werden alle verfügbaren Elemente, unterteilt in Kategorien, angezeigt. Hierbei ist zu beachten, dass Elemente keiner, einer oder mehreren Kategorien zugeordnet sein können. Um Elemente außerdem besser zu finden, gibt es zusätzlich ein Eingabefeld für Suchbegriffe. Hierbei funktionieren Suchbegriffe wie Filter. Elemente, bei denen in der Beschreibung oder in deren Tags keins der Worte vorkommen, nach denen gesucht wird, werden nicht angezeigt. Beim *einfachen Klick* auf eines der angezeigten Elemente öffnet sich am unteren Rand ein Beschreibungsfeld in welchem Informationen bezüglich dieses Elements dargestellt werden.

Um ein Element auf der Arbeitsfläche zu platzieren, gibt es zwei Möglichkeiten. Zum Einen kann der Benutzer einen Doppelklick auf eines der Elemente ausführen. Dieser bewirkt, dass das Auswahlfenster geschlossen und das Element an der angegebenen *Target-Position* angelegt wird. Alternativ kann der Benutzer den ‘‘Platzieren’’-Button am unteren rechten Rand verwenden. Dieser bewirkt, dass das Element an der *Target-Position* angelegt wird, das Auswahlfenster jedoch nicht geschlossen wird.

### 2.2.5 Fenster für Einstellungen



**Abb. 2.18.** Fenster für Einstellungen

Um Änderungen an den Voreinstellungen vorzunehmen, steht ein eigenes Fenster zur Verfügung (Abb. 2.18). In diesem lassen sich alle Einstellungen ändern, die für den normalen Benutzer relevant sind. Sollte der Benutzer doch mehr Möglichkeiten der Personalisierung benötigen, so kann dieser über den “Erweitert” Button ein weiteres Fenster öffnen, in dem mehr Einstellungen zur Verfügung stehen.

### 2.2.6 Fenster für System Ein-/Ausgaben (*Konsole*)

Für den einfachen Zugriff auf Ein-/Ausgaben, die dem Benutzer von grafischer Software normalerweise verborgen bleiben, gibt es ein eigenes Konsolen-Fenster (Abb. 2.19). Dieses lässt sich über den Toolbar-Eintrag “Konsole” / “Log” (je nachdem, ob der Prompt aktiviert ist) öffnen. In diesem Fenster werden alle Java-eigenen Ein-/Ausgaben angezeigt. Wenn der Prompt in den Einstellungen aktiviert ist, lassen sich hier auch Eingaben für die interne Kommandozeile tätigen.



Abb. 2.19. Konsole

### 2.2.7 Berichte

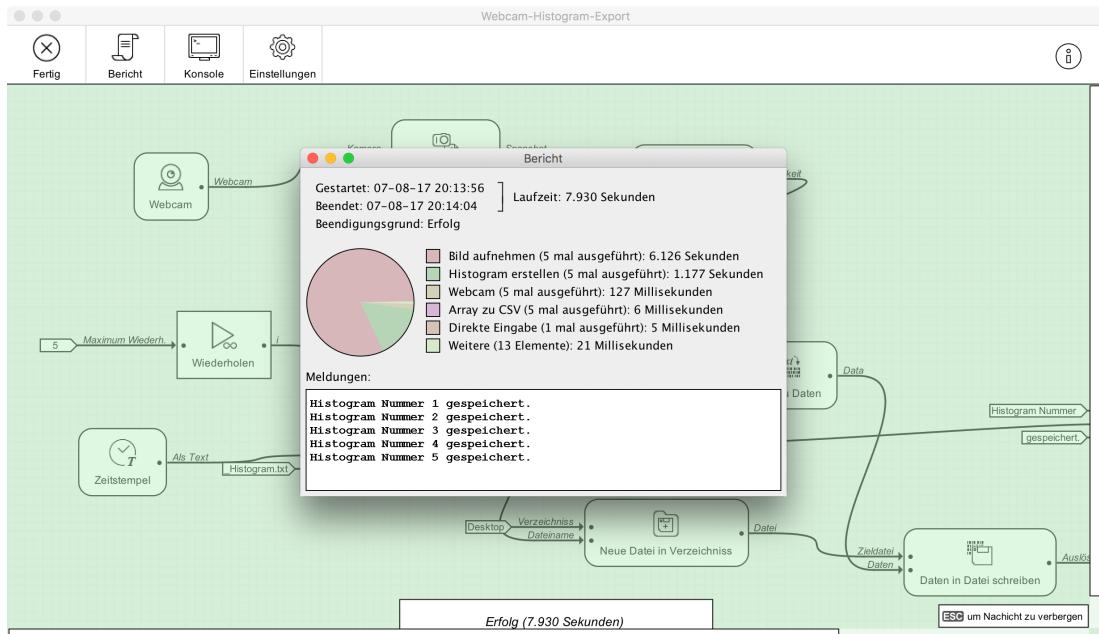


Abb. 2.20. Bericht

Nach erfolgreicher oder nicht erfolgreicher Ausführung eines Projektes hat der Benutzer die Möglichkeit, sich einen Bericht anzeigen zu lassen (Abb. 2.20).

Folgende Informationen sind in diesem Bericht enthalten:

- Startzeit
- Endzeit
- Laufzeit
- Meldungen
- Übersicht der Elemente, die am längsten ausgeführt wurden, mit passendem Tortendiagramm.

## 2.2.8 Debugger

### Haltemarken und Auslöser



Abb. 2.21. Haltemarke



Abb. 2.22. Auslöser Toolbar

Haltemarken (Abb. 2.21) und gesetzte Auslöser entscheiden darüber, ob und wann eine Projektausführung angehalten wird. Haltemarken können an alle Elemente gehängt werden, die selbst keine Kontexte erzeugen. Auslöser, also Zustandswechsel, die ein Halten auslösen, werden über die entsprechenden Einträge in der Toolbar (Abb. 2.22) festgelegt.

## Kennzeichnungen

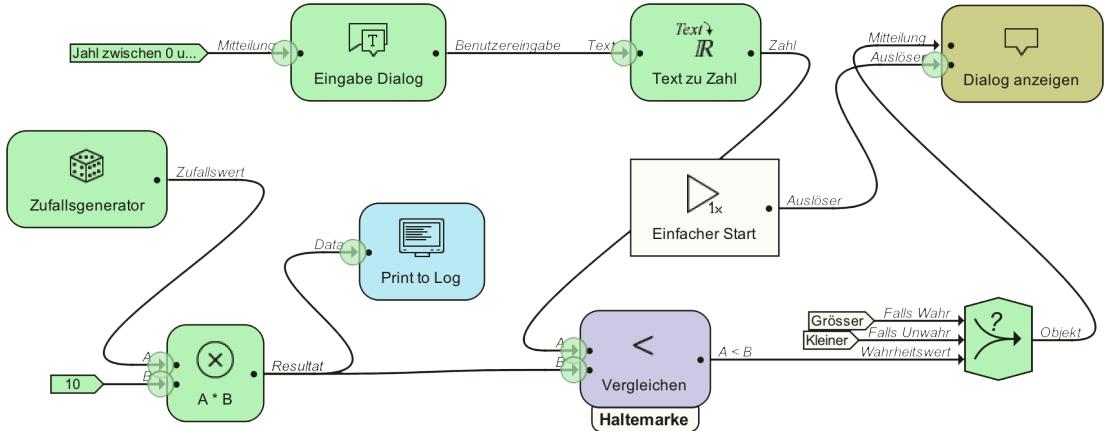


Abb. 2.23. Debugger

### Farbliche Kennzeichnungen

Durch die Hintergrundfarbe (Abb. 2.24) der Elemente (Abb. 2.23) ist es für den Benutzer ersichtlich, welchen Zustand jedes Element momentan besitzt. Hierbei ist darauf zu achten, dass Elemente in verschiedenen Kontexten verschiedene Zustände haben können.

■ Vorbereiten ■ Warten auf Eingänge ■ Arbeitet ■ Liefern an Ausgänge aus ■ Beendet ■ Fehlgeschlagen (Fatal) □ Unbekannter Zustand

Abb. 2.24. Farben Zustände

### Kennzeichnung von belieferten Eingängen

Eingänge, die bereits beliefert sind, sind mit grünen Kreisen (Abb. 2.25) markiert.



Abb. 2.25. Eingegangene Daten

## Inspect-Overlay

Während der Laufzeit besteht die Möglichkeit, einzelne Elemente zu inspizieren. Durch einfaches Klicken auf ein Element öffnet sich das Inspect-Overlay (Abb. 2.26), in dem angezeigt wird, welche Daten an den jeweiligen Eingängen dieses Elementes bereits angekommen sind. Hierbei ist zu beachten, dass Elemente je nach Kontext verschiedene Daten an ihren Eingängen anliegen haben. Vorher ist es also ratsam, sich zu vergewissern, dass man in der Kontext-Übersicht den gewünschten Kontext gewählt hat.

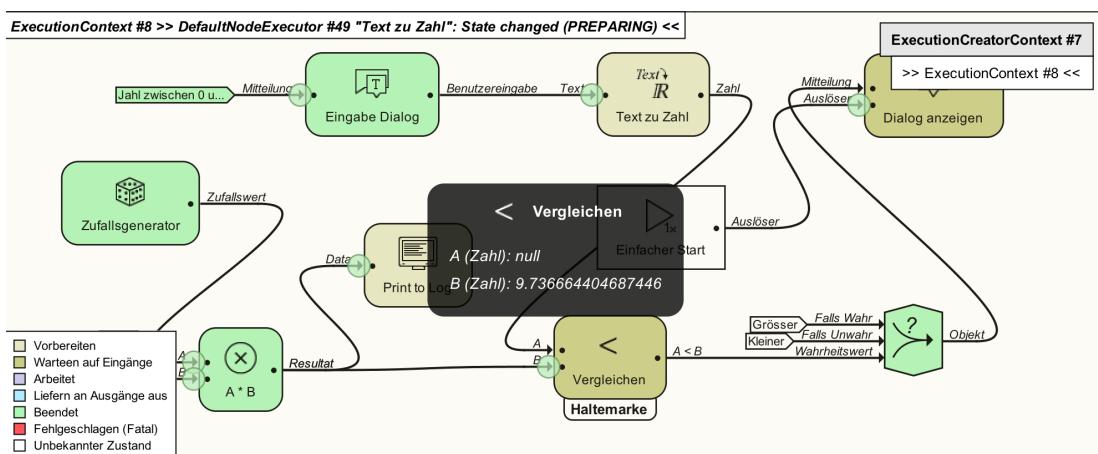


Abb. 2.26. Inspect-Overlay

## Kontext-Übersicht

Die Kontext-Übersicht (Abb. 2.27) ermöglicht es, zu sehen, welche Kontexte aktuell aktiv sind und erlaubt es zwischen diesen zu wechseln. Der Kontext, der aktuell betrachtet wird, wird mit einem hellen Hintergrund hervorgehoben. Durch *Klicken* auf einen anderen Kontext, lässt sich in diesen wechseln. Falls ein Kontext aktuell ein Halten ausgelöst hat, ist dieser mit >> << gekennzeichnet.

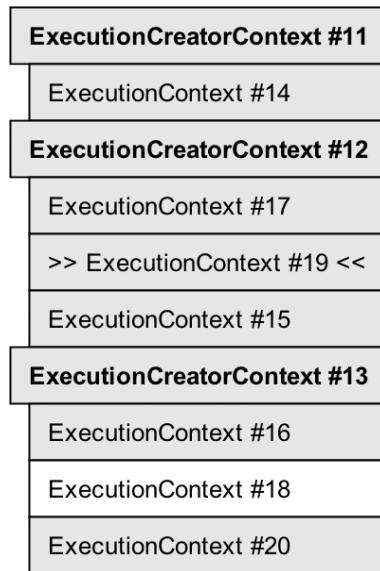


Abb. 2.27. Kontext-Übersicht

## Monitor-Fenster

Ein weiteres Werkzeug, um Projekte zu debuggen, bietet das Monitor-Fenster (Abb. 2.28). Hier werden zusätzliche Informationen zur Laufzeit sowie ein Log der Debugger-Ereignisse angezeigt.

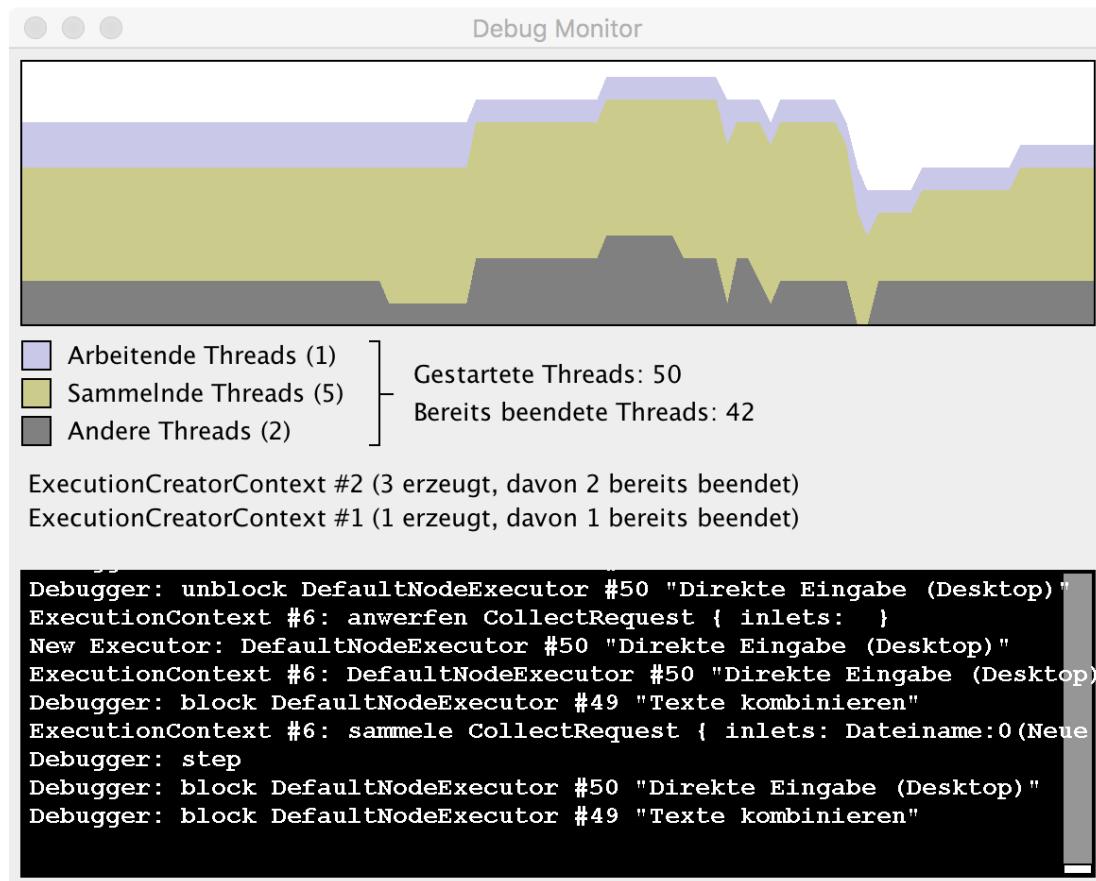


Abb. 2.28. Monitor-Fenster

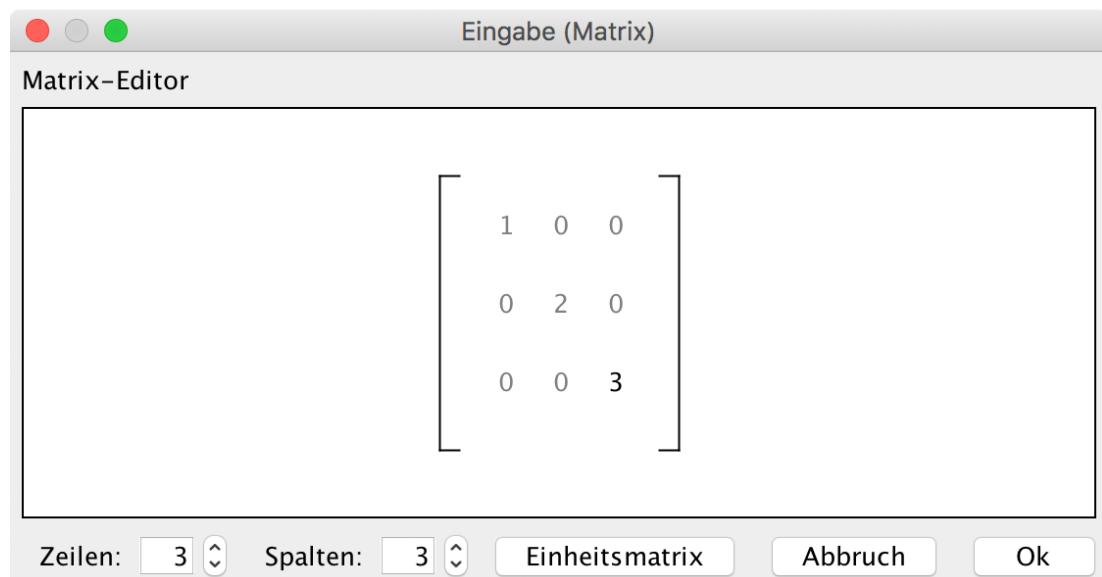
### 2.2.9 Direkte Eingaben

Für viele Datentypen steht die Möglichkeit zur Verfügung, über das Kontextmenü > „Direkter Wert für Eingang setzen“ einen Wert direkt zuzuordnen. Hierfür öffnet sich in diesem Fall ein Fenster, in dem der Wert definiert werden kann (Abb. 2.29 und Abb. 2.30). Um diesen nach dem Setzen wieder zu verändern, reicht ein *einfacher Kick* auf das entsprechende *Direkte Eingabe* Element.

Folgende Datentypen können grafisch über die *Direkte Eingabe* zugewiesen werden:

- “Zahl”
- “Zahl...”
- “Wahrheitswert”
- “Text”
- “8 Bit Ganzzahl”
- “8 Bit Ganzzahl...”

- “16 Bit Ganzzahl”
- “16 Bit Ganzzahl...”
- “32 Bit Ganzzahl”
- “32 Bit Ganzzahl...”
- “64 Bit Ganzzahl”
- “64 Bit Ganzzahl...”
- “32 Bit Gleitkommazahl”
- “32 Bit Gleitkommazahl...”
- “64 Bit Gleitkommazahl”
- “64 Bit Gleitkommazahl...”
- “Farbe”
- “SmartIdentifier”
- “Vektor”
- “Matrix” (Abb. 2.29)
- “Quellcode” (Abb. 2.30)
- “Dateipfad”
- “Dateipfad...”



**Abb. 2.29.** Eingabe (Matrix)

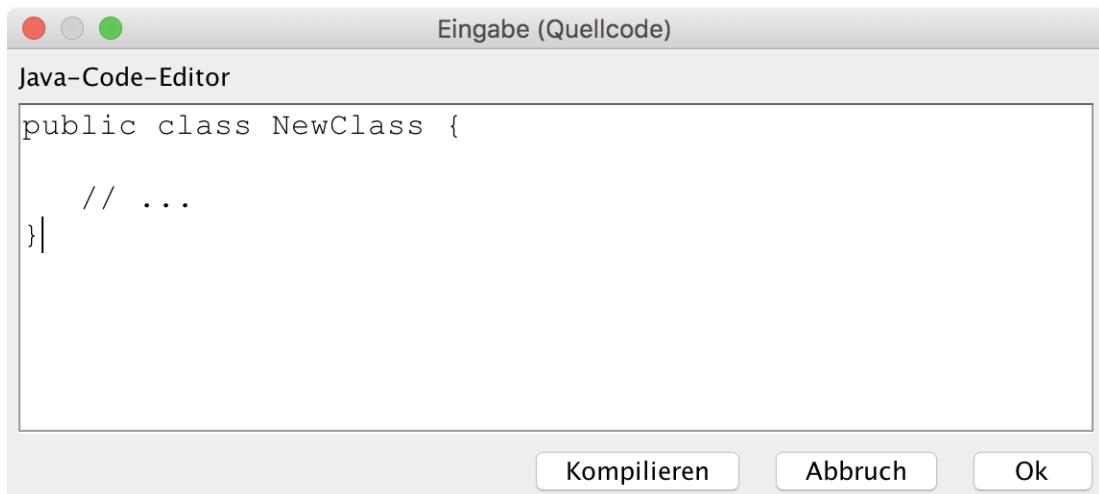


Abb. 2.30. Eingabe (Quellcode)

## 2.3 Verwendung als Programmierer (Erweiterung des bestehenden Frameworks)

Schwerpunkt bei der Entwicklung des Programms lag darauf, die Anwendung möglichst offen zu gestalten und es fremden Programmierern einfach zu machen, eigene Element-Definitionen hinzuzufügen.

Um neue Element-Definitionen zum Programm hinzuzufügen, reicht es aus, entsprechend kompatible (kompilierte) *class*-Dateien in einen der folgenden Verzeichnisse abzulegen:

- Im Programm-Verzeichnis im Paket `reflection.nodedefinitions`. (Hierbei ist auf die korrekte Paketbezeichnung `package reflection.nodedefinitions` zu achten)
- Innerhalb des Projekt-Verzeichnisses im Unterverzeichnis `nodedefinitions`.

### 2.3.1 Umsetzung von Element-Definitionen

Eine Klasse ist eine kompatible Element-Definition, wenn diese von der Klasse `reflection.common.NodeDefinition` ableitet.

Um Kompilier-Fehler zu vermeiden, ist es empfehlenswert das Paket `reflection.common`<sup>9</sup> innerhalb der Entwicklungsumgebung im Klassenpfad liegen zu haben (hierfür reicht es einfach, dieses Paket in das aktuelle Quellverzeichnis zu kopieren. Hierbei darauf achten dass die Ordnerstruktur des Paketes `./reflection/common/` bewahrt bleibt).

<sup>9</sup> <https://github.com/colbach/Bachelor-Projekt/tree/master/Hauptprogramm/src/reflection/common>

Zur Unterstützung ist es darüberhinaus zusätzlich förderlich, die Pakete `reflection.customdatatypes`<sup>10</sup> sowie `reflection.additionalnodedefinitioninterfaces`<sup>11</sup> auch über den Klassenpfad verfügbar zu haben. Das Paket `reflection.customdatatypes` stellt Datentypen zur Verfügung, die zusätzlich zu den bereits gängigen Java-Datentypen über Eingänge entgegengenommen werden sowie über Ausgänge weitergereicht werden können. Das Paket `reflection.additionalnodedefinitioninterfaces` stellt Schnittstellen zur Verfügung, welche es erlauben, Element-Definitionen um zusätzliche Funktionalitäten zu erweitern, die nicht bereits von `reflection.NodeDefinition` abgedeckt werden.

Es gibt fünf Schnittstellen, die in Bezug auf Element-Definitionen am wichtigsten sind. Diese sind: `NodeDefinition`, `ContextCreator`, `InOut`, `ContextCreatorInOut`, sowie `API`, aus dem Paket `reflection.common`. Im Folgenden werde ich im Einzelnen auf die Verwendung dieser Schnittstellen eingehen.

Am offensichtlich wichtigsten ist die Schnittstelle `NodeDefinition`. Diese Schnittstelle muss von jeder Element-Definition implementiert werden. Folgende Methoden werden von `NodeDefinition` verlangt:

- `public int getInletCount()`: Muss die Anzahl der Eingänge zurückgeben. Virtuelle Eingänge werden nicht mitgezählt.
- `public Class getClassForInlet(int inletIndex)`: Muss die Datentypen für die Eingänge 0 bis `getInletCount()-1` zurückgeben.
- `public String getNameForInlet(int inletIndex)`: Muss den Namen für die Eingänge 0 bis `getInletCount()-1` zurückgeben.
- `boolean isInletForArray(int inletIndex)`: Muss zurückgeben, ob die Eingänge 0 bis `getInletCount()-1` eine Liste als Eingabe erwarten.
- `public boolean isInletEngaged(int inletIndex)`: Muss zurückgeben, ob die Eingänge 0 bis `getInletCount()-1` verpflichtend verbunden sein müssen oder ob auf ein Standardwert (bzw. generierten Wert) zurückgegriffen werden kann.
- `public int getOutletCount()`: Muss die Anzahl der Ausgänge zurückgeben. Virtuelle Ausgänge werden nicht mitgezählt.
- `public Class getClassForOutlet(int outletIndex)`: Muss die Datentypen für die Ausgänge 0 bis `getOutletCount()-1` zurückgeben.
- `public String getNameForOutlet(int outletIndex)`: Muss den Namen für die Ausgänge 0 bis `getOutletCount()-1` zurückgeben.

<sup>10</sup> <https://github.com/colbach/Bachelor-Projekt/tree/master/Hauptprogramm/src/reflection/customdatatypes>

<sup>11</sup> <https://github.com/colbach/Bachelor-Projekt/tree/master/Hauptprogramm/src/reflection/additionalnodedefinitioninterfaces>

- `public boolean isOutletForArray(int outletIndex)`: Muss zurückgeben ob die Ausgänge 0 bis `getOutletCount()-1` eine Liste als Ausgabe weitergeben werden.
- `public String getName()`: Muss den Namen des Elementes zurückgeben.
- `public String getDescription()`: Muss die Beschreibung des Elementes zurückgeben. Text, der nach `//tags//` angegeben ist, wird abgeschnitten und dient nur dazu Elemente besser zu finden. Tags, die zwischen [ und ] angegeben werden, werden als Kategorien interpretiert.
- `public String getUniqueName()`: Muss einen eindeutigen Bezeichner dieser Element-Definition zurückgeben.
- `public String getIconName()`: Soll `null` oder den Namen der Datei angeben, die als Icon des Elementes verwendet werden sollen.
- `public int getVersion()`: Muss die Versionsnummer der Element-Definitionen zurückgeben.
- `public void run(InOut io, API api) throws Exception`: Dies ist die Methode, in der die eigentliche Arbeit durchgeführt werden soll.

Wie im Abschnitt Funktionsweise aufgeführt gibt es zusätzlich zu *Einfachen Elementen* auch *Kontext Erzeugende Elemente*. Um solche kennzuzeichnen existiert die zusätzliche Schnittstelle `ContextCreator`. `ContextCreator` leitet von `NodeDefinition` ab und erwartet somit von der implementierenden Klasse, alle Methoden dieser zu implementieren. Die Schnittstelle `ContextCreator` führt jedoch selbst **keine** neuen Methoden ein. Die Schnittstelle dient alleinig dazu *Kontext Erzeugende Element-Definitionen* zu kennzeichnen und nicht diese um eine Funktionalität zu erweitern.

Bei der Entwicklung von neuen Element-Definitionen sind zusätzlich die Schnittstellen `InOut`, `ContextCreatorInOut` sowie `API` relevant. Bevor ich auf diese eingehe, muss angemerkt werden, wieso es sich bei diesen um Schnittstellen und nicht um Klassen handelt. Der Grund hierfür liegt nicht in einer technischen Notwendigkeit, sondern daran, wie das Paket `reflection.common` verwendet werden soll. Sinn dieses Paketes ist, dass dieses (mit den in ihr befindlichen Klassen und Schnittstellen) in einen beliebigen Klassenpfad kopiert werden kann und damit als Schnittstelle zwischen meinem Programm und den von anderen Entwicklern erstellten Klassen fungieren kann. Würden sich Klassen, die in Abhängigkeit mit andern Klassen meines Programms stehen, innerhalb von diesem Paket befinden, so würde dieses einfache Kopieren unweigerlich zu Kompilierfehlern führen, die nur sehr schwer vermieden werden könnten. Um dieses Problem zu umgehen, habe ich von Klassen, die extern benötigt werden, obwohl diese interne Abhängigkeiten besitzen, jeweils nur eine Schnittstelle in das Paket `reflection.common` gelegt. Auf diese Weise stelle ich sicher, dass Objekte dieser Klassen extern verwendet werden können, ohne dass ihre genaue Implementierung extern bekannt sein muss.

`InOut`-Objekte dienen dazu, Daten an Eingängen abzurufen und Daten an Ausgänge weiterzugeben. Hierfür stehen unter anderem folgende Methoden zur Verfügung:

- `public Object[] in(int i, Object[] def) throws TerminatedException;`  
n: Ruft Daten an Eingang *i* ab. Falls *i* nicht verbunden, wird *def* zurückgegeben.
- `public Object in0(int i, Object def) throws TerminatedException;`  
Ruft Liste von Daten an Eingang *i* ab. Falls *i* nicht verbunden wird, *def* zurückgegeben.
- `public void out(int i, Object ausgabe) throws TerminatedException;`  
Gibt Daten an Ausgang *i* weiter.
- `public void out(int i, Object[] ausgabe) throws TerminatedException;`  
Gibt Liste von Daten an Ausgang *i* weiter.
- `public boolean outConnected(int i) throws TerminatedException;`  
Gibt an, ob Ausgang *i* verbunden ist.
- `public long getContextIdentifier() throws TerminatedException;`  
Gibt die Identifikation des aktuellen Kontextes zurück.

`ContextCreatorInOut` ist eine Erweiterung von `InOut`. Wenn es sich bei einer Element-Definition um eine *Kontexte Erzeugende Element-Definition* handelt (diese also von `ContextCreator` ableitet), erhält die `run`-Methode als `io` Parameter ein Objekt, das `ContextCreatorInOut` implementiert. Dieses kann dementsprechend von `InOut` nach `ContextCreatorInOut` gecastet werden und erweitert dieses um folgende Methoden:

- `public void startNewContext() throws Exception, TerminatedException;`  
Startet einen neuen Kontext mit den Ausgangs-Daten, die zu diesem Zeitpunkt bereits weitergegeben wurden.
- `public int getRunningContextCount() throws TerminatedException;`  
Gibt zurück wieviele Kind-Kontexte bereits laufen. (Bevor `startNewContext()` das erste Mal aufgerufen wurde dementsprechend 0)

API-Objekte dienen dazu, eine Reihe von Funktionalitäten der `run`-Methode zur Verfügung zu stellen. Die angebotenen Funktionalitäten würden sich größtenteils auch selbst implementieren lassen, jedoch sollen diese die Entwicklung von Element-Definitionen unterstützen und wiederholenden Programmcode verhindern.

## Beispiel: Primzahl-Test

```

import reflection.common.*;

public class PrimTestNodeDefinition implements NodeDefinition {

    public int getInletCount() { return 1; }

    public Class getClassForInlet(int index) { return Long.class; }

    public String getNameForInlet(int index) { return "n"; }

    public boolean isInletEngaged(int i) { return true; }

    public boolean isInletForArray(int index) { return false; }

    public int getOutletCount() { return 1; }

    public Class getClassForOutlet(int index) { return Boolean.class; }

    public String getNameForOutlet(int index) { return "Ist Primzahl"; }

    public boolean isOutletForArray(int index) { return false; }

    public String getName() { return "Primzahl-Test"; }

    public String getDescription() {
        return "Testet ob es sich bei einer Zahl n um eine Primzahl handelt."
        + Dies kann bei grossen Zahlen eine längere Zeit in Anspruch nehmen."
        + TAG_PREAMBLE + " [Math] Prim Primzahl Test Probieren ";
    }

    public String getUniqueName() { return "buildin.PrimTest"; }

    public String getIconName() { return "Is-Prim_30px.png"; }

    public int getVersion() { return 1; }

    public void run(InOut io, API api) {
        Long n = io.inN(0, 1).longValue();
        io.out(0, isPrim(n, io, api));
    }

    public static boolean isPrim(long n, InOut io, API api) { // Hilfsmethode
        if (n == 1)
            return false;
        if (n < 0) {
            api.additionalPrintErr("Primzahlen sind immer positiv");
            return false;
        }
        if (n <= 2)
    }
}

```

```
        return true;
    for (long i = 2; i <= n / 2; i++) {
        if (n % i == 0)
            return false;
        if (i % 1000 == 0)
            io.terminatedTest();
    }
    return true;
}

}
```

## TerminatedException

Viele Methoden der Schnittstellen `InOut`, `ContextCreatorInOut` und `API` sind mit `throws TerminatedException` gekennzeichnet. Der Grund hierfür ist, dass auf diese Art laufende `run`-Methoden abgebrochen werden können, wenn die Ausführung vorzeitig beendet wurde. Um zu verhindern, dass in Situationen, in denen keine Framework-Methoden verwendet werden, die Ausführung nicht abgebrochen werden kann, soll aus diesem Grund wie im Beispiel gezeigt von Zeit zu Zeit manuell die Methode `io.terminatedTest()` aufgerufen werden.

## Datentypen

Um Element-Definitionen konsistent zu halten, ist es zu empfehlen in der Regel folgende Klassen (bzw. Schnittstellen) als Datentypen zu verwenden:

- `java.lang.Boolean`
- `java.lang.String`
- `java.lang.Character`
- `java.lang.Number` (Schnittstelle; Implementierung: `Byte`, `Short`, `Integer`, `Long`, `Float` und `Double`)
- `java.lang.Byte`
- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Float`
- `java.lang.Double`
- `java.util.Date`
- `java.awt.Color`
- `java.lang.Object`
- `java.awt.Image` (Abstrakte Klasse; Ableitung: `java.awt.BufferedImage`)

- `java.nio.File`
- `reflection.customdatatypes.BooleanGrid`
- `reflection.customdatatypes.math.MathObject` (Abstrakte Klasse; Ableitungen: `NumberMathObject`, `Matrix` und `Vector`)
- `reflection.customdatatypes.math.Matrix` (Abstrakte Klasse; Ableitungen: `OneDimensionalArrayBasedMatrix`, `PrimitiveDoubleWrappingMatrix`, `PrimitiveFloatWrappingMatrix` und `TwoDimensionalArrayBasedMatrix`)
- `reflection.customdatatypes.math.Vector` (Schnittstelle; Implementierung: `ArrayBasedVector`)
- `reflection.customdatatypes.SourceCode`
- `reflection.customdatatypes.Function` (Abstrakte Klasse, muss selbst implementiert werden)
- `reflection.customdatatypes.rawdata.RawData` (bzw. deren Erweiterungen `RawDataFromFile` und `RawDataFromNetwork`)
- `reflection.customdatatypes.camera.Camera` (Schnittstelle; Implementierung: `Webcam`)
- `reflection.customdatatypes.SmartIdentifier`

# 3

---

## Technische Umsetzung

### 3.1 Implementierung

#### 3.1.1 Umfang

**CLOC<sup>1</sup>** Ausgabe:

```
$ cloc ~/GitHub/Bachelor-Projekt/Hauptprogramm/src

 527 text files.
 527 unique files.
 14 files ignored.

github.com/AlDanial/cloc v 1.72  T=3.37 s (152.0 files/s, 16483.5 lines/s)
-----
Language           files      blank     comment      code
-----
Java              513        8624       2108      44897
-----
SUM:              513        8624       2108      44897
-----
```

Das reine Hauptprogramm umfasst 513 Java-Dateien organisiert in 113 Paketen. Summiert sind in diesen 44897 Zeilen Java-Code enthalten (leere Zeilen und Kommentare ausgeschlossen). Es sind keine fremden Dateien mit Ausnahme der Datei *Gauss.java*, die ich (mit Erlaubnis) aus den Unterlagen der Vorlesung *Computergrafik* von *Prof. Dr. Ing. Fritz Nikolai Rudolph* entnommen habe.

Da der Umfang des Programmcodes eine gewisse Größe erreicht hat, ist es mir leider nicht möglich, alle Teile meiner Implementierung zu erklären. Stattdessen werde ich versuchen, grob von oben herab alle Module kurz zu beschreiben und nur bestimmte Implementierungsdetails genauer zu erläutern.

---

<sup>1</sup> <http://cloc.sourceforge.net/>

### 3.1.2 Aufbau

Das Programm ist modular aufgebaut und besteht grundlegend aus folgenden Modulen (intern):

- Hauptmodul (Paket: `main.*`; Wichtigste Klassen: `MainClass`, `ComponentHub`)
- Hilfs-Klassen (Paket: `utils.*`)
- Model (Paket: `model.*`; Wichtigste Klassen: `Project`, `Node`)
- Projektausführung (Paket: `projectrunner.*`; Wichtigste Klassen: `ProjectRunner`, `ProjectExecution`)
- Kommandozeile (Paket: `commandline.*`; Wichtigste Klasse: `CommandLine`)
- Grafische Benutzeroberfläche (Paket: `view.*`; Wichtigste Klasse: `MainWindow`)
- Persistente Einstellungen (Paket: `settings.*`; Wichtigste Klasse: `Settings`, `GeneralSettings`)
- Protokollierung (Paket: `logging.*`; Wichtigste Klassen: `AdvancedLogger`, `AdditionalLogger`)
- Über Reflexion geladene Klassen und geteilte Klassen/Schnittstellen (Pakete: `reflection.*`, `model.resourceloading.*`; Wichtigste Klasse: `NodeDefinition`)

Der Aufbau in Abb. 3.1 soll einen Überblick darüber verschaffen, wie die Klassen der einzelnen Module zusammenarbeiten.

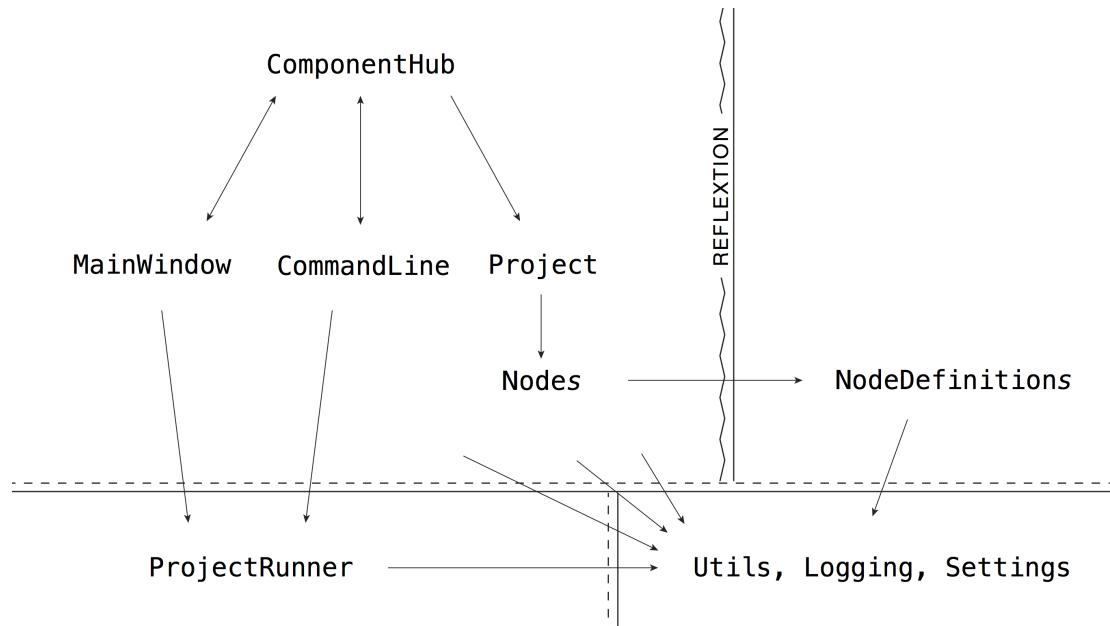


Abb. 3.1. Aufbau Module

Im Folgenden werde ich auf die einzelnen Module eingehen und beschreiben, welche Tätigkeit diese erfüllen und wie diese funktionieren.

### 3.1.3 Hauptmodul

#### MainClass

Das Hauptmodul hat die Aufgabe, die Anwendung zu starten und alle nötigen Komponenten zu laden. Je nachdem, ob Konsole, grafisches Benutzerinterface oder beides gestartet werden soll, werden die verschiedenen Module geladen.

#### ComponentHub

Die Klasse `ComponentHub` stellt das Drehkreuz, über das das grafische Benutzerinterface, die Kommandozeile und das Model kommunizieren. Zugriff auf den `ComponentHub` erfolgt über ein Singleton und ist theoretisch aus dem gesamten Programm möglich. Der `ComponentHub` kann jeweils eine Instanz auf ein `Project`-Objekt, ein `CommandLinePrompt`-Objekt und ein `MainWindow`-Objekt halten. Ist die Kommandozeile oder das grafische Benutzerinterface nicht aktiv, so bleibt dieses Attribut `null`.

### 3.1.4 Hilfs-Klassen

Eine ganze Reihe nützliche Hilfsklassen sind in dem Paket `utils.*` zusammengefasst. Die hier befindlichen Hilfs-Methoden und Hilfs-Strukturen werden durchgängig durch das Programm an den verschiedensten Stellen verwendet und sollen den restlichen Code übersichtlicher und lesbarer machen sowie Redundanz und die damit verbundene Fehleranfälligkeit vermeiden.

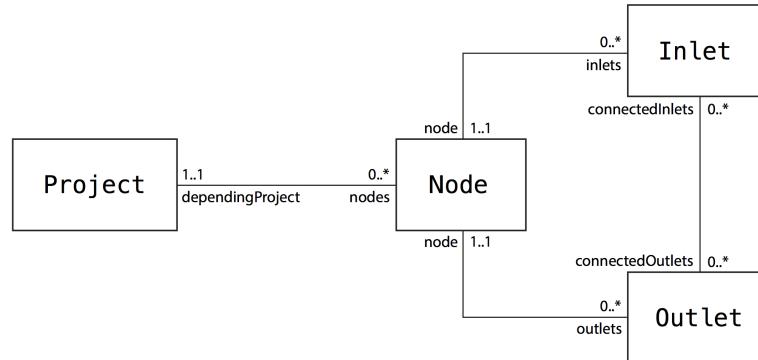
### 3.1.5 Model

Das Model ist grundlegend, wie in Abb. 3.2 dargestellt, aufgebaut.

Klasse	Enthält (gekürzt)	Beschreibung
<code>Project</code>	Elemente (Liste von <code>Node</code> -Objekten), Verweise auf projektspezifische Verzeichnisse und Dateien, Projekteinstellungen ( <code>ProjectSettings</code> )	Repräsentiert ein bestimmtes, vom Benutzer erstelltes Projekt.

Klasse	Enthält (gekürzt)	Beschreibung
<b>Node</b>	Eingänge (Liste von <b>Inlet</b> -Objekten), Ausgänge (Liste von <b>Outlet</b> -Objekten), Position (X- und Y-Koordinate), Identifikation ( <code>long</code> ), Zugehörige Element-Definition ( <code>NodeDefinition</code> )	Repräsentiert ein in einem Projekt platziertes Element.
<b>Inlet</b>	Verbundene Ausgänge (Liste von <b>Outlet</b> -Objekten), Identifikation ( <code>long</code> ), Zugehöriges Element ( <code>Node</code> ), Index ( <code>int</code> )	Repräsentiert ein zu einem Element zugehörigen Eingang.
<b>Outlet</b>	Verbundene Eingänge (Liste von <b>Inlet</b> -Objekten), Identifikation ( <code>long</code> ), Zugehöriges Element ( <code>Node</code> ), Index ( <code>int</code> )	Repräsentiert ein zu einem Element zugehörigen Eingang.

Zu beachten ist, dass die Klasse `Node` selbst keine Informationen über Namen, Icons oder Typen der Ein- und Ausgänge enthält. Informationen über diese sind nicht im Model, sondern in den über Reflexion geladenen Element-Definitionen (`NodeDefinition`) enthalten.



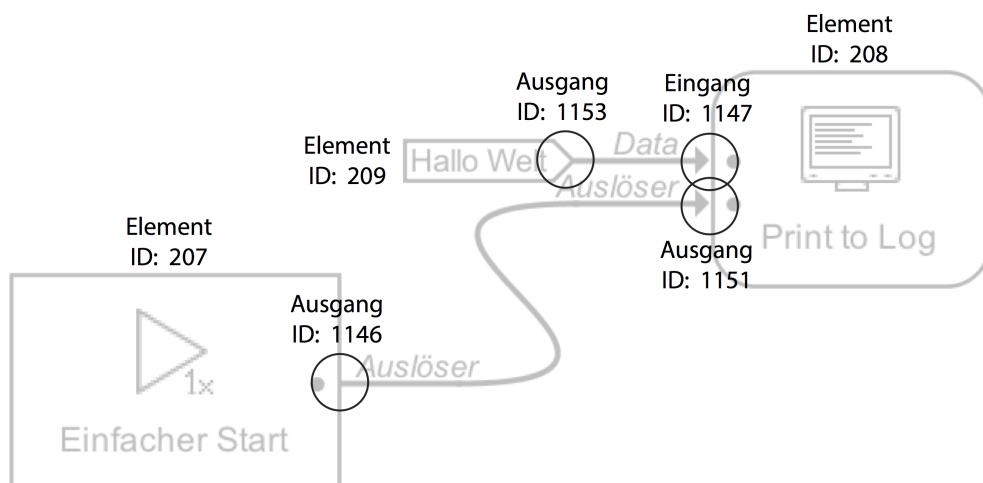
**Abb. 3.2.** Aufbau Model

## Serialisierung und Deserialisierung von Projekten

Um Projekte persistent zu speichern, müssen diese serialisiert werden. Dies könnte theoretisch mittels der Java-eigenen Standardimplementierung für die Serialisierung von Objekten realisiert werden, jedoch wäre dies aus zwei Gründen problematisch. Zum einen könnte dies zu Inkompatibilität zwischen verschiedenen Programmversionen führen, zum anderen würde dies es erschweren Projekt-Dateien händisch zu lesen oder zu bearbeiten. Um die bestmögliche Kontrolle über die Projektdateien zu haben, habe ich mich dazu entschieden die Serialisierung und Deserialisierung der Projektdateien selbst zu übernehmen.

Um eine transparente und stabile Projekt-Serialisierung zu garantieren, habe ich mich dazu entschieden, Projekte strukturiert in das JSON-Format umzuwandeln und dann als UTF-8 String zu serialisieren. Die JSON-Struktur besteht auf der obersten Ebene aus drei Arrays. Das Array "nodes" enthält alle Elemente des Projektes, das Array "inlets" alle Eingänge und das Array "outlets" alle Ausgänge. Hierbei ist es nicht entscheidend, ob die Ein-/Ausgänge verbunden oder nicht verbunden sind. Da Ein-/Ausgänge keine Angaben über ihren Index enthalten, ist es zwingend nötig, dass alle Ein-/Ausgänge (auch die nicht verbundenen) in der Struktur enthalten sind, da sich dieser sonst nicht wiederherstellen lassen würde. Innerhalb der JSON-Struktur werden alle Objekte mittels ihrem ID-Feld identifiziert. Diese Identifikationen entsprechen der des Models und bleiben über den gesamten Lebenszyklus des Projektes konsistent. Über ein Feld namens `definitionUniqueNameVersion` wird jedem Element eine Element-Definition zugeordnet, die beim Deserialisieren über reflexion geladen werden muss. Das Feld `settabledata` enthält vom Benutzer gesetzte *direkte Eingaben*.

Wie die JSON-Struktur aufgebaut ist, soll anhand des folgenden Beispiels demonstriert werden. Hier dargestellt, ist ein einfacher "Hello World"-Aufbau aus dem ersten Beispiel des Abschnittes 2.2.2:



Nach der Umwandlung als JSON-Struktur sieht der Aufbau wie folgt aus:

```
{
  "nodes": [
    {
      "id": 209,
      "inlets": [],
      "outlets": [ 1153 ],
      "definitionUniqueNameVersion": "special.directinput(java.lang.String)[0]",
      "uiCenterX": -2147483648,
      "uiCenterY": -2147483648,
      "settabledata": [ "r00ABXQACkhbGxvIFdlbHQ=" ]
    },
    {
      "id": 207,
      "inlets": [],
      "outlets": [ 1146 ],
      "definitionUniqueNameVersion": "buildin.Start1x[1]",
      "uiCenterX": 156,
      "uiCenterY": 217,
      "settabledata": null
    },
    {
      "id": 208,
      "inlets": [ 1147, 1148, 1149, 1150, 1151 ],
      "outlets": [ 1152 ],
      "definitionUniqueNameVersion": "buildin.PrintLog[1]",
      "uiCenterX": 393,
      "uiCenterY": 146,
      "settabledata": null
    }
  ],
  "inlets": [
    {
      "id": 1147, "connectedOutlets": [ 1153 ] },
    {
      "id": 1148, "connectedOutlets": [ ] },
    {
      "id": 1149, "connectedOutlets": [ ] },
    {
      "id": 1150, "connectedOutlets": [ ] },
    {
      "id": 1151, "connectedOutlets": [ 1146 ] }
  ],
  "outlets": [
    {
      "id": 1153, "connectedInlets": [ 1147 ] },
    {
      "id": 1146, "connectedInlets": [ 1151 ] },
    {
      "id": 1152, "connectedInlets": [ ] }
  ]
}
```

Auffallend bei diesem Beispiel ist, dass anstelle von `Hello Welt` im Feld `settabledata` des Elementes mit der ID 208 `r00ABXQACkhbGxvIFdlbHQ=` zu lesen ist. Dies liegt daran, dass *direkte Eingaben* über die Java-Standart-Serialisierung in ein `byte`-Array umgewandelt und dann als Base64-String in die JSON-Struktur eingepflegt werden.

## Schreiben und Lesen von Projekten

Projekt-Dateien werden in einem eigenen Verzeichnis in einem vom Benutzer definierten Pfad angelegt und wieder geladen. Hierbei sieht die Verzeichnis-Struktur wie folgt aus:

*Projekt-Name/*

```
→ nodedefinitions/
  → info.txt
  → ... (Optional: projekteigene Element-Definitionen)
→ properties.txt
→ structure.json (Projekt-Struktur als JSON-Datei)
→ versions/
  →... (Vorhergehende Projektversionen)
```

Bei jedem Speichervorgang wird die jeweils vorhergehende Projekt-Version in das Verzeichnis *Projekt-Name/versions/* verschoben. Innerhalb dieses Verzeichnisses werden alte Projekte-Verzeichnisse wie folgt benannt: *Versionsnummer (dd-MM-yyyy hh-mm-ss)*.

Die Datei *properties.txt* dient dazu, projektspezifische Parameter und Einstellungen zu speichern. Wie diese Datei genau aufgebaut ist, wird im Abschnitt *Persistente Einstellungen* genauer erläutert.

## Projektausführung

### Grundaufbau

Die Klassen, die für die Ausführung von Projekten zuständig sind, sind grundlegend, wie in Abb. 3.3 dargestellt, miteinander verbunden. Hierbei ist anzumerken, dass es sich nur um eine vereinfachte unvollständige Darstellung handelt, die nur die wichtigsten Klasse beinhaltet. Insgesamt ist für die Ausführung ein Konstrukt von 60 Klassen und Schnittstellen zuständig. Insbesondere sind folgende Elemente in der Darstellung nicht berücksichtigt: Herumgereichte Callback-Referenzen, Kontroll- und Steuerelemente sowie Debugger. In der folgenden Liste gehe ich auf die wichtigsten Klassen, die für die Projektausführung zuständig sind, ein und erkläre deren Funktion:

- **ProjektRunner:** Diese Klasse ist in der Hierarchie die oberste Klasse. Über das *Singleton-Pattern* wird sichergestellt, dass es nur genau eine Instanz dieser Klasse geben kann. Über die Methode `executeProject(...)` kann die Ausführung von Projekten gestartet werden. Als Parameter erwartet die Methode unter anderem eine Reihe von Callback-Objekten, die während der Ausführung bei bestimmten Ereignissen oder bei Beendigung aufgerufen werden. Außerdem gibt die Methode

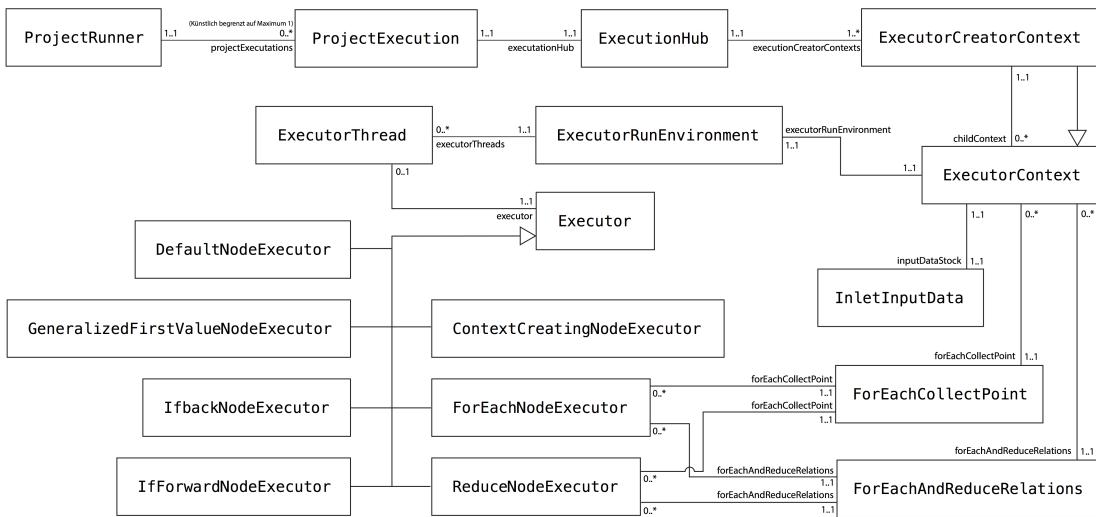


Abb. 3.3. Grundaufbau Projektausführung

ein Objekt der Klasse `ProjectExecutionRemote` zurück. Falls bereits zu viele Projektausführungen laufen, wird eine `ToManyConcurrentProjectExecutionsException` geworfen (In der aktuellen Version ist die maximale Anzahl 1).

- `ProjectExecutionRemote` (nicht in Darstellung): Dient zur Steuerung einer Projektausführung. Objekte dieser Klasse zusammen mit Objekten der Klasse `DebuggerRemote` (nur bei aktivem Debugger) sind die einzigen Objekte der Projektausführung, auf die während der Ausführung von außen zugegriffen werden kann.
- `ProjectExecution`: Diese Klasse repräsentiert eine Projektausführung. Im Konstruktur wird eine eindeutige Identifikation dieser Ausführung vergeben und der Debugger sowie eine Reihe von Kontroll- und Hilfsobjekten instanziert. Außerdem wird eine Instanz der Klasse `ExecutionHub` angelegt.
- `ExecutionHub`: Der `ExecutionHub` startet für jedes *Kontext Erzeugende Element* eine Instanz der Klasse `ExecutorCreatorContext` und sammelt diese Instanzen.
- `ExecutorContext`: Diese Klasse repräsentiert einen Kontext der Ausführung. Dieser besteht grundlegend aus einer Instanz der Klasse `ForEachCollectPoint` zur Sammlung der Daten aus *Für Alle*-Strukturen, einer Instanz der Klasse `ForEachAndReduceRelations` zur Ermittlung, die *Für Alle*-Elemente mit welchen *Sammeln*-Elementen verbunden sind sowie einer Instanz der Klasse `ExecutorRunEnvironment`. Ihre wichtigsten Methoden sind `submitExecutorIfNotSubmitted`, um Elemente anzustoßen, die noch nicht in `executorRunEnvironment` enthalten sind, `collect` um Daten von Eingängen zu sammeln (blockierend) sowie `deliver` um Daten von Ausgängen auszuliefern (und gegebenenfalls diese Elemente anzustoßen).

- **ExecutorCreatorContext:** Diese Klasse erweitert die Klasse `ExecutorContext` um die Funktionalität, neue Kontexte zu erzeugen (also weitere Kind-Instanzen der Klasse `ExecutorContext`). Diese Funktionalität wird von der Klasse `ContextCreatorInOutImplementation` umgesetzt, die während der Laufzeit an *Kontext Erzeugende Elemente* übergeben wird.
- **executorRunEnvironment:** Dient zur Verwaltung der `ExecutorThread`-Objekte. Die wichtigste Methode ist `submit`, die `Executor`-Objekte annimmt und zu diesen einen `ExecutorThread` startet.
- **ExecutorThread:** Diese Klasse erbt von der Klasse `Thread` und verpackt jeweils ein Objekt der Klasse `Executor`, sodass diese ausgeführt werden kann, ohne dass der aufrufende Thread blockiert wird.
- **Executor:** Diese Klasse ist abstrakt und wird von den Klassen `IfBackNodeExecutor`, `IfForwardNodeExecutor`, `ForEachNodeExecutor`, `ReduceNodeExecutor`, `GeneralizedFirstValueNodeExecutor`, `ContextCreatingNodeExecutor` und `DefaultNodeExecutor` erweitert. Diese haben jeweils die Aufgabe, ein Element auszuführen. Hierfür wird an dem entsprechenden Element (`Node`) die Element-Definition (`NodeDefinition`) abgefragt und von dieser die Methode `run` aufgerufen. Als Parameter wird jeweils ein Objekt der Klasse `InOutImplementation` und `APIImplementation` mitgegeben.
- **InOutImplementation** (nicht in Darstellung): Dient dazu, Daten für die Eingänge eines Elementes an Instanzen der Klasse `NodeDefinition` zu übergeben und Daten von Ausgängen von diesen zu sammeln.
- **APIImplementation** (nicht in Darstellung): Stellt Framework-Funktionen Elementen Instanzen der Klasse `NodeDefinition` zur Verfügung.
- **ContextCreatorInOutImplementation** (nicht in Darstellung): Erweitert die Klasse `InOutImplementation` um die Methode `startNewContext()` um einen neuen Kontext zu erzeugen.

## Debugger

Wenn beim Start der Projektausführung der zusätzliche Parameter `debug` auf `true` gesetzt ist, wird eine Instanz der Klasse `Debugger` erzeugt und diese durch die gesamte Projektausführung weiter gereicht. Ist der Debugger nicht aktiv wird eine `null`-Referenz verwendet. Der Debugger stellt einen Logger zur Verfügung, über den sämtliche Ereignisse protokolliert werden. Außerdem besitzt der Debugger Methoden, die bei jedem Zustandswechsel von Elementen aufgerufen werden. Diese Methoden haben im Normalfall, keinen Effekt können jedoch in bestimmten Situationen blockieren. Eine dieser Situationen kann beispielsweise eintreten, wenn es sich bei dem Element, das gerade den Zustand wechselt, um ein Element handelt, das als Breakpoint geführt wird (Diese werden von der Klasse `Breakpoints` verwaltet). Ist dieser Fall eingetreten, werden sämtliche Elemente blockiert bis von außen

die Methode `releaseBlock()` aufgerufen wurde (bzw. eine andere Methode welche diese intern aufruft). Da die Klasse `Debugger` eine der Klassen ist, die nur intern verwendet werden, gibt es eine Klasse `DebuggerRemote`, die bestimmte Methoden der Klasse `Debugger` nach außen verfügbar macht.

### 3.1.6 Kommandozeile

Das Modul für die Kommandozeile ist darauf ausgelegt, einfach erweiterbar zu sein und hat mir während der Entwicklung geholfen, Funktionen des Programms zu testen, bevor die nötigen UI-Elemente dafür fertig waren. Insbesondere besteht die Kommandozeile aus vier wichtigen Elementen, auf die ich im Folgenden eingehe:

- `CommandLineThread`: Diese Klasse leitet von `Thread` und dient dazu, dass die Kommandozeile, ohne zu blockieren, ausgeführt werden kann.
- `CommandLinePrompt`: Kapselt das `CommandLine`-Objekt, interpretiert Benutzereingaben und gibt diese an das `CommandLine`-Objekt weiter.
- `CommandLine`: Die eigentliche Kommandozeile. Hier werden die Funktionen (Instanzen der Klasse `CommandLineFunction`) gesammelt und können ausgeführt werden.
- `CommandLineFunction` und dessen Implementierungen: Die Schnittstelle `CommandLineFunction` definiert, welche Methoden eine Kommandozeilen-Funktion besitzen müssen, damit diese von der Kommandozeile aufgerufen werden können. Folgende Kommandozeilen-Funktionen sind in der aktuellen Version vorhanden: `AliasesCommandLineFunction`, `AnalyseProjectCommandLineFunction`, `CancelCommandLineFunction`, `CancelPromptCommandLineFunction`, `CountOnRunWindowsCommandLineFunction`, `DisposeOnRunWindowsCommandLineFunction`, `ExitCommandLineFunction`, `FileExistsCommandLineFunction`, `GetCommandLineFunction`, `HelpCommandLineFunction`, `PrintArgumentsCommandLineFunction`, `PWDCommandLineFunction`, `RedrawGUIClientCommandLineFunction`, `RepairProjectCommandLineFunction`, `ResetGeneralSettingsCommandLineFunction`, `RunCommandLineFunction`, `SetCommandLineFunction`, `SetGetCommandLineFunction`, `SystemInformationCommandLineFunction`, `TestCommandLineFunction`

### 3.1.7 Grafische Benutzeroberfläche

#### Fenster und Dialoge

Die Übersicht in der Abb. 3.4 stellt dar, welche Fenster und Dialoge sich über das Hauptfenster erreichen lassen. Nicht in dieser Übersicht enthalten sind Fenster, die:  
(1) während der Projektausführung programmiertechnisch geöffnet werden können,  
(2) Dialoge für *Direkte Eingänge* oder (3) Dialoge bei Fehlern oder Warnungen.

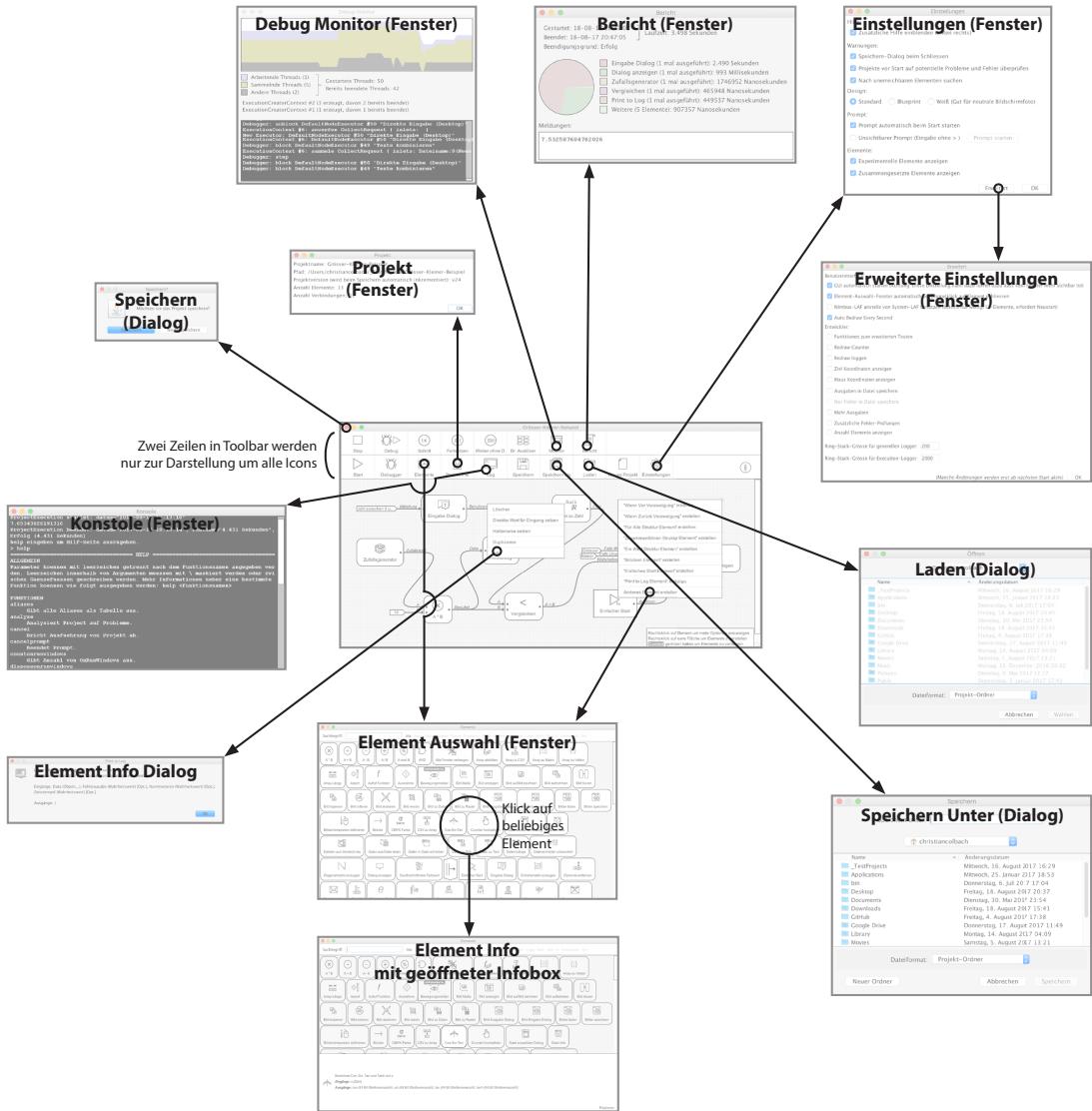


Abb. 3.4. Storyboard

### Fenster

Folgende Fenster wurden für das Programm implementiert und sind im Programmcode enthalten:

- Hauptfenster (**MainWindow**)
- Konsole / Log (**ConsoleWindow**): Zur Darstellung der Konsole oder Log.
- Debug Monitor (**DebugMonitorWindow**): Zur Darstellung von technischen Informationen zur aktuellen Projektausführung (hier sollen später auch noch andere Informationen wie Speicherauslastung etc. zu finden sein).

- Bericht (**RunReportWindow**): Zur Darstellung des Berichtes bei Beendigung einer Projektausführung.
- Einstellungen (**SettingsWindow**): Zur Darstellung und Bearbeitung von Einstellungen.
- Erweiterte Einstellungen (**AdvancedSettingsWindow**): Zur Darstellung und Bearbeitung von erweiterten Einstellungen (größtenteils nicht für den normalen Nutzer relevant).
- Element Auswahl (**NodeCollectionWindow**): Zur Auswahl und dem Platzieren von neuen Elementen.
- Projekt (**ProjectWindow**): Zur Darstellung von projektspezifischen Informationen (später sollen hier auch projektspezifische Einstellungen zu finden sein).
- Text zeigen (**ShowTextWindow**): Zur Darstellung von Text (während der Projektausführung).
- Bild zeigen (**ShowBitmapWindow**): Zur Darstellung von Bildern (während der Projektausführung).
- Mathematisches Objekt zeigen (**ShowMathObjectWindow**) Zur Darstellung von mathematischen Objekten (während der Projektausführung).
- Histogram zeigen (**ShowNumberArrayWindow**): Zur Darstellung von Werte-Listen (während der Projektausführung).

Die Fenster setzen teils mehr, teils weniger auf Standardkomponenten. Viele grafische Komponenten stammen nicht aus dem Standard oder einem fremden Framework, sondern wurden eigens für das Programm implementiert.

### *Dialoge*

Auch wenn eine Reihe von Dialogen bereits durch das Java-Swing-Framework (`javax.swing.*`) vorgegeben werden und damit nicht selbst implementiert werden müssen, decken diese jeweils nur sehr spezielle Fälle ab (Eingabe von Text, Datei-Öffnen-Dialog, Farbauswahl-Dialog, etc.). Aus diesem Grund war es nötig, für bestimmte Aufgaben eigene Dialoge zu entwickeln. Folgende Dialoge wurden eigenständig entwickelt:

- Java-Code-Eingabe-Dialog (**CodeInputDialog**): Aufgabe dieses Dialoges ist es, dem Benutzer die Möglichkeit zu geben, neuen **Java-Code** einzugeben bzw. bestehenden Java-Code zu editieren, zu kompilieren und zu testen.
- Vektor-Eingabe-Dialog (**VectorInputDialog**): Aufgabe dieses Dialoges ist es, dem Benutzer die Möglichkeit zu geben, neue **Vektoren** zu erstellen bzw. bestehende Vektoren zu editieren.
- Vektor-Eingabe-Dialog (**VectorInputDialog**): Aufgabe dieses Dialoges ist es, dem Benutzer die Möglichkeit zu geben, neue **Matrizen** zu erstellen bzw. bestehende Matrizen zu editieren.

## Hauptfenster

Das Hauptfenster setzt sich komplett aus eigenen Komponenten zusammen und enthält keine Standardkomponenten.

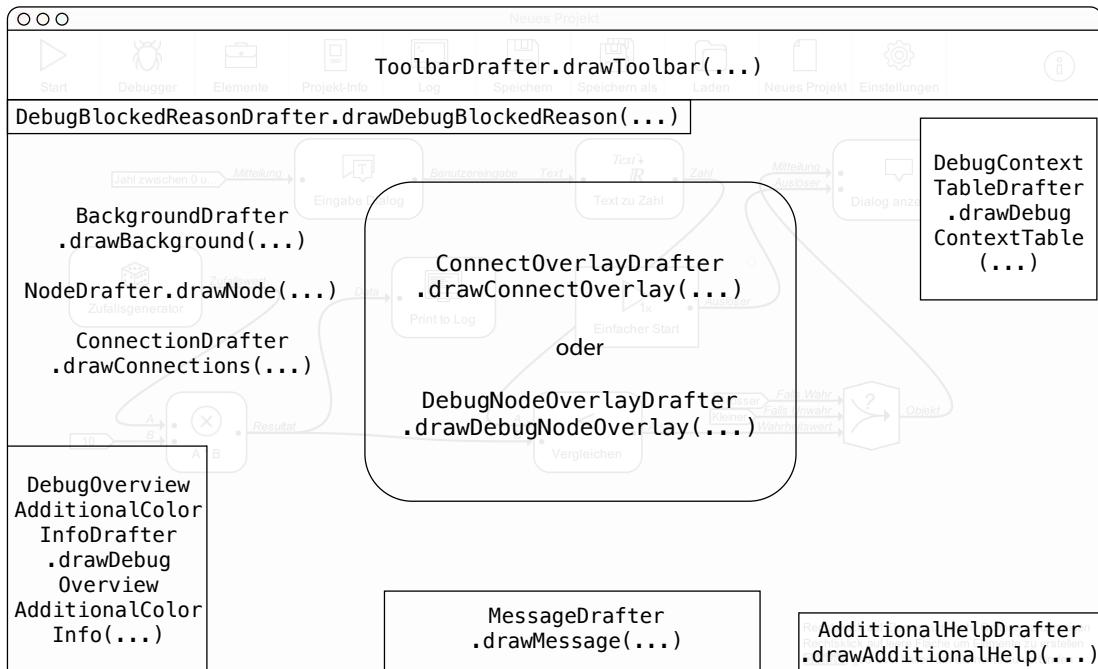


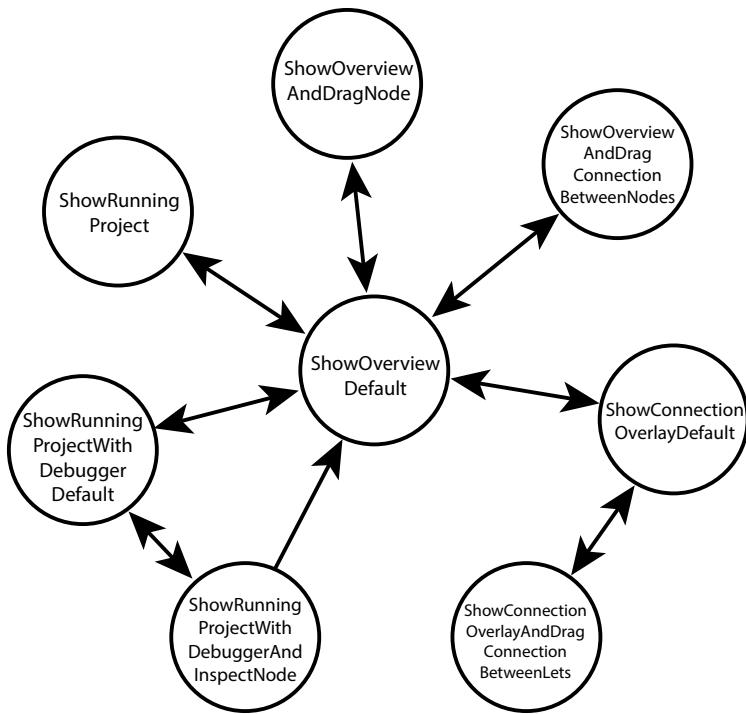
Abb. 3.5. Zeichen-Methoden

### ShowStates

Das Hauptfenster befindet sich zu jedem Zeitpunkt in einem **ShowState**. Anhand des Showstates wissen die verschiedenen Zeichen-Methoden (Abb. 3.5), welche Komponenten zu welchem Zeitpunkt gezeichnet werden müssen. ShowStates können nicht in beliebiger Reihenfolge durchlaufen werden, sondern können jeweils immer nur in bestimmte Zustände wechseln (Abb. 3.6).

## Entwickelte Komponenten

Auch wenn verschiedene der hier vorgestellten Komponenten nur in einem Fenster / Dialog verwendet werden, lag der Fokus bei diesen darauf, diese Komponenten allgemein zu entwickeln, sodass sie einfach für andere Applikationen wiederverwendet werden können.

**Abb. 3.6.** ShowStates*InputKonzept*

Damit eigene Komponenten, die nicht von der Swing-Klasse `JPanel` ableiten, auf Benutzereingabe reagieren können habe ich eine eigene Klasse `InputManager` entwickelt. An dieser können sich Komponenten anmelden welche die eigene Schnittstelle `MouseAndKeyboardListener` unterstützen. Diese Schnittstelle ermöglicht es der Klasse `InputManager`, zu entscheiden, welche Benutzereingaben welcher Komponente zuzuordnen sind. Klickt der Benutzer innerhalb eines Fensters auf eine Stelle oder gibt in einem Fenster, das gerade fokussiert ist, ein Symbol über die Tastatur ein, so wird diese Eingabe über ein an diesem Fenster angemeldeten `ActionListener` an den `InputManager` übergeben und an die entsprechende an diesem angemeldete Komponente übergeben.

*Ladebalken***Abb. 3.7.** Ladebalken

Wird das Programm gestartet, wird der Benutzer mit einem Ladebalken (Abb. 3.7) begrüßt. Auch wenn Ladebalken eher unbeliebt sind, so erfüllen diese doch den Zweck, dem Benutzer eine Rückmeldung darüber zu geben, dass das Programm gestartet ist und zeitnah bereitsteht und genutzt werden kann. Auch wenn die Implementierung eines Ladebalken nicht den Anschein hat, technisch komplex zu sein, so gab es trotzdem einige Aspekte bei der Umsetzung von diesem zu beachten.

Ziel eines Ladebalken ist, dem Benutzer möglichst schnell eine Rückmeldung über das aktuelle Programm zu geben. Aus diesem Grund war es mir wichtig, die Zeit zwischen Start der Applikation und Anzeige dieses Ladebalkens zu optimieren. Hierbei ist es interessant, dass auf dem Lade-Screen keine Schriften angezeigt werden, kein Zufall ist, sondern damit zu tun hat, dass das Laden von Schriften unter Java rund 1-3 Sekunden in Anspruch nehmen kann. Die meisten mit Java umgesetzten Programme, die auf dem ersten Screen bereits Text anzeigen, starten aus diesem Grund verzögert. Um diesen Effekt zu vermeiden, habe ich beim Lade-Screen auf die Darstellung von Text verzichtet und lade diese im Hintergrund (indem ich diese in einen Offscreen-Bereich zeichne).

Da der Ladebalken nicht auf Benutzereingaben reagieren muss und den Programmstart möglichst nicht verzögern soll, steht der Code, der für das Zeichnen dieses Balkens zuständig ist, fest programmiert im Programmcode der Zeichenmethode des Hauptpanels.

### Toolbar

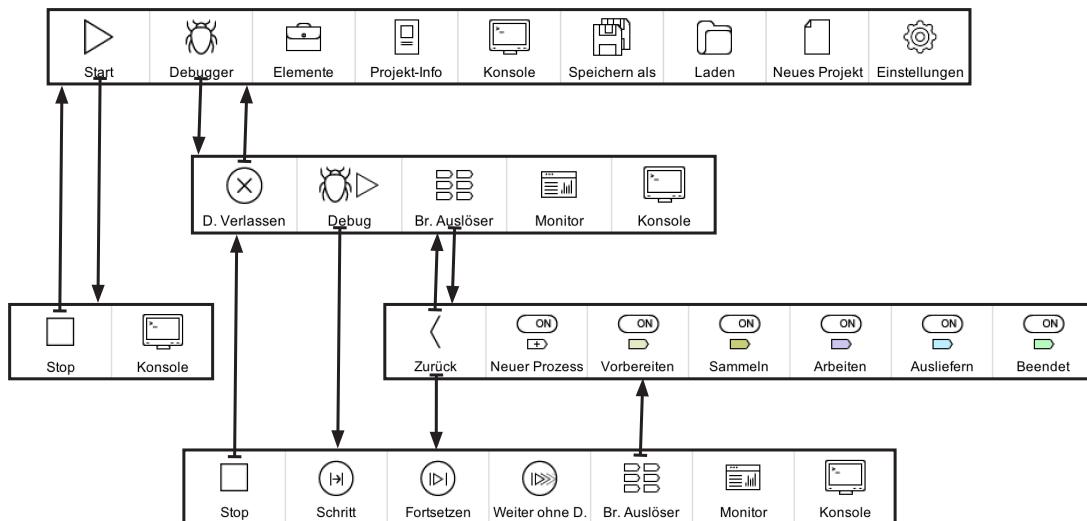


Abb. 3.8. Toolbar Storyboard

Obwohl die Toolbar (Abb. 3.8) als allgemeine Komponente entwickelt wurde, wird sie nur im Hauptfenster verwendet und ist dort eine der wichtigsten Komponenten. Die Toolbar wird durch die gleichnamige Klasse `Toolbar` beschrieben und durch die Klasse `ToolbarDrafter` gezeichnet. Die Toolbar hat keine Position, da sie automatisch davon ausgeht, sich im oberen Bereich der View zu befinden. Sie implementiert das Interface `MouseAndKeyboardListener` und kann damit an einem `InputManager` angemeldet werden. Die Toolbar enthält Instanzen der abstrakten Klasse `ToolbarItem`. Diese setzt sich zusammen aus einem Namen, einem Icon und zwei abstrakten Methoden, die zur Laufzeit entscheiden, ob der Eintrag sichtbar ist und was bei einem Klick passieren soll.

### Scrollbar



Abb. 3.9. Scrollbar

Diese Komponente wird an zwei Stellen verwendet:

- Im Hauptfenster (vertikal und horizontal)
- In der allgemeinen Komponente Konsolen-Panel (nur vertikal)

Eine Scrollbar (Abb. 3.9) wird durch die gleichnamige Klasse `Scrollbar` beschrieben und durch die Klasse `ScrollbarDrafter` gezeichnet. Eine Scrollbar besteht aus einer Scroll-Richtung (vertikal oder horizontal), einer Fläche (`Area`), einer Position (zwischen 0 und 1), und einer repräsentierten Höhe/Breite. Die Komponente implementiert das Interface `MouseAndKeyboardListener` und lässt sich somit an einem `InputManager` anmelden. Ein allgemeines Problem bei der Swing-Scrollbar ist es, dass schlecht kontrollierbare Effekte auftreten, wenn sich die Größe der Fläche, auf welche sie sich bezieht, ändert. Sinn dieser eigenen Komponente ist es, dass sich die repräsentierte Fläche dynamisch ändern kann, ohne dass die Position springt oder sonstige nicht benutzerfreundliche Effekte auftreten. Auch wurde bei der Implementierung der Scrollbar darauf Wert gelegt, dass zwischen Bar und Rahmen das gleiche Verhältnis besteht wie zwischen repräsentierter Höhe/Breite und Komponenten Höhe/Breite. Die Höhe/Breite der Scrollbar entspricht also immer  $\frac{(\text{Scrollbar Größe})^2}{(\text{Repräsentierte Größe})}$ .

### Konsolen-Panel

Diese Komponente wird nicht wie die anderen Komponenten durch eine Zeichnen-Methode gezeichnet, sondern ist eine Swing-Komponente, die von `JPanel` ableitet. Der Grund hierfür liegt darin, dass diese Komponente in Verbindung mit anderen Swing-Komponenten verwendet wird.

### 3.1.8 Persistente Einstellungen

Um Einstellungen persistent speichern zu können, habe ich die Klasse `Settings` entwickelt. Diese Klasse speichert Einstellungen in einer `HashMap<String, String>` und stellt eine Reihe von Methoden zur Verfügung, um bequem auf diese zuzugreifen. Außerdem bietet die Klasse die Möglichkeit, Einstellungen aus einer Datei zu lesen sowie in eine Datei zu schreiben. *Keys* und *Values* werden zeilenweise jeweils wie folgt gespeichert: `key:value`. Vorteil dieser Schreibweise ist dass *value* jeden möglichen String abgesehen vom Zeilenumbruch annehmen kann, ohne dass Zeichen maskiert werden müssen. Dies ist gerade deswegen praktisch, da es möglich sein soll, diese Einstellungen händisch zu ändern, ohne die Applikation starten zu müssen. Auf diese Weise können neue Features testweise aktiviert werden, ohne dafür einen Eintrag in dem Benutzerinterface erstellen zu müssen.

Auch wenn die Klasse `Settings` als allgemeine Komponente entwickelt wurde und keine abstrakten Methoden besitzt, ist sie trotzdem als abstrakt gekennzeichnet. Dies hat den Grund, dass diese Klasse nicht direkt instanziert wird, sondern als Basisklasse für die Klassen `GeneralSettings` und `ProjectSettings` verwendet wird.

Die Klasse `GeneralSettings` verwaltet Einstellungen, die die allgemeine Applikation betreffen, die Klasse `ProjectSettings` hingegen verwaltet Einstellungen, die das aktuelle Projekt betreffen. Zum jetzigen Zeitpunkt gibt es jedoch keine projekt-spezifischen Einstellungen, sodass hier nur technische Parameter die Versionierung betreffend gespeichert werden.

Das folgende Beispiel zeigt, wie die Einstellungs-Datei einer bestehenden Installation aussehen kann:

```
view.usenimbuslaf=false
view.background=default
developer.mousecoordinates=false
view.additionalhelp=true
checkprojectsbeforrun=true
developer.logmore=false
developer.logtofile=false
developer.logonlyerrorstofile=false
lastprojectpath=/Users/christiancolbach/_TestProjects/Grösser-Kleiner-Beispiel
developer.targetcoordinates=false
developer.additionalchecks=false
invisibelprompt=false
experimentalnodedefinitions=true
developer.logredraw=false
developer.advancedtesting=false
startguionstartup=true
checkforunreachablenodes=true
developer.ringstacksizeforexecutionLogger=2000
developer.nodecount=false
view.nodedcollectionautodisposeondoubleclick=true
developer.redrawcounter=false
```

---

```

startpromptonstartup=true
warnifprojectnotsaved=true
composednodedefinitions=true
alternativeassetdirectory=/Users/christiancolbach/GitHub/Bachelor-Projekt/Quellcode
Hauptprogramm/assets
developer.ringstacksizeforgeneralLogger=200

```

### 3.1.9 Protokollierung

Da mir die Funktionalität des Java-eigenen-Loggers (`java.util.logging.Logger`) nicht ausreicht, habe ich mich dazu entschieden, für mein Programm einen eigenen Logger zu entwickeln. Meine Implementierung besitzt folgende Features:

- Zugriff auf die letzten  $n$  Ausgaben (Hierbei ist  $n$  ein zur Laufzeit definierbarer Wert)
- Automatisches Abfangen von Ausgaben über die Systemstreams `System.out` und `System.in` (auf diese Weise kann für Ausgaben weiterhin das gewohnte `System.out.print(...)` verwendet werden).
- Unterscheidung zwischen normalen Ausgaben und Fehlern.
- System, um wichtige Ausgaben von nebensächlichen Ausgaben zu trennen und diese nur bei Bedarf in `System.out` bzw. `System.in` einzuspeisen (Ausgaben müssen mittels `AdditionalLogger.out.println(...)` oder `AdditionalLogger.out.println(...)` ausgegeben werden, Ausgaben werden ignoriert, falls *Mehr Ausgaben* unter *Erweiterte Einstellungen* deaktiviert ist).
- Ausgaben in Echtzeit in eine Logdatei zu schreiben (Falls *Ausgaben in Datei speichern* unter *Erweiterte Einstellungen* aktiviert ist).

Anzumerken ist, dass die Funktionalität der Klasse `AdvancedLogger` nur deswegen in dieser Form möglich ist, weil ein zusätzlicher *PrintStream LogStream* um `System.out` und `System.in` gekapselt wird und wieder auf diese beiden Attribute zurückgeschrieben wird. Auf diese Art und Weise können alle Systemausgaben abgefangen werden, unabhängig davon über welche Methode diese ausgegeben werden.

### 3.1.10 Über Reflexion geladene Klassen und geteilte Klassen/Schnittstellen

Element-Definitionen (Klassen, die das Interface `NodeDefinition` implementieren), die als Grundlage dienen, um Elemente zu erzeugen, werden nicht direkt über den Klassenpfad geladen, sondern mittels Reflexion. Element-Definitionen werden in Instanzen der Klasse `NodeDefinitionsLibrary` geladen und dort gesammelt. Wird eine Instanz der Klasse `NodeDefinitionsLibrary` erzeugt, so wird dem Konstruktor als Parameter ein Pfad mitgegeben. In diesem Pfad wird rekursiv nach Klassen

gesucht. Alle gefundenen Klassen werden instanziert (vorausgesetzt diese besitzen einen Standardkonstruktor) und falls diese die Schnittstelle `NodeDefinition` implementieren zur `NodeDefinitionsLibrary` hinzugefügt. Insgesamt habe ich bis zum jetzigen Zeitpunkt 157 Element-Definitionen implementiert.

### Hilfstoß *Templategenerator*

Um die Entwicklung von Element-Definitionen zu unterstützen, habe ich ein Tool entwickelt, das unter Eingabe der notwendigen Informationen Templates unter einem angegebenen Pfad erzeugt (Abb. 3.10). Auf diese Weise kann die Implementierung neuer Elemente beschleunigt und Flüchtigkeitsfehler vermieden werden.

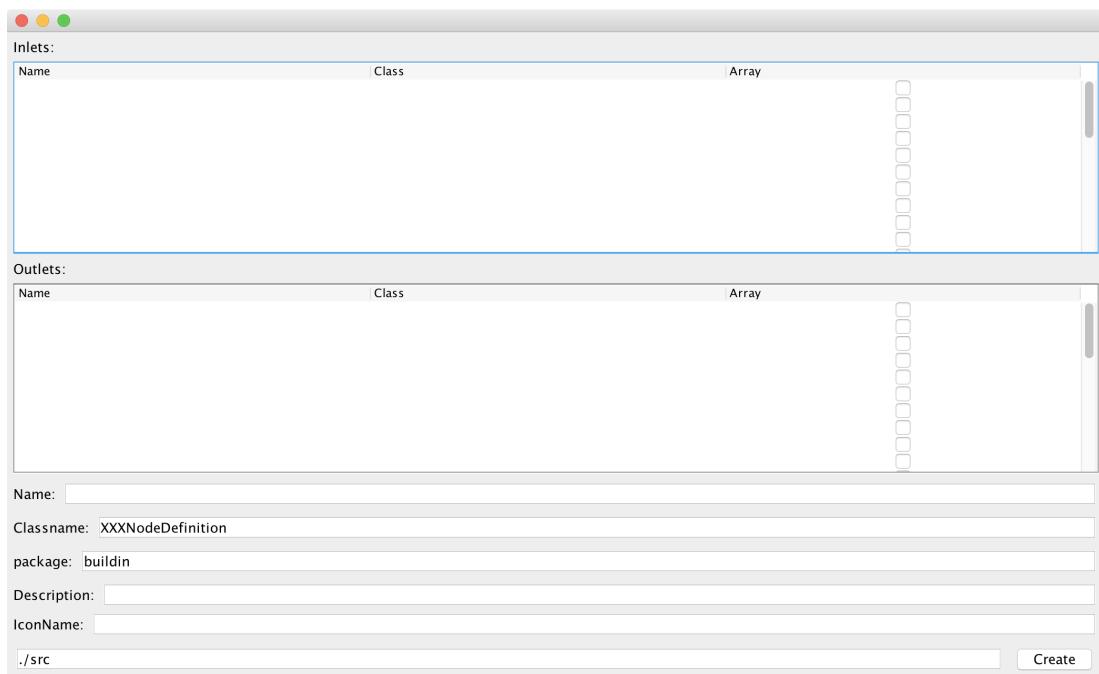


Abb. 3.10. Templategenerator

### 3.1.11 Weitere Konzepte

#### Assets

Um sicher zu stellen, dass das Laden von Grafiken innerhalb der Benutzeroberfläche nicht zu unerwarteten Verzögerungen führt und zu vermeiden dass Bilddateien mehrfach von der Festplatte geladen werden, habe ich die Klasse `ImageAsset` eingeführt. Jede Instanz der Klasse `ImageAsset` steht jeweils für eine Grafik, die

im Speicher vorgeladen ist. Instanzen dieser Klasse können jedoch nicht von außen über den Konstruktor erzeugt werden, da dieser `private` ist und somit nicht von außerhalb verwendet werden darf. Um Instanzen der Klasse zu erzeugen, muss die Fabrik-Methode `getImageAssetForName(String asset)` verwendet werden. Diese prüft ob bereits ein `ImageAsset` dieser Grafik existiert. Existiert dieser nicht, so wird dieser erzeugt und zurückgegeben. Um jedoch Verzögerungen beim ersten Laden eines `ImageAsset` zu vermeiden, werden beim Programmstart alle im definierten Assets-Verzeichnis befindlichen Dateien vorgeladen.

### 3.1.12 Verwendete Bibliotheken

Folgende fremde Bibliotheken werden von der Applikation oder der dazugehörigen Element-Definitionen verwendet:

- Apache Commons IO: Zum vereinfachten Lesen und Schreiben von Dateien.
- Jackson: Zum Lesen und Schreiben von JSON-Dateien.
- QRGen: Zur Erzeugung von QR-Codes.
- JavaMail: Zum Senden von Emails.
- JavaCV: Zum Zugriff auf das OpenCV-Framework und damit verbunden zur Verwendung von angeschlossenen Webcams.

## 3.2 Verwendete und erstellte Grafiken

Die verwendeten Grafiken für Icons wurden teilweise selbst erstellt, stammen aus einem Iconpack, den ich von Icons Mind<sup>2</sup> erworben habe oder bauen auf diesem auf.

---

<sup>2</sup> <https://iconsmind.com>

# **4**

---

## **Analyse**

### **4.1 Vorteile des Programmierkonzeptes**

#### **4.1.1 Parallelisierung**

Ein Nebeneffekt des Konzeptes, anhand dessen Projekte ausgeführt werden, ist, dass die gesamte Ausführung (soweit möglich) parallel stattfindet, ohne dass das Projekt vom Benutzer gesondert darauf ausgelegt werden muss. Dies führt gerade bei Mehrkern-Prozessoren zu einer besseren Effizienz gegenüber Programmen, die nicht parallelisiert sind.

#### **4.1.2 Keine Programmierkenntnisse notwendig**

Ein Vorteil, den dieses grafische Programmierkonzept gegenüber vielen anderen grafischen Programmierkonzepten hat, ist, dass es nicht versucht, stumpf ein Benutzerinterface um ein bestehendes textbasiertes Programmierparadigma herum zu bauen sondern ein eigenes darstellt. Dies ist gerade für Benutzer von Vorteil, die über keine Programmierkenntnisse verfügen.

#### **4.1.3 Schnelle Ergebnisse**

Ein weiterer Vorteil dieses grafischen Programmierkonzeptes ist, dass gerade einfache Tätigkeiten ohne einen großen Aufwand erledigt werden können. Tätigkeiten, bei denen es bei anderen Programmierkonzepten nötig ist, erst ein Rahmenprogramm zu entwickeln, bevor erste Ergebnisse sichtbar sind, können in meinem Programmierkonzept mit dem Verbinden von nur wenigen Elementen bedient werden.

#### **4.1.4 Einfache Erweiterung**

Da Elementdefinitionen erst zur Laufzeit geladen werden und nicht Teil des Programmcodes sind, ist die Programmierumgebung leicht um Elemente zu erweitern.

Steht einmal kein passendes Element zur Verfügung, so kann dieses einfach zur Umgebung hinzugefügt werden.

## 4.2 Nachteile des Programmierkonzeptes

### 4.2.1 Begrenzter Baukasten

Die Auswahl an Elementen ist begrenzt. Bei der Umsetzung von Projekten entstehen häufig Situationen, in denen benötigte Elemente fehlen. Dieses Problem wird sich dadurch vermindern lassen, dass mit der Zeit neue Element-Definitionen hinzugefügt werden, jedoch wird dadurch dass Element-Definitionen immer nur sehr spezielle Fälle abdecken können, das grundsätzliche Problem bestehen bleiben. Es wird nicht zu verhindern sein das für bestimmte Fälle kein passendes Element vorhanden sein wird.

### 4.2.2 Eingeschränkter Funktionsumfang des Benutzerinterface

Auch wenn bereits viel Aufwand in der Entwicklung des Benutzerinterfaces steckt, so ist dieses noch sehr weit davon entfernt, dass es mit herkömmlichen Entwicklungsumgebungen mithalten kann. Viele Funktionen (wie Copy-Paste, Shortcuts, automatische Updates oder anpassbare Bedienelemente) welche in etablierten Entwicklungsumgebungen vorhanden sind, fehlen.

### 4.2.3 Hoher Ressourcenverbrauch

Durch den bei der Ausführung großen Verwaltungsaufwand und die hohe Anzahl parallel laufender Threads, ist die Ausführung in den meisten Situationen sehr träge. Auch wenn das Konzept durch eine hohe Parallelisierung punkten kann, so wird dieser Geschwindigkeitsvorsprung gerade bei Projekten, die aus vielen einzelnen Elementen bestehen größtenteils wieder aufgefressen. Das Hauptproblem entsteht dadurch, dass für den Rechner eigentlich einfache Aufgaben, wie die Addition von zwei Zahlen, den gleichen Verwaltungsaufwand benötigen wie Operationen, die insgesamt viel komplexer sind.

### 4.2.4 Fehlen von Funktionalitäten konventioneller Programmierparadigmen

In vielen Situationen scheinen Konzepte aus konventionellen Programmiersprachen zu fehlen. Dies hat zur Folge, dass oft kreatives Umdenken erforderlich ist, um ohne diese auszukommen und manche Lösungen deswegen auf den ersten Blick für Programmierer umständlich wirken.

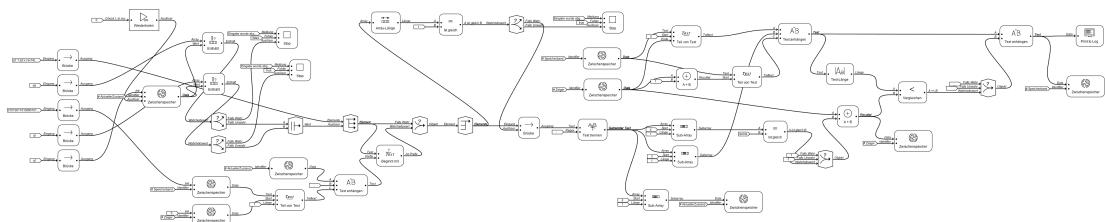
## 4.3 Turing-Vollständigkeit

Schlussendlich drängt sich noch die Frage auf, ob das entwickelte Programmierkonzept Turing-vollständig ist. Dies ist in erster Linie deswegen interessant, weil Turing-Vollständigkeit impliziert, dass sich (bei unbegrenztem Speicher und Rechenzeit) theoretisch jedes programmiertechnisch lösbarer Problem damit lösen lässt.

Allgemein ist ein erster Ansatzpunkt, um intuitiv zu bestimmen, ob eine Sprache Turing-vollständig ist, die Frage, ob eine Sprache entweder (1) While-Konstrukte oder (2) If-Konstrukte in Verbindung mit Goto-Befehlen besitzt. Dies ist auf den ersten Blick **nicht** der Fall. Jedoch gibt es hier ein Konstrukt, das sich auf eine ähnliche Art und Weise verwenden lässt. Durch die Verwendung eines Wiederholen-Elementes und If-Elementen lässt sich ein While artiges Konstrukt zusammenbauen. Aus diesem Grund könnte das entwickelte Programmierkonzept Turing-vollständig sein.

### 4.3.1 Beweis Turing-Vollständigkeit

Um den Beweis zu liefern, dass das entwickelte Programmierkonzept Turing-vollständig ist, habe ich mit meinem Programmierkonzept eine *Universelle Turingmaschine* aufgebaut (Abb. 4.1). Diese erhält auf der linken Seite eine beliebige Turingmaschine in kodierter Form und führt diese aus. Das Projekt steht auf GitHub zum Download<sup>1</sup> bereit.



**Abb. 4.1.** Umsetzung Universelle Turingmaschine

Konkret verarbeitet das Projekt eine Turingmaschine, die in der Form, wie in Abb. 4.2 dargestellt, kodiert ist. Die Existenz einer universellen Turingmaschine innerhalb meines Programmierkonzept beweist, dass dieses Turing-vollständig ist.

<sup>1</sup> <https://github.com/colbach/Bachelor-Projekt/tree/master/Beispiele/Universelle%20Turingmaschine>

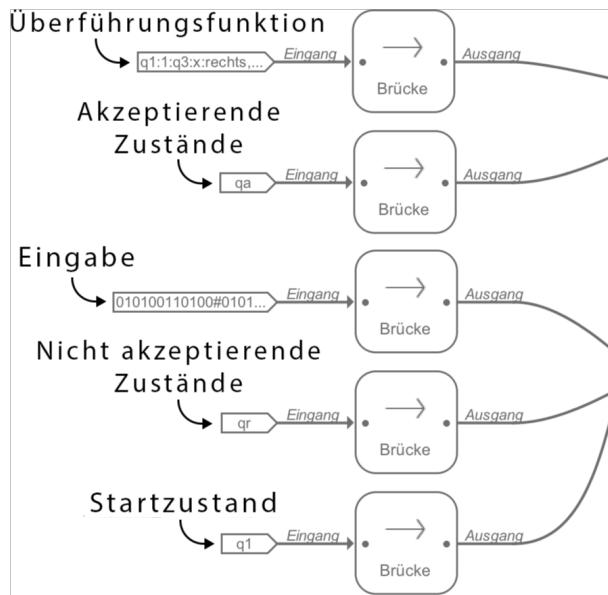


Abb. 4.2. Eingaben Universelle Turingmaschine

### 4.3.2 Beispiel

Als Beispiel habe ich folgende Turingmaschine zum Testen verwendet: Die Turingmaschine soll bestimmen, ob zwei binäre Folgen, die mit einem # voneinander getrennt sind, gleich oder nicht gleich sind. 010100110100#010100110100 soll also von der Turingmaschine akzeptiert werden und 010100110100#010110110110 soll abgelehnt werden. Realisiert werden kann dies über folgende Turingmaschine:

#### Überführungsfunction

```
q1:1:q3:x:rechts, q1:0:q2:x:rechts, q1:#:q8:#:rechts, q2:0:q2:0:rechts,
q2:1:q2:1:rechts, q2:#:q4:#:rechts, q3:0:q3:0:rechts, q3:1:q3:1:rechts,
q3:#:q5:#:rechts, q4:x:q4:x:rechts, q4:0:q6:x:links, q5:x:q5:x:rechts,
q5:1:q6:x:links, q6:0:q6:0:links, q6:1:q6:1:links, q6:x:q6:x:links,
q6:#:q7:#:links, q7:0:q7:0:links, q7:1:q7:1:links, q7:x:q1:x:rechts,
q8:x:q8:x:rechts, q8:_:qa:_:rechts
```

#### Akzeptierende Zustände

qa

#### Eingabe (zum Testen)

010100110100#010100110100

#### Nicht akzeptierende Zustände

qr

### Startzustand

q1

Ergebnis (nach Ausführung der Testeingabe)

Ergebnis der Ausführung ist, dass die Eingabe akzeptiert wurde (Abb. 4.3).

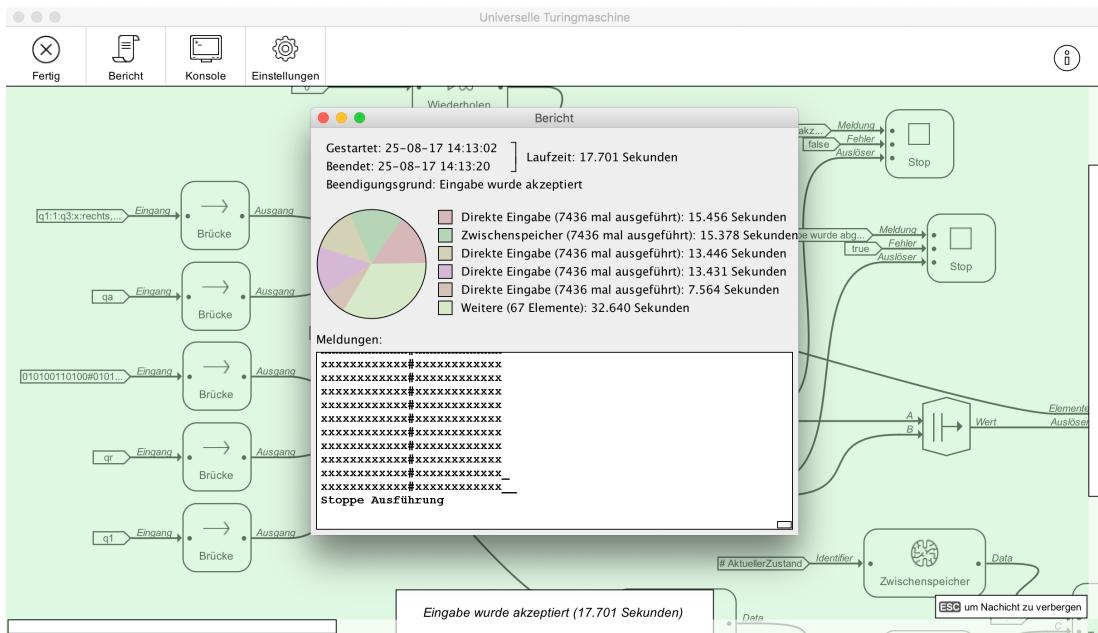


Abb. 4.3. Ergebnis der Ausführung

Anzumerken ist, dass meine Implementierung nicht besonders effizient ist, dies war jedoch auch nicht das Ziel bei der Umsetzung einer Universellen Turingmaschine. Konkret hat die vollständige Ausführung dieser Turingmaschine mit einer Eingabelänge von 25 Zeichen 17 Sekunden benötigt.

# **5**

---

## **Fazit**

Bei dieser Arbeit wurde ein alternatives grafisches Programmierkonzept ausgearbeitet sowie die dazugehörige Software zu diesem umgesetzt.

Auch wenn das entwickelte Programmierkonzept seine Schwächen hat, habe ich jedoch feststellen können, dass das Konzept insgesamt verwendbar ist und hilfreich sein kann, um Aufgaben schnell und einfach zu lösen.

# 6

---

## Ausblick

### 6.1 Optimierung

Auch wenn bereits viel Zeit in die Optimierung meines Programmcodes geflossen ist, gibt es einige Punkte, die noch verbesserungswürdig sind.

#### 6.1.1 Verwendung von Threadpools

Ich hatte mich zu Beginn dazu entschieden, mit vielen einzelnen Threads, anstelle von Threadpools, zu arbeiten. Der Grund hierfür lag darin, dass es mir am Anfang der Entwicklung am wichtigsten erschien, möglichst schnell einen funktionierenden Prototypen zu haben. Nachdem die Ausführungsumgebung immer mehr an Umfang gewonnen hatte, wurde es schwerer und schwerer die Threads in Threadpools zu organisieren. Da immer genug neue Features auf meiner To-Do-Liste standen, ist der Punkt Threadpools immer weiter nach hinten in meiner Prioritätenliste gewandert. Diesen Punkt möchte ich bei Möglichkeit gerne noch aufarbeiten, um eine stabile Projektausführung auch bei mehreren Tausend parallelen Threads zu garantieren.

#### 6.1.2 Verbesserung der automatischen Kompatibilität

Um die Kompatibilität zwischen Elementen zu erhöhen, habe ich ein Modul entwickelt, das Datentypen automatisch untereinander konvertiert. Dies funktioniert prinzipiell sehr gut, jedoch besteht in der aktuellen Version das Problem, dass diese Konvertierung immer auf Basis der Typen der Ein- und Ausgänge arbeitet. Dies klingt im ersten Moment nicht nach einem Problem, jedoch hat sich gezeigt, dass hierbei Inkompabilitäten auftreten können, wenn Daten über Elemente weitergereicht werden, die als Typ eine Superklasse des entgegengenommenen Datentyps haben. Um dieses Problem zu umgehen, möchte ich das System, das für diese Konvertierung zuständig ist, grundlegend neu organisieren um zu ermöglichen, dass diese Konvertierung erst während der Laufzeit anhand der real übergebenen Daten durchgeführt wird und nicht anhand der Datentypen der Ein- und Ausgänge.

## 6.2 Ausbau des Frameworks

Grundsätzlich ist der Umfang der Element-Definitionen noch sehr eingeschränkt, viele Aufgaben können mit dem aktuellen Baukasten unmöglich durchgeführt werden. Mein Ziel ist es, mein Programm zu verwenden, um weitere Projekte damit umzusetzen.

## 6.3 Erweiterung der grafischen Benutzeroberfläche

Obwohl der größte Teil des Aufwandes bei der Entwicklung in die Benutzeroberfläche geflossen ist, gibt es eine Reihe von Features , die ich mir selbst noch wünsche. Folgende Features plane ich, zu implementieren:

- Grafische Ansicht um Projektversionen zu verwalten
- Möglichkeit Änderungen rückgängig zu machen mit *ctrl+z*
- Eigener Profiler

## 6.4 Bündeln diverser Teilmodule als Library

Auch wenn bereits viel Energie in die Entwicklung der Anwendung geflossen ist, bin ich mir bewusst, dass es bis zu einer produktiv nutzbaren Version noch viel Arbeit ist. Worin ich momentan jedoch bereits einen größeren Wert sehe ist in der Funktionalität diverser Teilmodule. Mein Plan ist es, Teile meines Programms in eine eigene Library zu packen und sie dann als Grundlage für zukünftige Projekte zu verwenden und diese mit der Zeit weiter zu entwickeln. Folgende Teilmodule finde ich für dieses Vorhaben interessant:

- Logger (`logging.*`)
- Kommandozeile (`commandline.*`)
- Grafische Konsole (`view.console`)
- Persistente Einstellungen (`settings.*`)
- Eigene grafische Komponenten (u. a. `view.sharedcomponents.*`)
- Utility-Klassen (`utils.*`)

**A**

---

**Erklärung des Kandidaten**

Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen- und Hilfsmittel verwendet.

---

Datum

Unterschrift des Kandidaten