

FlowDesign

Projekt Softwaretechnik

Wintersemester 2016



Studiengang
Softwaretechnik und Medieninformatik

Michael Watzko (749567)

Oliver Wasser (749384)

Julian Klissenbauer-Mathä (749564)

Inhaltsverzeichnis

1 Einleitung	5
1.1 Motivation	5
1.2 Rollenverteilung	5
2 Umsetzung	6
2.1 Entwicklungsansatz	6
2.2 Versionsverwaltung	6
2.3 Entwicklungsumgebung	6
2.4 Lizenz	6
3 FlowDesign	7
3.1 System-Umwelt Diagramm	7
3.2 Maskenprototyp	7
3.3 Fluss Diagramm	7
3.3.1 Elemente	8
3.3.2 Notation	11
4 Anforderungen	12
4.1 Funktionale Anforderungen	12
4.1.1 Diagramme	12
4.1.2 Plattformunabhängigkeit	12
4.1.3 Projektbasiertes Arbeiten	12
4.1.4 Verlinkbarkeit	12
4.1.5 Einhaltung von Verknüpfungslogiken	12
4.1.6 Codegenerierung (optional)	12
4.2 Nicht-Funktionale Anforderungen	13
4.2.1 Aussehen und Handhabung	13
4.2.2 Fehlertolleranz bzw. Fehlerabsicherung	13
4.2.3 Skalierbare Darstellung	13
5 Projektplan	14
5.1 Implementierung - 25.10.16 - 12.12.16	14
5.2 Abnahme IT-Designers - 13.12.16 - 23.12.16	14
5.3 Abschlusspräsentation - 02.01.17 - 10.01.17	14
5.4 Repository Übergabe - 23.01.17	14

6 Protokoll zum Kundentreffen am 13. Dezember 2016	15
7 Protokoll zum Kundentreffen am 22. Dezember 2016	16
8 Module	18
8.1 Modulübersicht	18
8.2 Modul Core	18
8.3 Modul Data	19
8.3.1 Bindings	19
8.3.2 Flow Notation Parser	20
8.4 Modul Data-UI	21
8.5 Modul JavaFX	22
8.5.1 Erscheinungsbild	22
8.5.2 Übersetzung	22
8.6 Modul Model	23
8.7 Modul Storage	23
8.7.1 StorageHandler	23
8.7.2 Storage	24
8.7.3 Serializer	24
9 Einhaltung Programmierkonzepte	25
9.1 SOLID	25
9.1.1 Single-Responsibility-Prinzip	25
9.1.2 Open-Closed-Prinzip	25
9.1.3 Liskovsches Substitutionsprinzip	25
9.1.4 Interface-Segregation-Prinzip	26
9.1.5 Dependency-Inversion-Prinzip	26
10 Aufbau Diagramm-Element	27
10.1 Properties	27
10.2 Joint	27
10.3 Joint-Group	27
11 Erweiterung	28
11.1 Erweiterung um einen Diagrammtyp	28
11.1.1 Datenmodell	28
11.1.2 View	28
11.1.3 Handler	29

11.1.4 Serialisierung	30
11.1.5 Strings	34
11.2 Erweiterung um ein Diagramm-Element	35
11.2.1 Datenmodell	35
11.2.2 View	36
11.2.3 Image	37
11.2.4 Factory	38
11.2.5 Serialisierung	39
11.3 Weitere Referenzen	42
12 Benutzerhandbuch	43
12.1 Projektauswahlfenster	43
12.1.1 Öffnen eines kürzlich erstellten Projekts	43
12.1.2 Öffnen eines beliebigen Projekts	43
12.1.3 Erstellen eines neuen Projekts	44
12.2 Projektfenster	44
12.2.1 Menüleiste	44
12.2.2 Projektbaum und Anlegen neuer Diagramme	46
12.2.3 Datentypen bearbeiten	47
12.3 Zeichenfläche	49
12.3.1 Erstellen von Verbindungen	51
12.3.2 Fortgeschrittene Programmfunctionen	53
12.3.3 Ändern des Programmdesigns und Suche	58
13 Schlusswort	60
Abbildungsverzeichnis	61
Literatur	63

1 Einleitung

Für das Konstruieren benötigt jeder Ingenieur ein strukturiertes Vorgehen. FlowDesign soll genau dort unterstützend wirken. Zusammen mit der Firma IT-Designers GmbH hat sich unser Team im Wintersemester 2016/17 im Rahmen des Projekts Softwaretechnik an die Entwicklung eines Programms, das diesen Ansatz nutzt, gewagt.

1.1 Motivation

Wie erwähnt entstand das Projekt FlowDesign im Modul "Projekt Softwaretechnik" im vierten Semester des Studiengangs Softwaretechnik und Medieninformatik.

Um zu erfahren wie anspruchsvolle Entwicklungs- und Projektarbeit aussieht, erschien uns dieses Projekt als genau richtig. Es bietet ein relativ offenes Themenfeld, was großen Handlungsspielraum zulässt um unsere gewählten Entwicklungsansätze in der Praxis testen zu können. Daraus erhoffen wir uns unsere Fähigkeiten, sei es in der Entwicklung, Planung und generellen Zusammenarbeit mit dem Kunden oder im Team, weiterentwickeln zu können.

1.2 Rollenverteilung

Die Rollenverteilung für das Projekt sah dabei wie folgt aus:

Kevin Erath	Kunde	IT-Designers
Öhmer Haybat	Betreuer	IT-Designers
Andreas Rössler	Betreuer Professor	HS-Esslingen
Oliver Wasser	Projektmanager und Dokumentation	
Julian Klissenbauer-Mathä	Chefentwickler	
Michael Watzko	Entwickler, Qualitätsmanager und Dokumentation	

2 Umsetzung

2.1 Entwicklungsansatz

Wir haben uns entschieden eine völlig neue Entwicklung zu beginn, da uns die gewählte Programmiersprache des vorherigen Teams, welches sich mit FlowDesign beschäftigte, nicht zugesagt hatte. Mit Java glauben wir Vorteile etwa im Bezug auf Plattformunabhängigkeit zu haben, welche mit C# und dem .NET-Framework nur schwer zu realisieren wären.

Unsere Implementierung ist demnach in Java geschrieben und benutzt das UI-Toolkit JavaFX für die Gestaltung der Oberfläche.

Einzig das Konzept des UI wurde von uns grob dem vorherigem Team nachempfunden.

2.2 Versionsverwaltung

Für die Versionverwaltung haben wir uns für Git entschieden. Hierbei haben wir auf einen bereits bestehenden, eigenen Git-Server auf Basis von Gitblit gesetzt. Hierdurch hatten wir zu jeder Zeit einen Überblick über die Aktivitäten an der Repository und konnten uns schnell über die letzten Änderungen informieren.

2.3 Entwicklungsumgebung

Als Entwicklungsumgebung haben wir IntelliJ IDEA eingesetzt. Nichtsdestotrotz wurde das Projekt als Maven-Projekt erstellt, wodurch die Maven-Ordnerstruktur als Grundaufbau verwendet wurde. Auch alle Abhängigkeiten wurden durch Maven aufgelöst. Somit ist es grundsätzlich möglich das Projekt auch ohne Entwicklungsumgebung zu übersetzen und auszuführen.

2.4 Lizenz

Um zukünftigen Teams, die sich mit der gleichen Aufgabe beschäftigen, eine Grundlage zu geben haben wir uns dazu entschlossen den Quelltext unter der GPL Lizenz zu veröffentlichen. Zudem wird der Code jederzeit online abrufbar sein.

3 FlowDesign

Bei FlowDesign handelt es sich um eine Methodik, welche dem Entwickler helfen soll im Voraus saubere Software zu entwerfen. Dabei hilft FlowDesign etwa bei der Vermeidung von Abhängigkeiten innerhalb eines Programms durch Entkopplung in einzelne Module, was ansonsten zu Problemen führen könnte. Datenflüsse werden betrachtet und passende Schnittstellen definiert.

3.1 System-Umwelt Diagramm

Beim System-Umwelt Diagramm handelt es sich um eine Betrachtung eines gegebenen oder geplanten Systems, in welcher Interaktionen mit anderen Systemen, Ressourcen und/oder Akteuren dargestellt werden können. Ergänzend hierzu wurde vom Kunden gewünscht, dass bei Interaktionen genauere Details in Listenform hinzugefügt werden können. Wie im Protokoll zum Kundentreffen am 13. Dezember 2016 (6) aufgezeichnet, sollen die einzelnen Interaktionen in der Liste auch mit den passenden Stellen der anderen Diagrammtypen verlinkt sein.

3.2 Maskenprototyp

Der Maskenprototyp ist besonders wichtig, um eine grobe Vorstellung zu geben wie das fertig Programm in etwa auszusehen hat. Dieser wird idealerweise in interaktiver Zusammenarbeit, etwa in Kundengesprächen, erstellt. Dabei wurde von unserem Kunden gewünscht, dass bewusst nur rudimentäre Oberflächen erstellbar sind. Damit soll verhindert werden den Eindruck zu vermitteln, dass das Programm bereits fertiggestellt sei.

3.3 Fluss Diagramm

Beim Fluss Diagramm wird der Ablauf eines Algorithmus dargestellt. Ein Ablauf kann für sich alleine stehen, oder anhand des Namens mit einer Aktion eines Maskenprototyp oder als Teil eines anderen Algorithmus verknüpft werden. Ein Flow Diagramm besteht aus Operationen die aneinandergereiht sind, dabei ist das Ergebnis einer Operation der Parameter für die nächste Operation. Im folgenden wird auf die Operationen und auf die Notation eingegangen die mit diesem Projekt dargestellt werden kann. Als Grundlage hat hierbei das "CheatSheet Flow Design" [5] gedient. Zuerst werden die einzelnen Elemente erläutert (3.3.1) und anschließend die Notation erklärt (3.3.2).

3.3.1 Elemente

Ein Fluss Diagramm hat immer ein Anfang (kleiner Kreis ohne Text) und ein Ende (kleiner Kreis mit X), zwischen dem die Operationen aufgeführt werden.

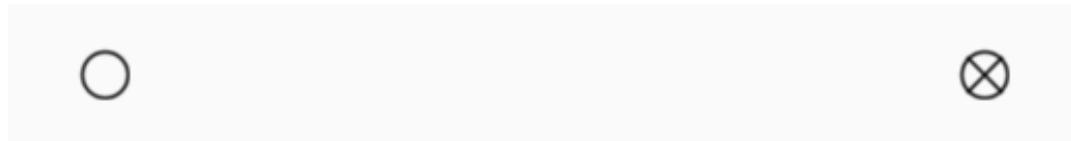


Abbildung 1: Start-Element links, End-Element rechts

Eine Operation ist ein größerer Kreis mit dem Namen oder Beschreibung der hier auszuführenden Operation.

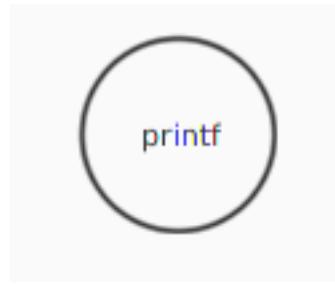


Abbildung 2: Operation-Element

Eine Operation kann auch einen Zustand speichern, der über weitere Aufrufe hinweg persistent ist. Dabei wird ein Container in die untere rechte Ecke gezeichnet. Dem Zustand kann auch ein Name gegeben werden.

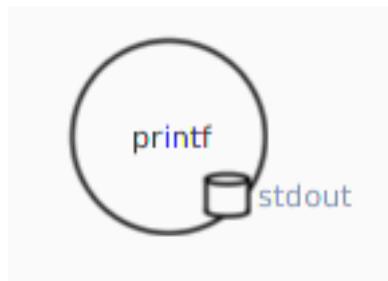


Abbildung 3: Operation-Element mit Zustandsvariable

Falls eine Operation auf eine Ressource zugreift, wird dies durch ein Dreieck in der unteren rechten Ecke gezeichnet. Auch die Ressource kann benannt werden.

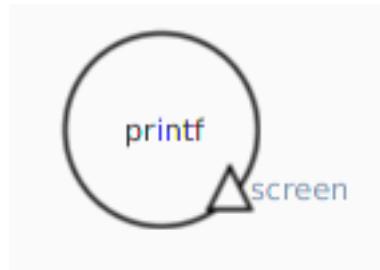


Abbildung 4: Operation-Element mit Ressourcenzugriff

Ein Split-Element stellt die Aufteilung des Programmflusses dar. Das Element wird durch ein Viereck dargestellt unter dem eine Bezeichnung geschrieben werden kann.

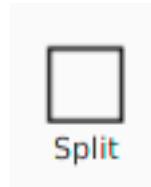


Abbildung 5: Split-Element

Zum zusammenführen eines Programmflusses wird ein Verbindungselement benötigt. Dies ist ein Strich bei dem mehrere Flüsse (Abbildung 6 links) wieder zu einem zusammengeführt werden (Abbildung 6 rechts).

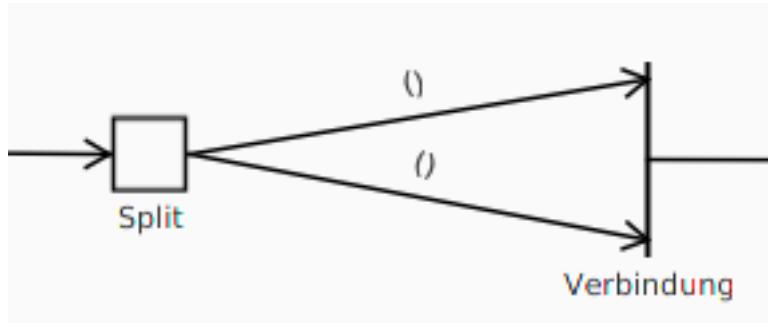


Abbildung 6: Verbindungs-Element mit Beispiel

Zugriffe auf das Client-System (UI, GUI, WebService, etc [5]) werden durch ein Portal dargestellt. Ein Portal ist auch ein Viereck, bei dem sich die Bezeichnung jedoch in dem Viereck befindet. Falls das Fluss Diagramm von einem Client-System verwendet wird, wird auch das Start und Ende Element durch ein Portal dargestellt.



Abbildung 7: Portal-Element

Die einzelnen Elemente werden durch Pfeile verbunden auf denen die Übergabedatentypen geschrieben werden. Eine genauere Erklärung für die Notation kann bei 3.3.2 gefunden werden. Optional kann auch an jede beliebige Stelle eine Kommentarbox eingefügt werden.

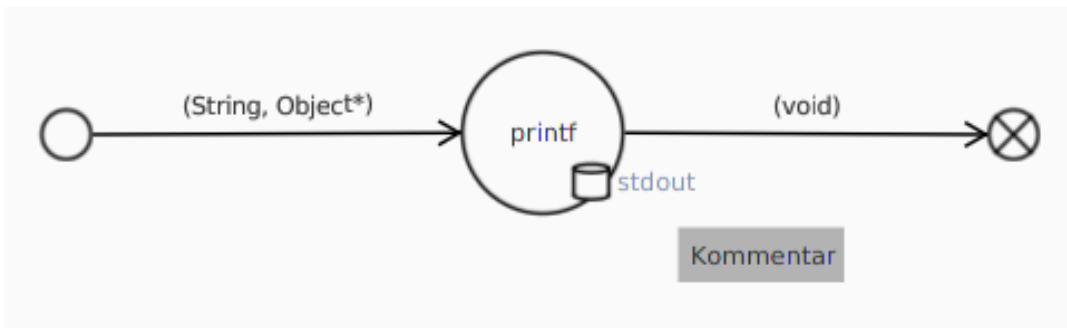


Abbildung 8: Beispielhafte Verknüpfung mit Kommentarbox

Abhängigkeiten und Schleifen können durch vertikale Verbindungslien dargestellt werden. Diese Abhängigkeit wird nicht durch einen Pfeil, sondern durch eine Linie mit einem Punkt am Ende dargestellt; wobei der Punkt bei der Operation ist, von der die andere abhängig ist.

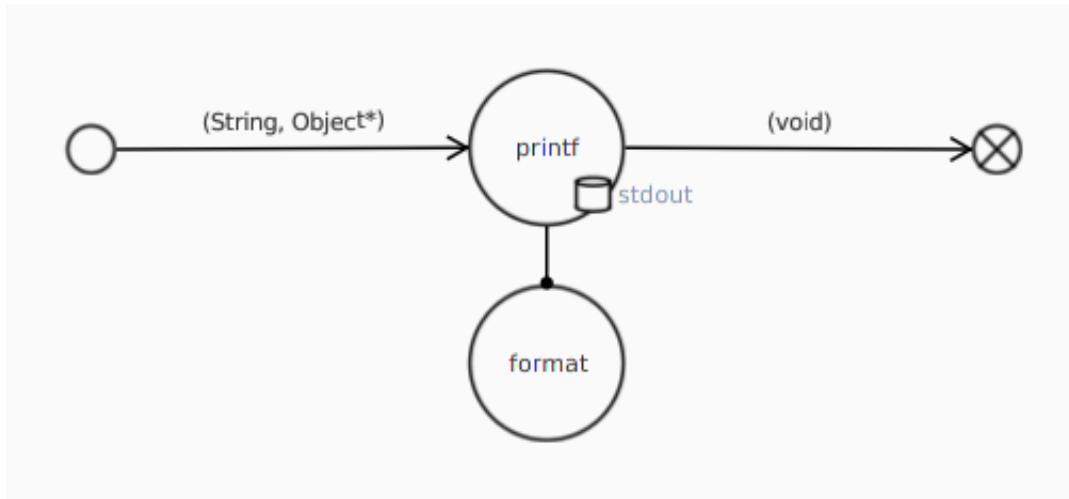


Abbildung 9: Beispielhafte Verknüpfung mit Abhängigkeit

3.3.2 Notation

Die Datentypen die sowohl im "CheatSheet Flow Design" [5] als auch geparsed werden können sind in der folgenden Tabelle aufgelistet. Wie bereits in 3.3.1 erwähnt, werden diese auf die Pfeile geschrieben.

Notation	Erklärung
(y)	Ein Parameter vom Typ "y"
(x, y)	Ein Parameter vom Typ "x" und ein weiterer vom Typ "y"
(y*)	Liste vom Typ "y"
(x, y*)	Ein Parameter vom Typ "x" und eine Liste vom Typ "y"
{y}	Mehrfachaufruf mit einem Parameter vom Typ "y"

Abbildungen 8 und 9 zeigen wie die Operation "printf" mit der folgenden Datentyp Notation aufgerufen wird: "(String, Object*)". Dies bedeutet, dass ein String und eine Liste von Objecten übergeben wird.

4 Anforderungen

Hervorgehend aus der Präsentation durch IT-Designers und unseren eignen Vorstellungen zum Projekt haben wir folgende Requirements ausgearbeitet:

4.1 Funktionale Anforderungen

4.1.1 Diagramme

Folgende Diagramme sollen dargestellt werden können:

- System-Umwelt-Diagramme
- Maskenprototypen
- Flow Diagramme

4.1.2 Plattformunabhängigkeit

Durch Entwicklung in Java soll das Programm auf Windows, Mac OS und anderen Unix-Systemen lauffähig sein.

4.1.3 Projektbasiertes Arbeiten

Diagramme werden einem Flow-Projekt zugeordnet. Neue Diagramme können jederzeit hinzugefügt und entfernt werden. Projekte sollen einfach importiert und exportiert werden können bzw. Projekte sollen abgespeichert und geladen werden können.

4.1.4 Verlinkbarkeit

Programmablauf für z.B. einen Button im Maskenprototyp sollen in den anderen Diagrammarten dargestellt werden können, eine automatische Verlinkung soll stattfinden.

4.1.5 Einhaltung von Verknüpfungslogiken

Verlinkungen sollen automatisch auf Korrektheit überprüft werden. Ungültige Verbindungen, also zwischen inkompatiblen Elementen, sollen blockiert werden.

4.1.6 Codegenerierung (optional)

Optional soll nach Erfüllung bereits genannter Requirements innerhalb des Projektplans versucht werden eine einfache Codegenerierung aus Diagrammen zu implementieren.

4.2 Nicht-Funktionale Anforderungen

4.2.1 Aussehen und Handhabung

Die Oberfläche soll leicht verständlich und intuitiv bedienbar sein.

4.2.2 Fehlertoleranz bzw. Fehlerabsicherung

Wenn der Benutzer falsche bzw. ungültige Eingaben macht, so soll dieser darauf hingewiesen werden.

4.2.3 Skalierbare Darstellung

Die Anwendung soll sowohl bei Auflösungen wie 1024x768 als auch bei Auflösungen wie 1920x1080 korrekt dargestellt werden. Zudem soll die Darstellung auf Retina-Displays bzw. im Allgemeinen bei aktiver Oberflächenskalierung fehlerfrei sein.

5 Projektplan

5.1 Implementierung - 25.10.16 - 12.12.16

- Versioning mit Git
- Implementierung der GUI mit JavaFX
- Implementierung der Verknüpfungslogik und Bedingungen
- Implementierung des persistenten Speichers

5.2 Abnahme IT-Designers - 13.12.16 - 23.12.16

- Treffen mit dem Kunden
- Fehlerkorrekturen
- Kleinere Anpassungen und Ergänzungen

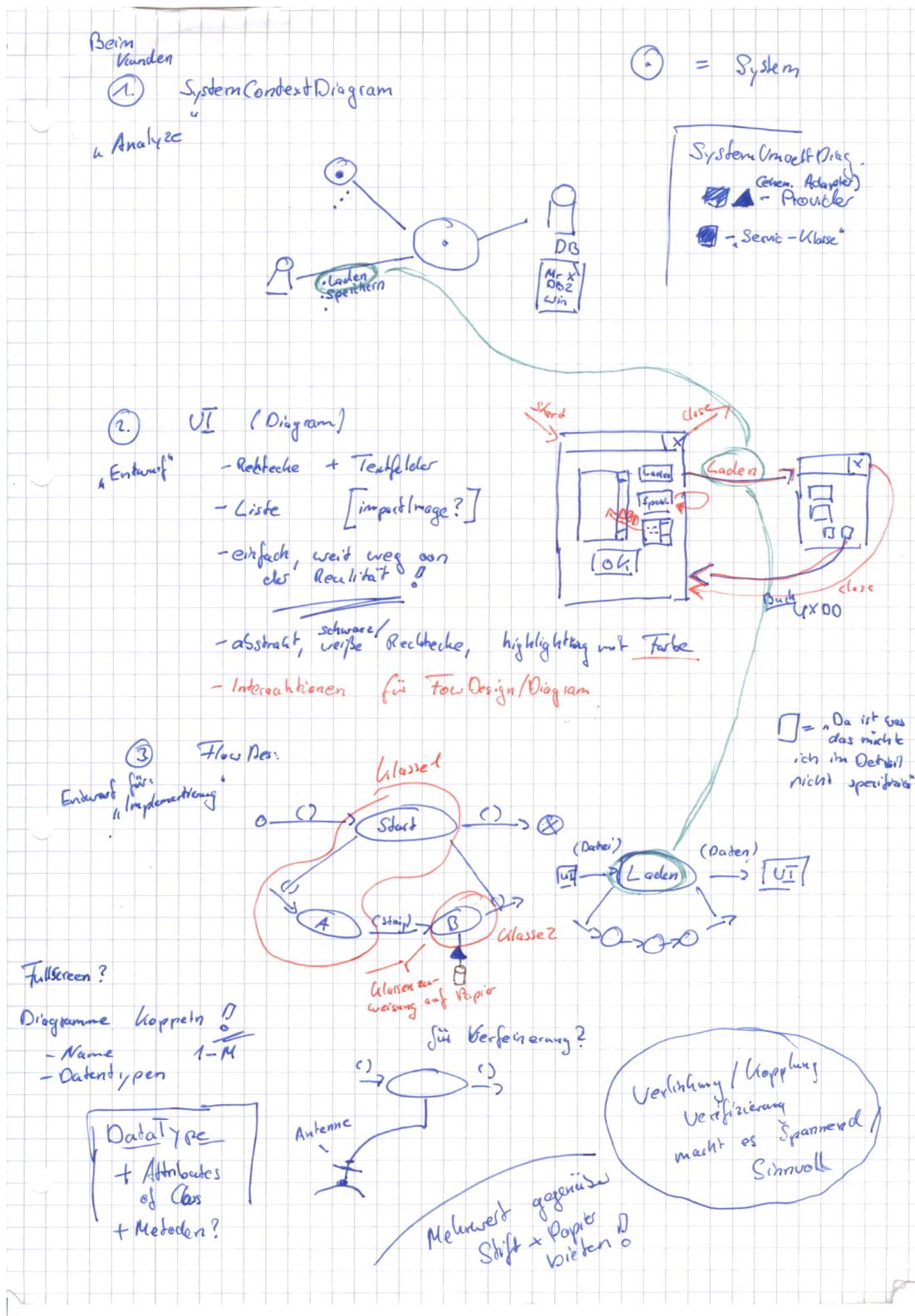
5.3 Abschlusspräsentation - 02.01.17 - 10.01.17

- Erstellen der Präsentation
- Ergänzen der Dokumentation

5.4 Repository Übergabe - 23.01.17

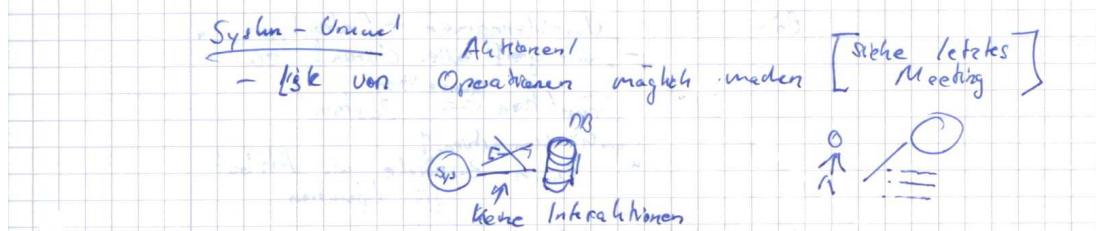
- Übergabe der Repository an den Kunden

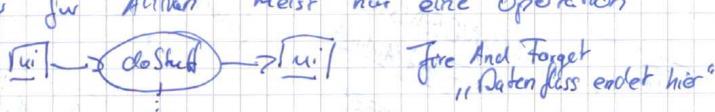
6 Protokoll zum Kundentreffen am 13. Dezember 2016



7 Protokoll zum Kundentreffen am 22. Dezember 2016

- Flow 2016-12-22
- mehrere Datentypen mit ; trennen
 - '*' hinter \$ gibt Störung → O...n / Liste / Stream
 - außen '*' → mehrere Datenflüsse / mehrmaliges „Abfeuern“
 - Generics testen
 - Single Level of Abstraction / SLA Hover Vorschau?
 - Vorschau von verbundenen Diagrammen? ↴
 - Mehrere Diagramme als Node? O-:O-O-O-O-
 - Reihe Klammern bei Ressourcen, keine Funktionsheit
- Tonne oder Provider
- Mehrwert:
 - Detail anschauen? Input/Output prüfen.
 - Diagramme aus Text erstellen



- Mask
- Aktionen zum Maskenwechsel (wilt Hover?)
 - Flow für Aktion meist nur eine Operation
- 
- 
- 
- Beschreibung: text ← beschreibt Student → Name: datename
• Kleinbuchstabe → Name: datename
• Großbuchstabe → Tip: String



8 Module

8.1 Modulübersicht

Die Teilbereiche des Projekts wurden in verschiedenen Modulen aufgeteilt.

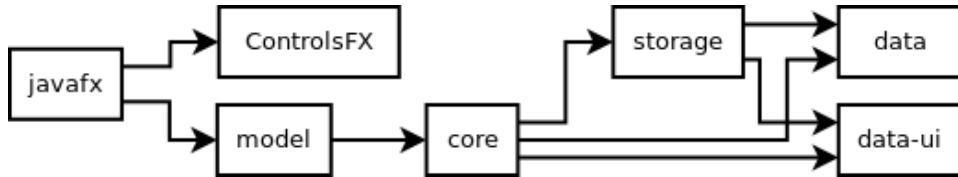
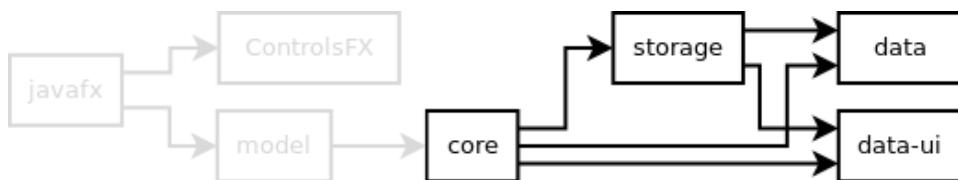


Abbildung 10: Abhängigkeitsgraph der Module

Der in Abbildung 10 gezeigte Graph enthält keine Abhängigkeiten nach JUnit und dem JDK aus Gründen der Übersicht. Zudem ist ControlsFX [1] kein eigen entwickeltes Modul und wird lediglich aus Gründen der Vollständigkeit aufgeführt.

Anzumerken ist auch, dass bis in das Modul Core (8.2) keine Abhängigkeiten gegenüber einer UI-Bibliothek existieren. Das Modul Data-UI stellt zwar Komponenten bereit, die für die Integration in eine Benutzeroberfläche benötigt werden, bleibt jedoch Benutzeroberflächenunabhängig.

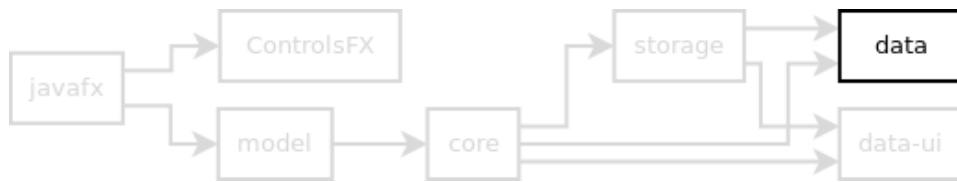
8.2 Modul Core



Abhängigkeiten: Modul Data (8.3), Modul Data-UI (8.4), Modul Storage (8.7)

Das Modul *core* (inklusive Abhängigkeiten) enthält alle Logik. Falls eine andere Benutzeroberfläche erstellt, oder die Logik anderweitig benötigt wird, sollte auf dieses Modul die Abhängigkeit aufgebaut werden.

8.3 Modul Data



Abhängigkeiten: Keine

Das Modul enthält die Haupt-Geschäftslogik und alle Datenmodelle. Da viel Logik nur darin besteht die referentielle Integrität bzw. Vollständigkeit der Daten zu gewährleisten ist dieses Modul im Gegensatz zum Modul Core (8.2) relativ schwergewichtig.

8.3.1 Bindings

Zur Kommunikation innerhalb dieses Moduls wird größtenteils das ObservableValue-Pattern verwendet. Hierbei wird auf die Java-eigene Klasse *PropertyChangeSupport* zurückgegriffen. Diese Klasse ermöglicht das Registrieren von Handlern bzw. das Auslösen von Events. Zudem unterstützt JavaFX diese Art von ObservableValue, wodurch es möglich ist, im Modul JavaFX (8.5) Properties von JavaFX mit Properties des Datenmodells zu verbinden. Beispielhaft sieht eine solche Klasse folgendermaßen aus.

```
public class Bean {  
    private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        String old = this.name;  
        this.name = name;  
        pcs.firePropertyChange("name", old, name);  
    }  
    public void addPropertyChangeListener(PropertyChangeListener listener) {  
        pcs.addPropertyChangeListener(listener);  
    }  
}
```

Abbildung 11: Beispiel Java Bean

8.3.2 Flow Notation Parser

Dieser Parser ist für die Umsetzung von textueller Flow-Notation in eine auswertbare Form zuständig. Hierbei können beispielsweise folgende Konstrukte verarbeitet werden:

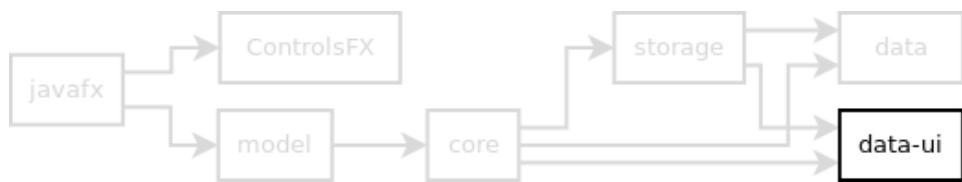
- {name:string}
- {int}
- {(name:string, amount:int)}
- (name:string)
- (float)*
- (float*, int)
- (float*, int)/(double*, int)

Vom Parser wird ein generisches *FlowAction* Objekt zurückgegeben, wenn der Vorgang erfolgreich war. Dieses Objekt kann eines der folgenden Klassen sein:

- MultiStream (**{test}**)
- Tupel (**(test)**)
- Type (**name:test**)
- Chain (**test/int**)

Objekte dieser Klassen enthalten entsprechend Möglichkeiten auf die Kind-Elemente (wenn vorhanden) oder Eigenschaften zuzugreifen.

8.4 Modul Data-UI

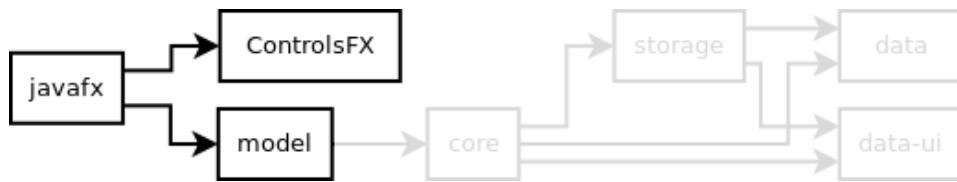


Abhängigkeiten: Keine

Im diesem Modul befinden sich Datenmodelle, die nicht direkt in die eigentliche Geschäftslogik eingebaut werden können / sollen, sondern eigentlich nur für die Oberfläche selbst benötigt werden.

Ein Beispiel hierfür ist das Changelog, welches beim Start der Anwendung geöffnet werden kann. Sollte in Zukunft beispielsweise die Möglichkeit eingebaut werden, Oberflächeneinstellungen zu speichern, so wäre dieses Modul der entsprechende Platz für die Datenmodelle.

8.5 Modul JavaFX



Abhängigkeiten: Modul Model (8.6), ControlsFX (ext. [1])

Dieses Modul enthält die Implementierung der Benutzeroberfläche. Hierbei enthält es Abhängigkeiten zu den Datenmodellen und weiterer Geschäftslogik. Neben der Benutzeroberfläche sind hier auch einige Ressourcen enthalten, beispielsweise Bilder, Stylesheets oder die Resource-Bundles, die die übersetzten Texte enthalten. Das Programm kann über die main-Methode in der Klasse *Main* gestartet werden.

8.5.1 Erscheinungsbild

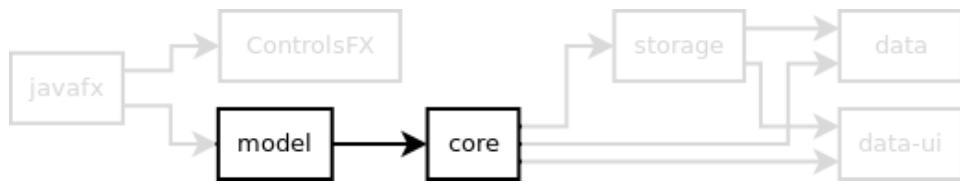
Um FlowDesign von anderen JavaFX-Applikationen zu differenzieren wurde das Aussehen der Oberflächenelemente komplett überarbeitet. Hierfür wurden alle verwendeten Elemente über entsprechende CSS-Regeln bearbeitet.

8.5.2 Übersetzung

Die komplette Anwendung ist sowohl in Deutsch als auch in Englisch verfügbar. Hierbei wird jeweils die im Betriebssystem ausgewählte Sprache verwendet. Sollte eine Sprache eingestellt sein, die nicht von der Anwendung unterstützt wird, so wird die Voreinstellung Englisch verwendet.

Um die Übersetzung zu realisieren wurden Java Resource-Bundles verwendet. Diese bestehen im wesentlichen aus einfachen Textdateien mit Key-Value Paaren (durch ein Gleichzeichen getrennt).

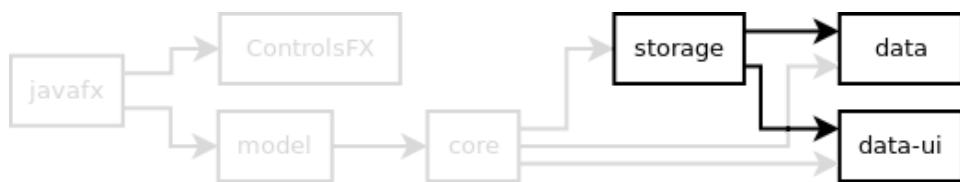
8.6 Modul Model



Abhängigkeiten: Modul Core ([8.2](#))

Dieses Modul wurde erst spät im Entwicklungsprozess mit einbezogen und beinhaltet bisher noch keine Klassen. Es ist allerdings dafür vorgesehen, Klassen zu enthalten, die für die Oberfläche gebraucht werden, allerdings unabhängig vom verwendeten UI-Toolkit sind.

8.7 Modul Storage



Abhängigkeiten: Modul Data ([8.3](#)), Modul Data-UI ([8.4](#))

Dieses Modul enthält sowohl eine abstrakte Definition von *Serializern* und *Storage-Handlern* als auch eine Implementation für XML inklusive *Serializer* für Komponenten im Modul Data ([8.3](#)) und Modul Data-UI ([8.4](#)).

8.7.1 StorageHandler

Die Klasse *StorageHandler* hält *Storages* und verteilt Serialisierungsaufgaben an das entsprechende *Storage* anhand eines Textidentifikators (für XML lautet dieser 'xml'). Eine weitere *Storage* Implementation kann anhand eines neuen Textidentifikators registriert werden, wodurch ein Umstieg von XML auf JSON, SQL oder eine andere Implementation vereinfacht wird.

8.7.2 Storage

Ein *Storage* stellt ein Speicherort für alle zum Projekt gehörenden Komponenten dar. Je nach Implementation kann dies XML (implementiert), JSON, SQL oder andere sein (nicht implementiert). Bei einem *Storage* können *Serializer* für weitere Komponenten registriert werden. Sowohl Lese- als auch Schreibhandles und Hilfsklassen für die *Serializer* werden über Generics in *Storage* definiert.

8.7.3 Serializer

Ein *Serializer* hat die Aufgabe eine Komponente zu serialisieren und wieder zu deserialisieren. Der Umfang eines *Serializers* sollte sich auf eine Komponente beziehen (siehe Single-Responsibility-Prinzip [9.1.1](#)).

9 Einhaltung Programmierkonzepte

Programmierprinzipien sind essentiell für die Wartbarkeit, Korrektheit und vor allem die Verständlichkeit eines Programmcodes. Die wachsende Komplexität von Programmen und der damit steigende Entwicklungsaufwand erfordern sauberes und strukturiertes arbeiten mehr den je.

9.1 SOLID

Bei SOLID handelt es sich um grundlegende Prinzipien der objektorientierten Programmierung, welche von Robert C. Martin in den frühen 2000ern formuliert wurden. Alle dieser Prinzipien dienen zur Verbesserung der Wartbarkeit und Erweiterbarkeit von Computerprogrammen.

9.1.1 Single-Responsibility-Prinzip

Das Single-Responsibility-Prinzip besagt, dass eine Klasse, Methode oder Funktion nur einer Aufgabe verschrieben sein soll. So soll die Funktion zu Berechnung einer Potenz nur dies tun und nicht gleichzeitig das Ergebnis zbsp auf einer GUI ausgeben. Dadurch entwickelt sich ein Art Baukastenprinzip bei dem die unterschiedlichen Funktionen einfach miteinander verknüpft und / oder ausgetauscht werden können, da keine unnötigen Abhängigkeiten aufgebaut werden. "Gottklassen", die viele verschiedene Funktionalitäten vereinen, sind nicht erlaubt.

9.1.2 Open-Closed-Prinzip

Das Open-Closed-Prinzip besagt, dass eine Klasse, Methode oder Funktion offen für Erweiterungen aber geschlossen für Veränderungen sein soll. So soll es möglich sein die Klasse Drucker um ein weiteres Papierformat zu erweitern, jedoch darf sich das Verhalten von verschiedenen von Drucker abgeleiteten Klassen nicht unterscheiden.

9.1.3 Liskovsches Substitutionsprinzip

Das Liskovsche Substitutionsprinzip besagt, dass ein Computerprogramm weiter funktionsfähig bleiben muss, auch wenn Objekte vom Typ T durch Objekte vom Typ S ersetzt werden, wobei S eine Unterklasse von T darstellt (vgl. [6], Z. 5). Oder kurz gesagt „Subclasses should be substitutable for their base classes“ ([4], S. 11).

Bei einer solchen Vorgehensweise ist es außerordentlich wichtig, dass das Prinzip **Design by Contract** eingehalten wird (vgl. [2], S. 5), damit das Computerprogramm

weiterhin richtig funktioniert. Nach außen hin müssen die Operationen beider Klassen in allen Fällen gleich funktionieren.

Die logische Konsequenz dieses Prinzips ist eine verbesserte Erweiterbarkeit und Korrektheit (vgl. [3], S. 2), da Funktionalität relativ einfach angepasst werden kann, ohne bestehenden Code zu verändern. Dadurch wird auch das Open-Closed-Prinzip (9.1.2) impliziert.

9.1.4 Interface-Segregation-Prinzip

Das Interface Segregation Principle verlangt, dass Schnittstellen jeweils auf einen Aufgabenbereich zugeschnitten sind. Anstatt einer großen Schnittstelle soll für jeden Aufgabenbereich eine eigene Schnittstelle definiert werden, die keine Abhängigkeiten zu anderen Teilen des Programms aufbaut, die für die eine Aufgabe nicht benötigt werden. Im Grunde genommen entspricht dies dem Single-Responsibility-Prinzip (9.1.1) für Schnittstellen.

Anderen Programmteilen wird dadurch ermöglicht mit Schnittstellen arbeiten zu können, ohne andere nicht benötigte Abhängigkeiten zu bilden: "Clients [oder andere Programmteile] sollten nicht dazu gezwungen werden, von Interfaces abzuhängen, die sie gar nicht brauchen" (vgl. [2] S. 10).

9.1.5 Dependency-Inversion-Prinzip

Das Dependency Inversion Principle beschreibt das bei hierarchischer Verteilung von Modulen, Module niedrigerer Ebenen von Modulen höherer Ebenen abhängen. Module höherer Ebenen stellen somit Anforderungen durch Interfaces, die Module niedrigerer Ebenen zu implementieren haben. Meist sind diese Interfaces dabei in eigene Module aufgeteilt. Die Abhängigkeit ist somit umgekehrt wie traditionell zuerst zu vermuten sei.

Mit "höheren Modulen" sind dabei Module gemeint, die Funktionalität meist weiter abstrahiert haben und auf Funktionalität von "niederen" - und sehr spezifischen - Modulen aufbauen.

10 Aufbau Diagramm-Element

10.1 Properties

Ein Diagramm-Element hat mindestens sechs Properties:

- X-Koordinate
- Y-Koordinate
- Breite
- Höhe
- Text
- Farbe

10.2 Joint

Ein Joint ist ein Verbindungsknoten. Je nach Konfiguration kann ein Joint als Eingang, Ausgang oder beides dienen. Bisher gibt es sowohl einen *FlowJoint* als auch einen *DependencyJoint*.

10.3 Joint-Group

Eine Joint-Group ist die implementierte Art und Weise wie gleichartige Joints gruppiert werden. Eine solche Gruppe besitzt Maximal- bzw. Minimalzahlen von Joints, wobei neue Joints durch eine Factory erstellt werden. Alle diese Informationen werden beim Konstruktorauftruf übergeben. Elemente wie die Operation haben beispielsweise vier Joint-Groups. Zwei für Flow- bzw. Abhängigkeits-Eingang und zwei für Flow- bzw. Abhängigkeits-Ausgang.

Solange Maximal- bzw. Minmalanzahl der Joints eingehalten werden, können dynamisch Joints hinzugefügt bzw. entfernt werden (von der Oberfläche allerdings noch nicht unterstützt).

11 Erweiterung

Beim Architekturentwurf wurde darauf geachtet das Programm so einfach wie möglich erweiterbar zu machen. Dieses Kapitel zeigt wie der vorhandene Code erweitert und um Inhalt ergänzt werden kann.

11.1 Erweiterung um einen Diagrammtyp

Im Folgenden soll ein neuer Diagramm-Typ mit dem Namen "Example" beispielhaft erstellt werden.

11.1.1 Datenmodell

Für alle bisherigen Diagramme wurde ein eigenes Paket erstellt. Das ist keine Voraussetzung, hilft allerdings bei der Strukturierung. Im Folgenden wird deshalb davon ausgegangen, dass das Paket *com.tallbyte.flowdesign.data.example* verwendet wird. Hier muss eine Klasse "ExampleDiagram" erstellt werden, welche von *Diagram* erbt. Sollte es erwünscht sein, dass Elemente des Diagramms von einer bestimmten Art sind, so kann dies über den generischen Parameter von *Diagram* angegeben werden.

```
public class ExampleDiagram extends Diagram<DiagramElement> {  
  
    public ExampleDiagram(String name) {  
        super(name, null);  
    }  
  
    public EnvironmentDiagram(String name, DiagramElement root) {  
        super(name, root);  
    }  
}
```

Abbildung 12: Beispiel Diagramm-Klasse

11.1.2 View

Das Diagramm hat keine eigentliche View. Diese Aufgabe wird direkt von der Klasse *DiagramPane* im Modul JavaFX (8.5) umgesetzt. Hier ist Diagramm-spezifisch nichts zu verändern.

11.1.3 Handler

Der *DiagramHandler* ist die Diagramm-spezifische Schnittstelle, die Aufgaben wie die Erstellung von neuen Diagramm-Instanzen oder die Bereitstellung verfügbarer Properties übernimmt. Zur Vereinfachung gibt es bereits eine abstrakte Klasse, die die komplexesten Aspekte des Interfaces *DiagramHandler* verbirgt. Diese kann erweitert werden womit folgende Klasse im Kontext des Beispiels zu erstellen ist.

```
public class ExampleDiagramHandler extends
    DiagramHandlerBase<ExampleDiagram, DiagramElement, DiagramImage> {

    public ExampleDiagram() {
        addEntries("System", System.class,
                  System::new,
                  EllipseDiagramImage::new,
                  SystemElementNode::new
        );
    }

    @Override
    protected ExampleDiagram createNewDiagramInstance(String name) {
        return new ExampleDiagram(name);
    }

    @Override
    public ObservableList<Property<?>> getDiagramProperties(ExampleDiagram
        diagram) {
        ObservableList<Property<?>> list =
            super.getDiagramProperties(diagram);

        // add properties

        return list;
    }
}
```

Abbildung 13: Beispiel Diagramm-Handler-Klasse

Der Methodenaufruf von *addEntries* wird später noch im Kapitel zum Hinzufügen

neuer Diagramm-Element erklärt (11.2). Ansonsten kann in der Methode *getDiagramProperties* eine Liste mit verfügbaren Properties für ein Diagramm erstellt werden. Diese Liste wird verwendet, um auf der Oberfläche Optionen für das jeweilige Diagramm einzublenden.

Damit der Handler auch aktiv genutzt wird, muss noch ein Methodenaufruf in der Klasse *DiagramHandler* gemacht werden.

```
static {
    addHandler(EnvironmentDiagram.class, new EnvironmentDiagramHandler());
    addHandler(FlowDiagram.class, new FlowDiagramHandler());
    addHandler(MaskDiagram.class, new MaskDiagramHandler());

    // diese Zeile muss eingefügt werden
    addHandler(ExampleDiagram.class, new ExampleDiagramHandler());
}
```

Abbildung 14: Diagramm-Handler verfügbar machen

11.1.4 Serialisierung

Damit der neu erstellte Diagramm-Typ nun auch gespeichert werden kann muss ein (XML-)Serializer erstellt werden. Bisher wurde für jeden Diagram-Typ ein eigenes Paket angelegt, in dem sich sowohl der *Serializer* für das *Diagram* als auch für dessen *Elemente* befinden. Das ist allerdings keine Voraussetzung. Für dieses Beispiel wird das Paket *com.tallbyte.flowdesign.storage.xml.example* angelegt. Die neue Klasse *XmlExampleDiagramSerializer* erbt von *XmlDiagramSerializer* in der schon viele rudimentäre Methoden zum de-/serialisieren von Diagrammen und deren Attribute existieren.

```
public class XmlExampleDiagramSerializer extends XmlDiagramSerializer
    implements XmlSerializer<ExampleDiagram> {

    @Override
    public void serialize(XMLStreamWriter writer, ExampleDiagram diagram,
        XmlSerializationHelper helper) throws IOException {
        // see other figure
    }

    @Override
    public ExampleDiagram instantiate() {
        // not happening here for diagrams
        return null;
    }

    @Override
    public ExampleDiagram deserialize(XMLStreamReader reader, ExampleDiagram
        serializable, XmlDeserializationHelper helper) throws IOException {
        // see other figure
    }
}
```

Abbildung 15: Beispiel XML-ExampleDiagram-Serializer-Klasse

Verwirrend mag vermutlich die *instantiate()* Implementation aussehen. Je nach zu serialisierender Klasse muss evtl. eine gültige und damit referenzierbare Instanz vor dem Laden dessen Inhalts verfügbar sein (zbsp. bei komplexeren Verknüpfungen). Falls dies der Fall sein kann (wie bei den *Elementen*), muss hier eine gültige neue Instanz zurückgegeben werden. Bei den Diagrammen ist dies jedoch nicht der Fall, somit kann hier *null* zurückgegeben werden. Das bedeutet aber auch, dass der Parameter *serializable* in *deserialize(...)* den Wert *null* hat.

```
@Override
public void serialize(XMLStreamWriter writer, ExampleDiagram diagram,
    XmlSerializationHelper helper) throws IOException {
    try {
        // write diagram related attributes
        serializeAttributes(writer, diagram, helper);
        // serialize more diagram specific attributes here...

        // write elements
        helper.getAssignedIdMap().clear();
        serializeElements(writer, diagram.getElements(), helper);

        // write connections (remove if no connnections available)
        serializeConnections(writer, diagram.getConnections(), helper);

    } catch (XMLStreamException e) {
        throw new IOException(e);
    }
}
```

Abbildung 16: Generische Implementation für serialize

Die generischen Methoden von *XmlDiagramSerializer* sollten die meiste Arbeit bereits erledigen können. Weitere Anpassungen sind jedoch einfach möglich.

Da in *instantiate()* darauf verzichtet wurde eine neue Instanz zu erstellen, muss dies nun in *deserialize(...)* erledigt werden.

```
@Override
public ExampleDiagram deserialize(XMLStreamReader reader, ExampleDiagram
    serializable, XmlDeserializationHelper helper) throws IOException {
    try {
        // load the diagram attributes
        Map<String, String> attributes = helper.getAttributes(reader);

        helper.getAssignedIdMap().clear();
        Queue<Map.Entry<String, MaskDiagramElement>> queue =
            deserializeElementTypes(
                reader,
                MaskDiagramElement.class,
                helper
            );

        // process more diagram specific attributes somewhere around here
        ExampleDiagram diagram = new ExampleDiagram(
            attributes.get(ATTRIBUTE_NAME)
        );

        // fill all the elements with proper values
        deserializeElements(
            reader, queue, MaskDiagramElement.class, helper
        );

        queue.stream()
            .map(Map.Entry::getValue)
            .forEach(diagram::addElement);

        // build all the connections
        deserializeConnections(reader, Connection.class, helper);
        return diagram;
    } catch (XMLStreamException e) {
        throw new IOException(e);
    }
}
```

Abbildung 17: Generische Implementation für deserialize

Der neue *Serializer* muss nun in dem entsprechenden Storage (in diesem Fall im *XmlStorage*) registriert werden. Sowohl der *Serializer* für das *Diagram* als auch für dessen *Elemente* werden bisher in Blöcken zusammengefasst, wie für *MaskDiagram* beispielhaft zu sehen ist.

```
// ...  
  
// MaskDiagram and elements  
register(MaskDiagram .class, new XmlMaskDiagramSerializer());  
register(MaskComment .class, new XmlMaskCommentSerializer());  
register(Rectangle .class, new XmlRectangleSerializer());  
register(SelfReference .class, new XmlSelfReferenceSerializer());  
  
// ExampleDiagram and elements  
register(ExampleDiagram.class, new XmlExampleDiagramSerializer());  
// further element-serializer-registrations shall be placed here
```

Abbildung 18: Registrieren des neuen Diagramm-Serializers

11.1.5 Strings

Da der Projektbaum und andere UI-Elemente automatisch für alle verfügbaren Diagrammtypen erstellt werden, müssen die angezeigten Texte extern verwaltet werden. Hierfür wird das Java-eigene Ressourcensystem verwendet. Im Ressourcen-Verzeichnis in der Maven-Struktur befindet sich das Resource-Bundle *MessagesBundle*. Hier müssen folgende Strings bereitgestellt werden:

- tree.overview.ExampleDiagram = Example
- menu.edit.new.ExampleDiagram = New Example-Diagram...
- popup.new.ExampleDiagram.title = New Example Diagram
- popup.new.ExampleDiagram.field.name = Name
- context.new.ExampleDiagram=New...

11.2 Erweiterung um ein Diagramm-Element

Im Folgenden soll ein neues Diagramm-Element mit dem Namen "ExampleElement" beispielhaft erstellt werden.

11.2.1 Datenmodell

Zuerst muss das entsprechende Datenmodell erstellt werden. Nach bisheriger Konvention gehören die Diagramm-Elemente für ein spezielles Diagramm in das gleiche Paket wie die entsprechende Diagramm-Klasse. Das ist allerdings keine Voraussetzung.

```
public class ExampleElement extends DiagramElement {

    public static final String JOINT_GROUP = "io";

    public ExampleDiagramElement() {
    }

    @Override
    protected Iterable<JointGroup<?>> createJointGroups() {
        return new ArrayList<JointGroup<?>>() {{
            add(new JointGroup<>(System.this, JOINT_GROUP, 4, 4, element ->
                new DependencyJoint(element, JointType.INPUT_OUTPUT, 0, 0),
                4));
        }};
    }

    public JointGroup<?> getJointGroup() {
        return getJointGroup(JOINT_GROUP);
    }
}
```

Abbildung 19: Beispiel Diagramm-Element-Klasse

Hier wird ein einfaches Element erstellt, dass eine einzige Joint-Gruppe bereitstellt, die minimal und maximal vier Joints zur Verfügung stellt, wobei einzelne Joints sowohl als Eingang als auch als Ausgang verwendet werden können. Zudem ist die Art der

Verbindung, welche von den Joints erstellt werden kann eine Abhängigkeitsverbindung (ein Kreis am Ziel).

11.2.2 View

Damit das neue Element später auch auf der Zeichenfläche angezeigt werden kann, muss erst eine View dafür erstellt werden. Alle Views müssen die Basis-Klasse *ElementNode* erweitern. Diese stellt bereits einige wichtige Funktionen wie das Verteilen von Joints auf der Oberfläche oder das Vergrößern / Verkleinern zur Verfügung.

```
public class ExampleElementNode extends ElementNode<DiagramImage> {

    private final ExampleElement example;

    public SystemElementNode(ExampleElement element, DiagramImage content) {
        super(element, content, Pos.CENTER);

        this.example = element;
    }

    @Override
    protected void setup() {
        super.setup();

        addJointsAcrossCircle(new JointGroupHandler(example.getJointGroup(),
            0, 1));
    }
}
```

Abbildung 20: Beispiel Diagramm-Element-Node

Durch den Methodenaufruf *addJointsAccrossCircle* werden alle Joints der Joint-Group des übergebenen ExampleElements komplett über den Umfang einer Ellipse verteilt, der sich durch Höhe und Breite des Elements ergibt. Die *0* steht hier für den Offset im Kreis (Intervall [0;1]), die *1* steht für den verwendeten Umfang (Intervall [0;1]) in Prozent.

Es stehen hier folgende Funktionen zur Verfügung:

- *addJointsAccrossRectangle*

- addJointsAcrossRectangleCentered
- addJointsAcrossCircle
- addJointsAcrossCircleCentered

Der dritte Parameter beim Super-Konstruktorauftruf steht für die Position des Textfelds für den Namen. Gültige Werte sind *Pos.CENTER* und *Pos.BOTTOM_CENTER*.

11.2.3 Image

Das *DiagramImage* stellt den Teil des Views dar, der für das eigentliche Erscheinungsbild verantwortlich ist, also z.B. ein Kreis bzw. Oval bei der Operation. Beim Erweitern des *DiagramImages* muss lediglich die *repaint()* Methode überschrieben werden. Zusätzlich ist es allerdings auch möglich eigene Properties zu definieren, um das Verhalten des Images zu steuern ([11.3](#)).

```
public class ExampleDiagramImage extends DiagramImage {  
  
    public EllipseDiagramImage() {  
  
    }  
  
    @Override  
    public void repaint() {  
        GraphicsContext context = getGraphicsContext2D();  
        double width = getWidth();  
        double height = getHeight();  
  
        context.clearRect(0, 0, width, height);  
        context.setStroke(getColor());  
        context.setLineWidth(1.5);  
        context.strokeOval(  
            context.getLineWidth(), context.getLineWidth(),  
            width - 2*context.getLineWidth(), height -  
            2*context.getLineWidth())  
    };  
}  
}
```

Abbildung 21: Beispiel Diagram-Image

11.2.4 Factory

Um das neue Element nun endgültig in der Oberfläche verfügbar zu machen muss schlussendlich noch eine Factory erstellt werden. Dies ist im entsprechenden *DiagramHandler* für den jeweiligen Diagramm-Typ zu machen. Für Flow-Diagramme also im *FlowDiagramHandler*. In diesem Fall ist es der *ExampleDiagramHandler*. Hier muss im Konstruktor folgende Zeile eingefügt werden.

```
addEntries(  
    "ExampleElement",  
    ExampleElement.class,  
    ExampleElement::new,  
    ExampleDiagramImage::new,  
    ExampleElementNode::new  
) ;
```

Abbildung 22: Beispiel Element Factory

Um unnötige Klassen bzw. riesige Code-Konstrukte zu vermeiden werden hier Lambda-Ausdrücke eingesetzt.

11.2.5 Serialisierung

Für die Serialisierung muss im Prinzip das gleiche wie bereits in der Serialisierung für das neue Diagramm (11.1.4) umgesetzt werden, dh. einen neuen Serializer erstellen und im *XmlStorage* registrieren. Auch hier steht eine nützliche Basisklasse bereit: *XmlElementSerializer*.

```
public class XmlExampleElementSerializer extends
XmlElementSerializer<ExampleElement> {

    @Override
    public ExampleElement instantiate() {
        // new empty instance of ExampleElement making it referencable
        return new ExampleElement();
    }

    @Override
    protected Map<String, String> saveAttributes(
        Map<String, String> attributes,
        ExampleElement element,
        XmlSerializationHelper helper) throws XMLStreamException {

        // add element specific attributes to the map before returning
        return super.saveAttributes(attributes, element, helper);
    }

    @Override
    protected void loadAttributes(
        Map<String, String> attributes,
        ExampleElement element,
        XmlDeserializationHelper helper) throws XMLStreamException {

        super.loadAttributes(attributes, element, helper);
        // load more element specific attributes from the map here
    }
}
```

Abbildung 23: Beispiel Serializer für ein Element

Nach Implementation das Registrieren nicht vergessen:

```
// ...  
  
// MaskDiagram and elements  
register(MaskDiagram .class, new XmlMaskDiagramSerializer());  
register(MaskComment .class, new XmlMaskCommentSerializer());  
register(Rectangle .class, new XmlRectangleSerializer());  
register(SelfReference .class, new XmlSelfReferenceSerializer());  
  
// ExampleDiagram and elements  
register(ExampleDiagram.class, new XmlExampleDiagramSerializer());  
register(ExampleElement.class, new XmlExampleElementSerializer());  
// further element-serializer-registrations shall be placed here
```

Abbildung 24: Registrieren des neuen Element-Serializers

11.3 Weitere Referenzen

Auf Grund der Offenheit des Systems sind eine Vielzahl von Modifikationen möglich, die nicht alle in dieser Dokumentation behandelt werden können. Allerdings sind bereits einige Beispiele im Programmcode vorhanden. Die folgende Liste zeigt einige Referenzen für *erweiterte* Funktionen:

- Extra Textfeld im View (*OperationalUnitElementNode#setup()*)
- Property-Verbindung mit dem Image (*EndElementNode#EndElementNode()*)

12 Benutzerhandbuch

12.1 Projektauswahlfenster

Das Programm startet mit einem Auswahlfenster für Projekte. Hier haben Sie die Möglichkeit zuletzt erstellte Projekte zu öffnen, andere existierende Projekte hinzuzufügen oder ein neues Projekt zu erstellen.

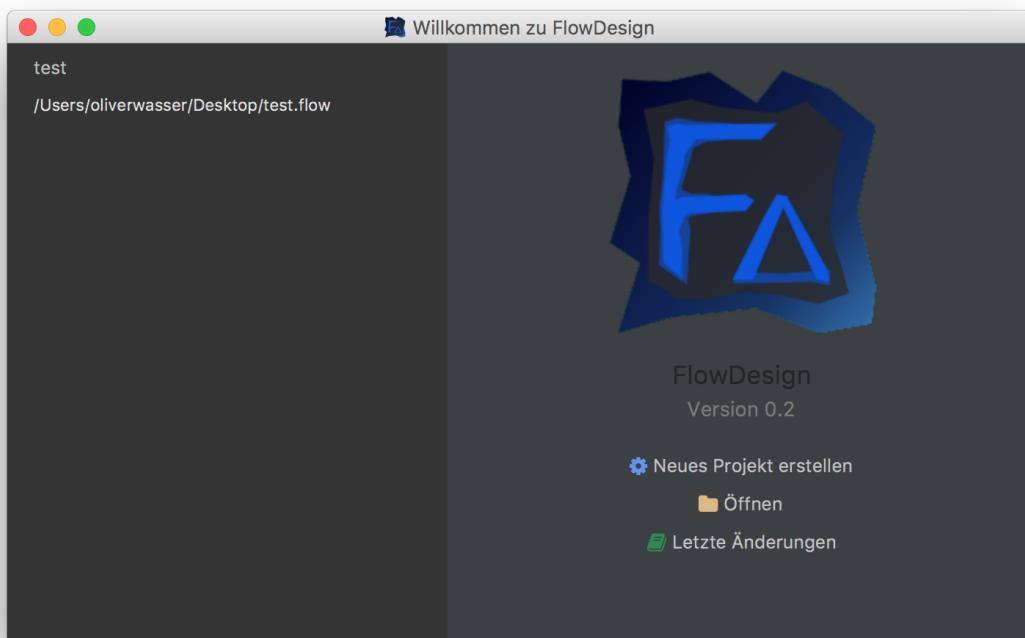


Abbildung 25: Auswahlfenster

12.1.1 Öffnen eines kürzlich erstellten Projekts

Zum Öffnen eines kürzlich erstellten Projekts, wählen Sie mit einem Doppelklick das gewünschte Projekt im linken Teil des Auswahlfensters.

12.1.2 Öffnen eines beliebigen Projekts

Zum Hinzufügen eines anderen bestehenden Projektes, wählen Sie mit einem Linksklick "Öffnen". Es erscheint ein Fenster zur Auswahl des Dateipfades. Wählen Sie nun das gewünschte Projekt als ".flow" Datei aus und bestätigen Sie anschließend.

12.1.3 Erstellen eines neuen Projekts

Zum Erstellen eines neuen Projektes, drücken Sie "Neues Projekt erstellen". Im folgenden Fenster tragen Sie einen Name und Speicherort für Ihr Projekt ein. Bestätigen Sie mit "Ok".

12.2 Projektfenster

12.2.1 Menüleiste



Abbildung 26: Menüleiste unter macOS

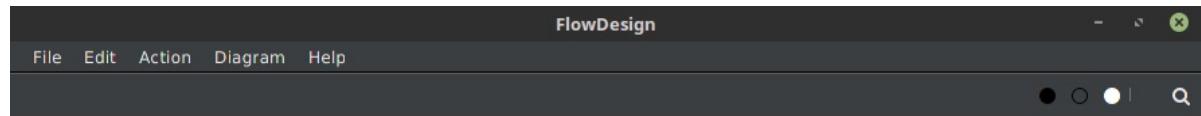


Abbildung 27: Menüleiste unter Linux/Windows

Die Menüleiste enthält die Auswahlpunkte "Datei", "Bearbeiten", "Aktion", "Diagramm" und "Hilfe". Die Sprache der Leiste wechselt automatisch zwischen Deutsch und Englisch, je nachdem unter welcher Sprache Ihr Betriebssystem eingestellt ist. Die für die einzelnen Aktionen nötigen Shortcuts werden Ihnen jeweils zugehörig in der Menüleiste angezeigt.

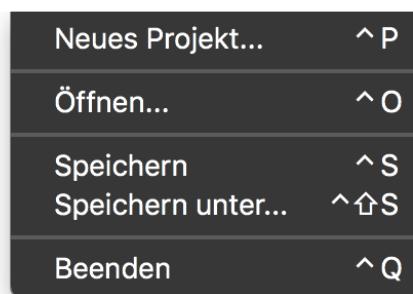


Abbildung 28: Menüleiste - "Datei"

Unter "Datei" ist es Ihnen möglich ein anderes Projekt zu öffnen, das aktuelle Projekt

zu speichern oder mit "Speichern unter" eine neue Kopie unter einem beliebigen Pfad abzulegen.

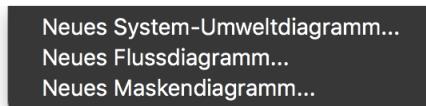


Abbildung 29: Menüleiste - "Bearbeiten"

Unter "Bearbeiten" haben Sie die Möglichkeit neue Diagramme jedes Typen zu erstellen.



Abbildung 30: Menüleiste - "Aktion"

"Aktion" enthält die Suche nach Diagramme. Diese kann ebenfalls mit dem Lupensymbol in der oberen rechten Ecke des Programmes abgerufen werden.

Selektion löschen	^☒
Vorschläge öffnen	^ Space
Zur Referenz springen	^ B
Selektion verändern	^↑→
Selektion verändern	^↑←
Aktion ausführen	^↖↔
Flow-Element hinzufügen	^ D
Abhängigkeit hinzufügen	^ W
Selektion verändern	^↑↓
Selektion verändern	^↑↑

Abbildung 31: Menüleiste - "Diagram"

Unter "Diagramm" finden Sie sämtliche Optionen die das intelligente Arbeiten mit Diagrammen betrifft. Dazu gehört der Quick-Jump in verlinkte Diagramme oder das Hinzufügen von Abhängigkeiten.

Über FlowDesign ^ A

Abbildung 32: Menüleiste - "Hilfe"

Wählen Sie "Hilfe", um Informationen über den aktuellen Programmbuild zu erhalten.

12.2.2 Projektbaum und Anlegen neuer Diagramme

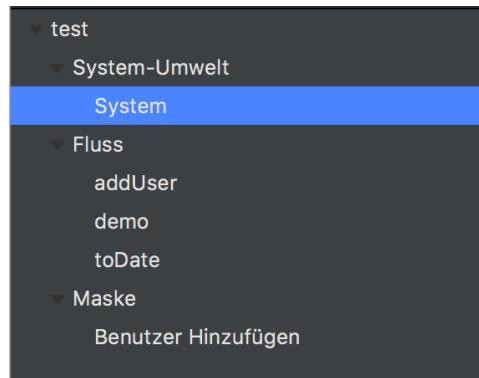


Abbildung 33: Projektbaum

Der Projektbaum befindet sich im Programm auf der linken Seite. In der obersten Zeile finden Sie Ihren zuvor gewählten Projektnamen wieder, gefolgt von den drei Diagrammtypen 'System-Umwelt', 'Fluss' und 'Maske'. Sie können beliebig viele Diagramme eines Typs erstellen.

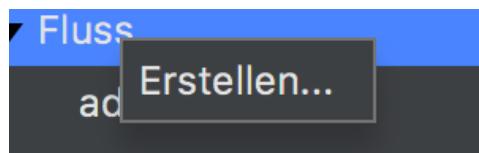


Abbildung 34: Projektbaum - Erstellen

Um ein neues Diagramm zu erstellen, drücken Sie mit der rechten Maustaste auf den gewünschten Diagrammtyp. Wählen Sie nun 'Erstellen' und vergeben Sie einen Namen, beachten Sie dabei das ein Name nur einmalig vergeben werden kann.

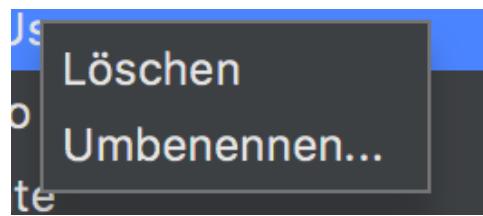


Abbildung 35: Projektbaum - Bearbeiten

Um bereits erstellte Diagramme zu löschen oder umzubenennen, drücken Sie mit der rechten Maustaste auf das gewünschte Diagramm und wählen Sie die Änderung welche Sie vornehmen möchten. Wenn Sie ein Diagramm umbenennen, wird die Namensänderung durch das Programm automatisch in allen anderen Diagrammen und Referenzen übernommen.

12.2.3 Datentypen bearbeiten

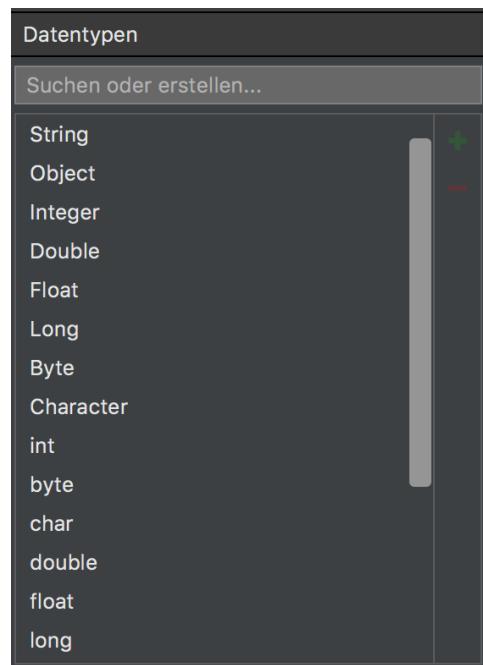


Abbildung 36: Datentypenliste

Sie können im linken Teil des Programmfensters eigene und bestehende Datentypen anlegen oder löschen. Diese werden alle in den einzelnen Diagrammen referenziert und können dort, etwa in der Auto vervollständigung, genutzt werden.

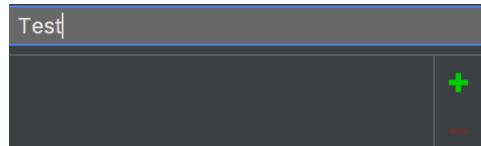


Abbildung 37: Datentypenliste - Hinzufügen

Um einen neuen Datentyp hinzuzufügen, geben Sie den gewünschten Namen im Textfeld ein. Existiert bereits ein Datentyp mit dem Namen, wird es Ihnen in der Liste angezeigt. Zum Hinzufügen betätigen Sie nun das grüne Plus-Symbol. Der neue Datentyp erscheint nun in der Liste.

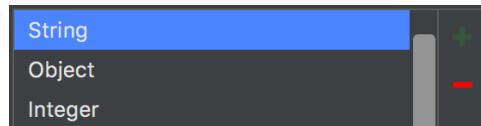


Abbildung 38: Zeichenfläche - Entfernen

Um einen Datentypen zu entfernen, markieren Sie diesen in der Liste mit einem Linksklick. Anschließend betätigen Sie das rote Minus-Symbol. Der Datentyp verschwindet nun aus der Liste. Datentypen, welche bereits in Diagrammen verwendet wurden, werden aus diesen nicht entfernt.

12.3 Zeichenfläche

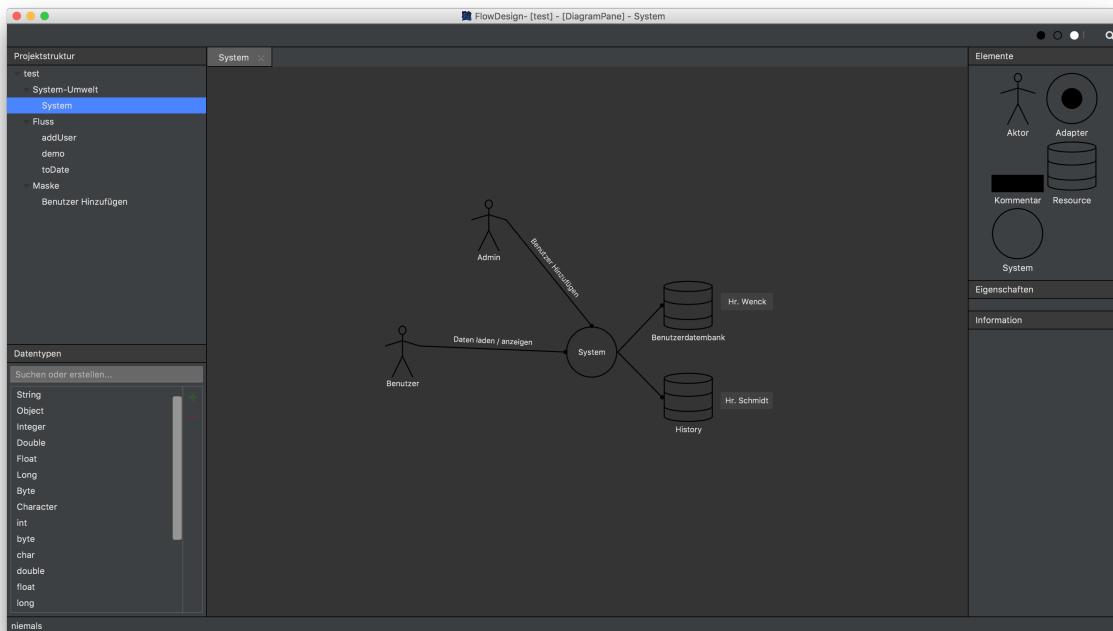


Abbildung 39: Zeichenfläche

Bei der Zeichenfläche handelt es sich um das Oberflächenelement zum Bearbeiten von Diagrammen.



Abbildung 40: Zeichenfläche - Tabs

Im oberen Randbereich befinden sich Tabs von geöffneten Diagrammen. Um zwischen Tabs zu wechseln, wählen Sie mit einem Linksklick oben den gewünschten Tab aus.

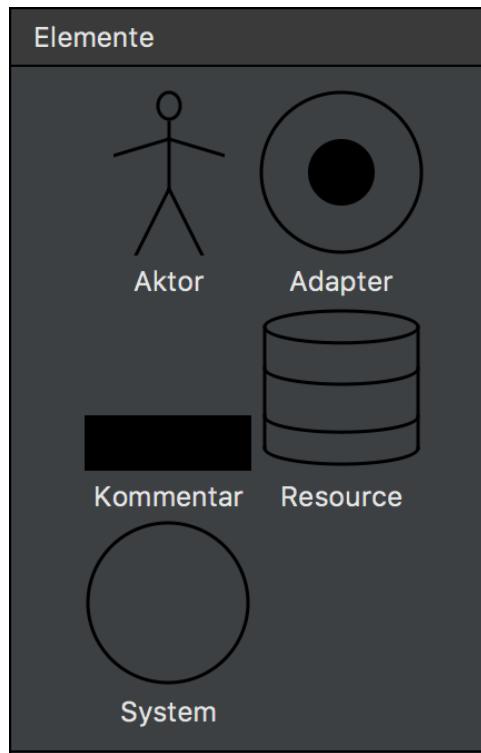


Abbildung 41: Zeichenfläche - Elemente

Rechts befindet sich eine Übersicht aller Elemente. Diese ändern sich je nach dem, in welchem Diagrammtyp man sich momentan befindet. Um ein Element hinzuzufügen, ziehen Sie dieses mit gedrückter Maustaste in das Zeichenfeld.



Abbildung 42: Zeichenfläche - Eigenschaften

Unter der Element-Auswahl befinden sich die Eigenschaften eines Elements. Um die Eigenschaften eines Elements zu verändern, markieren Sie es mit einem Linksklick. Je nach dem welches Element gewählt wurde, passen sich die dargestellten Eigenschaften automatisch an. Bei "Zustand" und "Typ" handelt es sich um spezielle Eigenschaften des Elements "Operation". Sollte eine Operation auf einen globalen Zustand des zu modellierenden Systems zugreifen, kann durch "Zustand" dessen Name angegeben werden. Mit "Typ" kann zwischen einem Ressourcenzugriff und einem Zustandszugriff unterschieden werden. Dabei ändert sich auch das angezeigte Icon der Operation. Eine weitere Besonderheit ist die Eigenschaft "Portal" im Flussdiagramm. Damit ist es möglich den Start/Ende durch Portale zu ersetzen.

12.3.1 Erstellen von Verbindungen

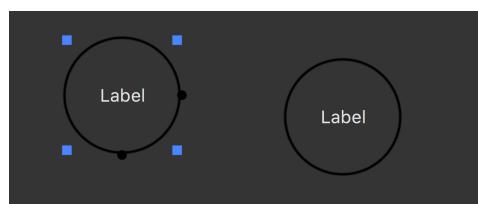


Abbildung 43: Zeichenfläche - Erstellen einer Verbindung

Um eine Verbindung zwischen zwei Elementen zu erstellen, ziehen Sie diese zunächst in das Diagramm. Anschließend wählen Sie mit dem Mauszeiger einen Verbindungspunkt aus und verbinden diesen bei gedrückter linker Maustaste mit einem passenden Punkt

des anderen Elementes. Beachten Sie dabei die korrekte Einhaltung der Notation, da das Programm falsche Verbindungen nicht zulässt.

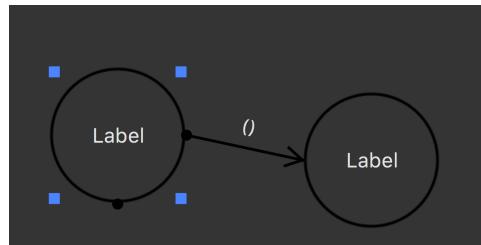


Abbildung 44: Zeichenfläche - Erstellen einer Verbindung II

Die Verbindung ist nun erstellt. Über dem Verbindungspeil lässt sich hier nun ein Datenfluss eintragen. Dazu wählen Sie die Klammer aus und tragen die gewünschten Datentypen ein.

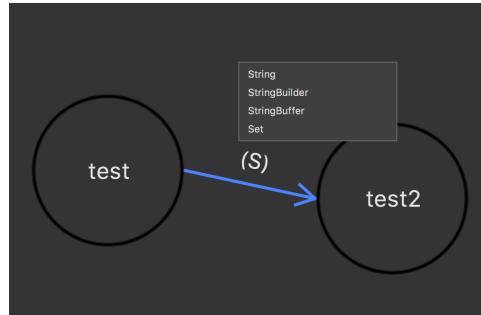


Abbildung 45: Zeichenfläche - Erstellen einer Verbindung III

Die Eingabe wird Ihnen durch eine automatische Autovervollständigung erleichtert. Es reicht den Anfangsbuchstaben des Datentyps anzugeben um eine Liste aller Typen mit jenen Buchstaben zu erhalten, welche sich in der Datentypenliste befinden. Nutzen Sie die Pfeiltasten zu Auswahl und bestätigen Sie mit Enter. Alternativ geben Sie den Typ manuell an.

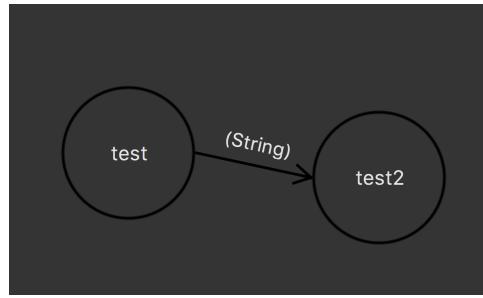


Abbildung 46: Zeichenfläche - Erstellen einer Verbindung IV

Sie sehen eine vollständige Verbindung mit Datenfluss. Mit einem Doppelklick auf das Element können Sie dieses ebenfalls umbenennen.

Um die Zeichenfläche nun zu verschieben, halten Sie die rechte Maustaste gedrückt und bewegen Sie die Maus. Zum Zoomen halten Sie Shift gedrückt und nutzen das Mausrad oder Touchpad.

12.3.2 Fortgeschrittene Programmfunctionen

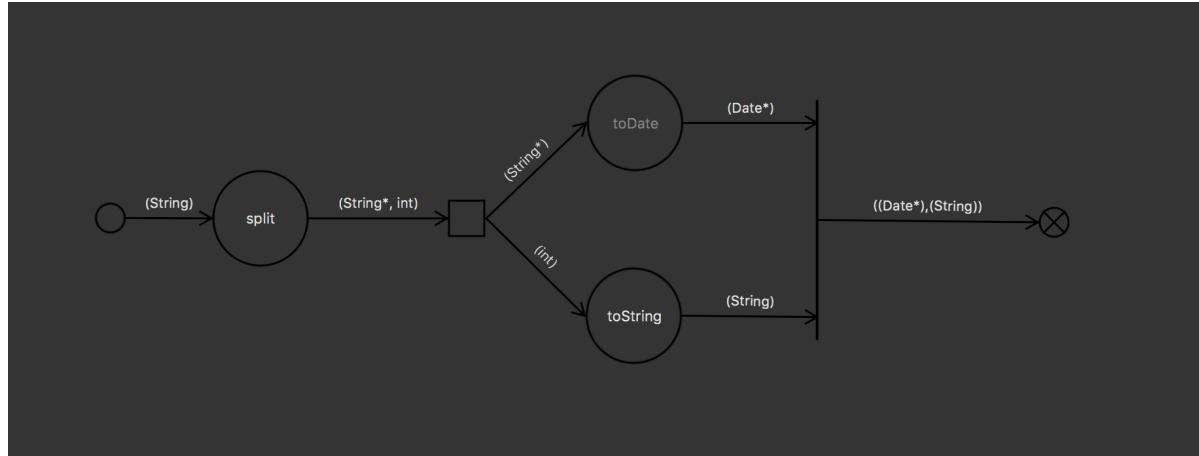


Abbildung 47: Zeichenfläche - Fortgeschrittene Funktionen

Sie kennen nun bereits die grundlegenden Funktionen und Navigation. Bei einem vollständig modellierten Projekt sind folgende Funktionen besonders hilfreich:

- Um in größeren Diagrammen schnell zwischen Elementen wechseln zu können, verwenden Sie die Tastenkombination Strg+Shift+Pfeiltaste rechts/links.
- Wenn Sie, etwa eine Operation in einem Flow-Diagramm, identisch zu einem Diagramm benennen, wird dieses automatisch referenziert. Dies erkennen Sie an der Grau-Färbung des Texts innerhalb des Elements.

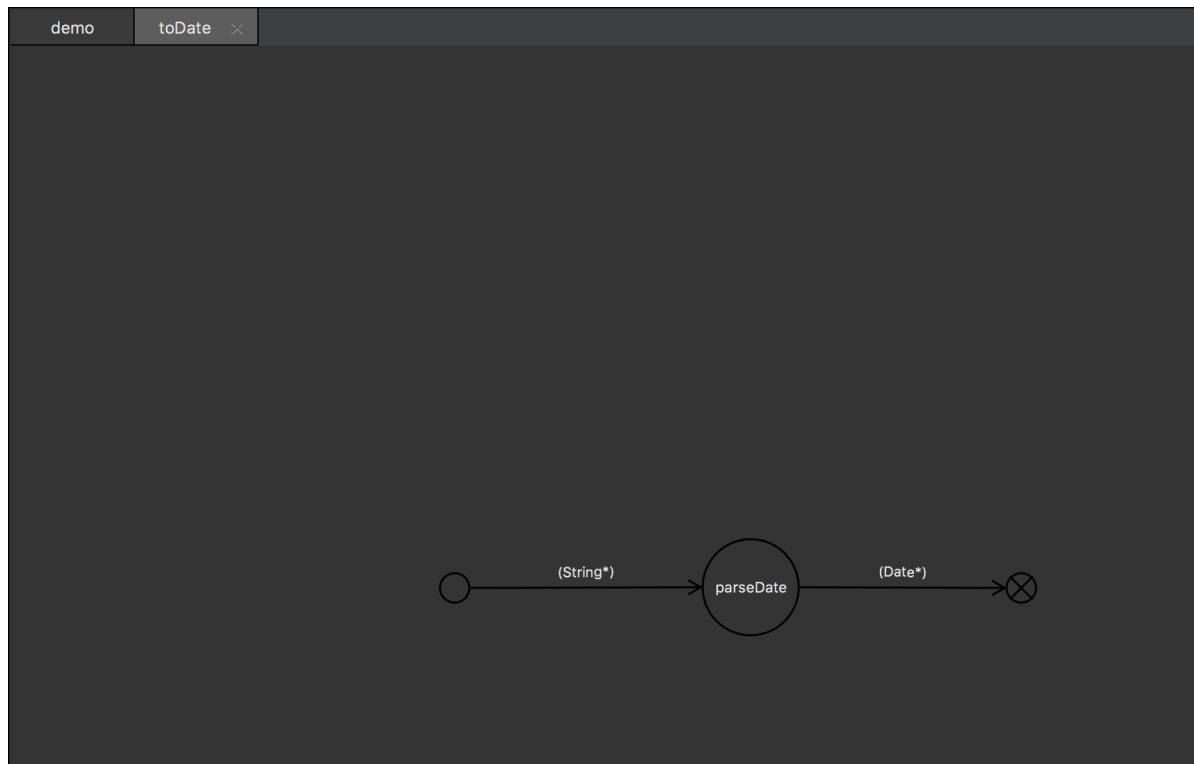


Abbildung 48: Zeichenfläche - Fortgeschrittene Funktionen - Quick Jump

Nutzen Sie Strg+B nach der Auswahl eines referenzierten Elements um direkt in das dazugehörige Diagramm zu springen. In diesem Fall die Referenz der Operation "toDate" in das dazugehörige gleichnamige Diagramm.

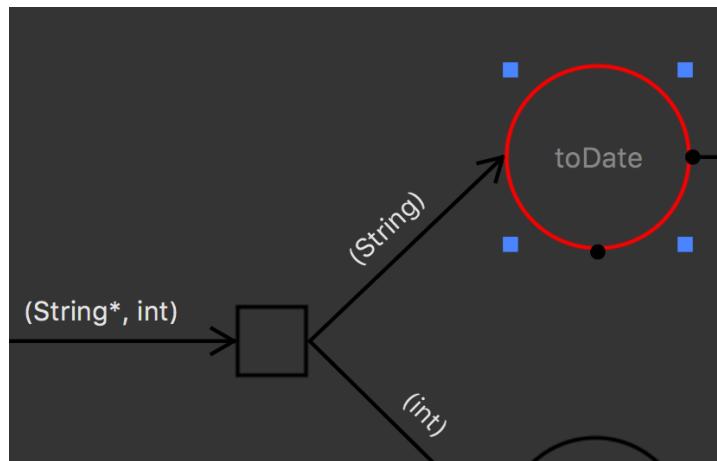


Abbildung 49: Zeichenfläche - Fortgeschrittene Funktionen - Eingabe/Ausgabe Typ

Das Programm überprüft, ob der Eingabe und Ausgabe Typ dem entspricht, was auch im referenzierten Diagramm hinterlegt ist. In diesem Beispiel erfordert "toDate" einen String-Pointer. Wird nur ein String übergeben, markiert sich das Element rot um auf einen Typemismatch aufmerksam zu machen. Wird der Fehler korrigiert oder die dahinter liegende Referenz angepasst, verschwindet der Fehler und damit auch die Färbung.

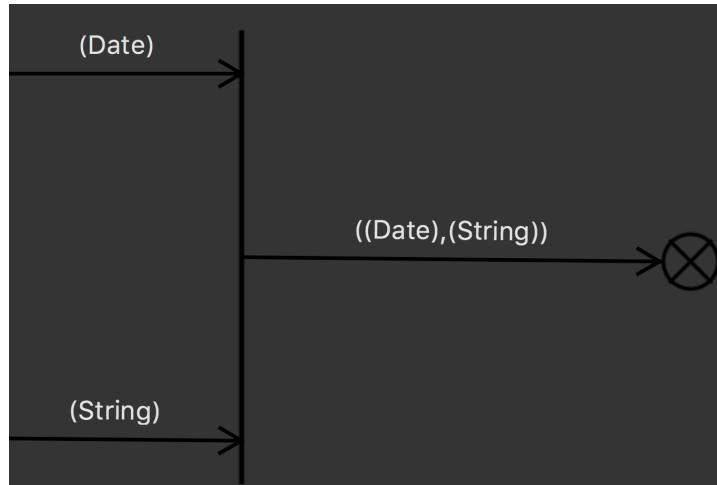


Abbildung 50: Zeichenfläche - Fortgeschrittene Funktionen - Automatisches übernehmen

Falls Sie den Ausgabetyp einer Operation ändern und diese, wie etwa in diesem Beispiel, mit einer anderen zusammenläuft, haben Sie die Möglichkeit die Änderung automatisch durch das Programm übernehmen zu lassen.

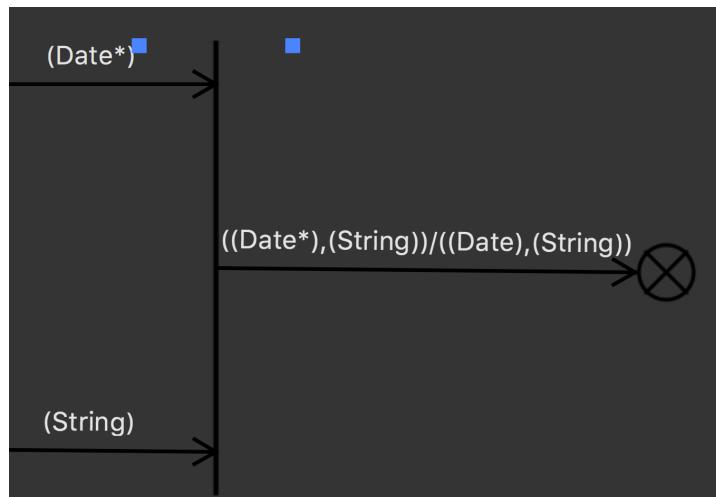


Abbildung 51: Zeichenfläche - Fortgeschrittene Funktionen - Automatisches übernehmen

”Date” wurde hier nun zu ”Date*” geändert. Das Programm zeigt Ihnen bei der zusammenlaufenden Ausgabe nun den aktuellen und vorherigen Status an.

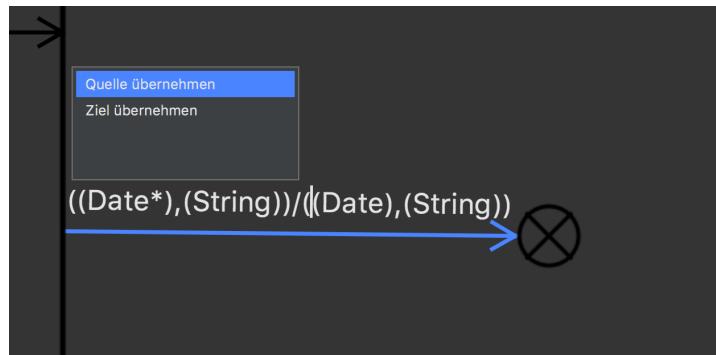


Abbildung 52: Zeichenfläche - Fortgeschrittene Funktionen - Automatisches übernehmen

Drücken Sie nun Strg+Alt+Enter nach Auswahl des gewünschten Feldes um das Kontextmenü aufzurufen. Wählen Sie ”Quelle übernehmen” um automatisch die Änderung übernehmen zu lassen.

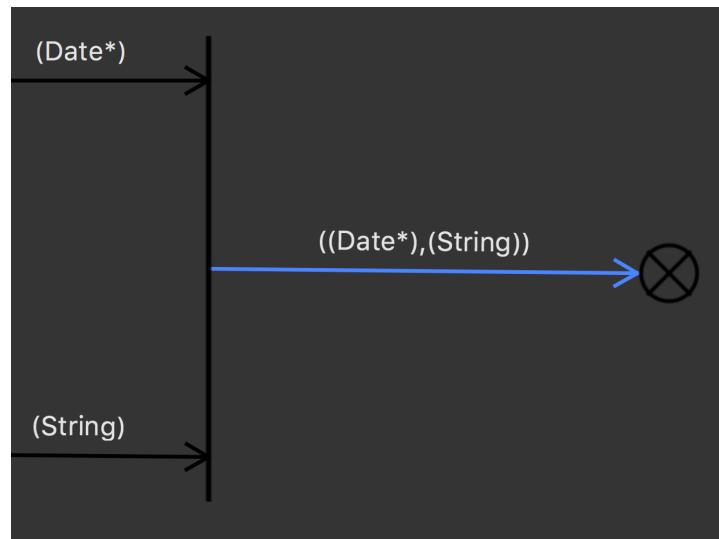


Abbildung 53: Zeichenfläche - Fortgeschrittene Funktionen - Automatisches übernehmen

Die Änderung wurde übernommen und die Typen stimmen jetzt mit der Änderung an der Quelle überein.

12.3.3 Ändern des Programmdesigns und Suche



Abbildung 54: Designauswahl und Suche

Das Programm bietet Ihnen die Möglichkeit, zwischen drei verschiedenen Designs zu wählen. Zum Ändern des aktuellen Designs befindet sich im oberen rechten Teil des Fensters eine Designauswahl. Sie haben hier außerdem die Möglichkeit mit Hilfe des Lupensymbols die Suche aufzurufen (ebenfalls über die Menüleiste möglich, siehe 9.2.1).

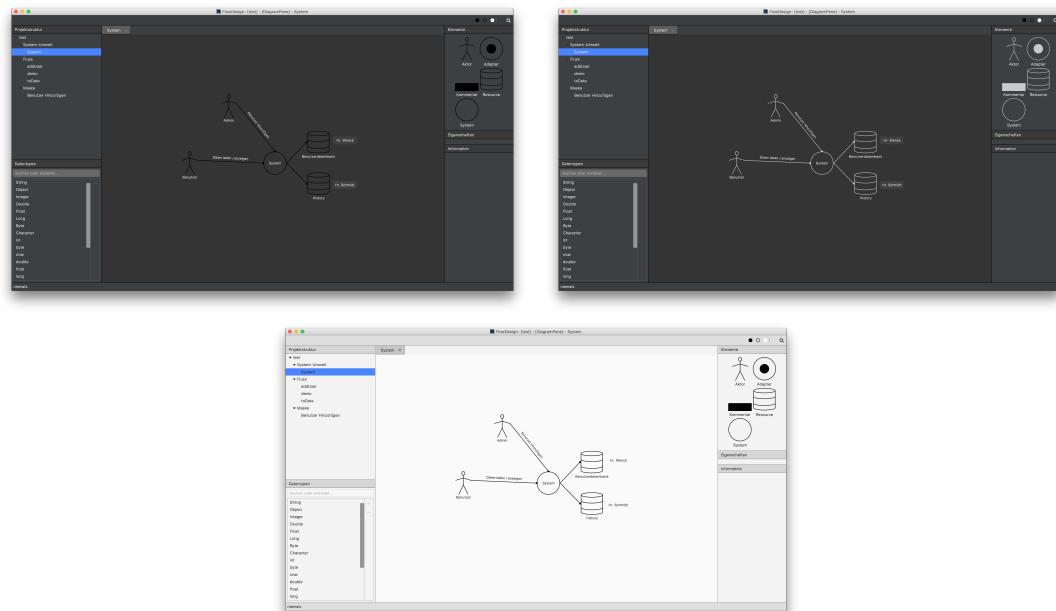


Abbildung 55: Design - Dark, Medium, Bright

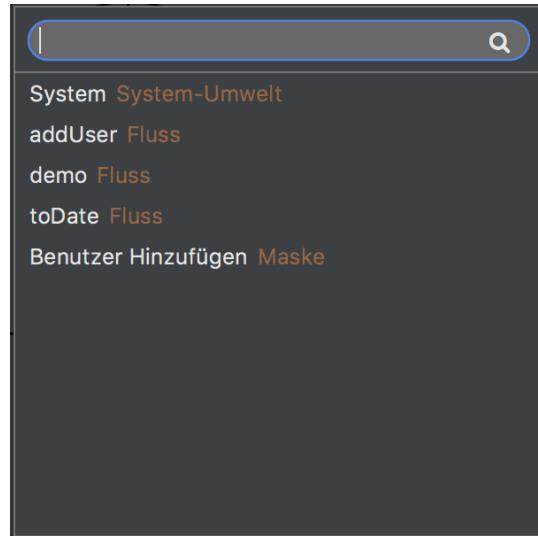


Abbildung 56: Suchfenster

Nach betätigen des Lupensymbols erscheint ein Suchfenster. Es zeigt alle aktuell verfügbaren Diagramme innerhalb Ihres Projekts an. Dabei ist der Name eines Diagramms weiß und dessen Typ orange hervorgehoben. Sie können nun nach einem beliebigen Diagramm suchen, nutzen Sie dabei die Pfeiltasten zur Auswahl und bestätigen Sie mit Enter um direkt in das Diagramm zu gelangen.

13 Schlusswort

Abschließend können wir behaupten, dass das ganze Team von diesem Projekt profitiert hat. Zum Einen konnten wir unser technisches Wissen erweitern, zum Anderen konnten wir uns im Umgang mit Kunden üben.

Alle vorgenommenen Features konnten umgesetzt werden. Lediglich auf die Codegenerierung, welche allerdings nur optional war, mussten wir verzichten, da sich andere Features beim Kundengespräch als wichtiger herausgestellt hatten.

Wir konnten einen leicht erweiterbaren Prototyp entwickeln, der zukünftigen Teams die Möglichkeit bietet direkt anspruchsvollere Funktionen zu implementieren. Da der Code unter eine Open Source Lizenz steht und öffentlich zugänglich ist kann er jederzeit von jedermann modifiziert und angepasst werden.

Abbildungsverzeichnis

1	Start-Element links, End-Element rechts	8
2	Operation-Element	8
3	Operation-Element mit Zustandsvariable	8
4	Operation-Element mit Ressourcenzugriff	9
5	Split-Element	9
6	Verbindungs-Element mit Beispiel	9
7	Portal-Element	10
8	Beispielhafte Verknüpfung mit Kommentarbox	10
9	Beispielhafte Verknüpfung mit Abhängigkeit	11
10	Abhängigkeitsgraph der Module	18
11	Beispiel Java Bean	20
12	Beispiel Diagramm-Klasse	28
13	Beispiel Diagramm-Handler-Klasse	29
14	Diagramm-Handler verfügbar machen	30
15	Beispiel XML-ExampleDiagram-Serializer-Klasse	31
16	Generische Implementation für serialize	32
17	Generische Implementation für deserialize	33
18	Registrieren des neuen Diagramm-Serializers	34
19	Beispiel Diagramm-Element-Klasse	35
20	Beispiel Diagramm-Element-Node	36
21	Beispiel Diagram-Image	38
22	Beispiel Element Factory	39
23	Beispiel Serializer für ein Element	40
24	Registrieren des neuen Element-Serializers	41
25	Auswahlfenster	43
26	Menüleiste unter macOS	44
27	Menüleiste unter Linux/Windows	44
28	Menüleiste - "Datei"	44
29	Menüleiste - "Bearbeiten"	45
30	Menüleiste - "Aktion"	45
31	Menüleiste - "Diagram"	45
32	Menüleiste - "Hilfe"	46
33	Projektbaum	46
34	Projektbaum - Erstellen	46
35	Projektbaum - Bearbeiten	47

36	Datentypenliste	47
37	Datentypenliste - Hinzufügen	48
38	Zeichenfläche - Entfernen	48
39	Zeichenfläche	49
40	Zeichenfläche - Tabs	49
41	Zeichenfläche - Elemente	50
42	Zeichenfläche - Eigenschaften	51
43	Zeichenfläche - Erstellen einer Verbindung	51
44	Zeichenfläche - Erstellen einer Verbindung II	52
45	Zeichenfläche - Erstellen einer Verbindung III	52
46	Zeichenfläche - Erstellen einer Verbindung IV	53
47	Zeichenfläche - Fortgeschrittene Funktionen	53
48	Zeichenfläche - Fortgeschrittene Funktionen - Quick Jump	54
49	Zeichenfläche - Fortgeschrittene Funktionen - Eingabe/Ausgabe Typ	55
50	Zeichenfläche - Fortgeschrittene Funktionen - Automatisches übernehmen	55
51	Zeichenfläche - Fortgeschrittene Funktionen - Automatisches übernehmen	56
52	Zeichenfläche - Fortgeschrittene Funktionen - Automatisches übernehmen	56
53	Zeichenfläche - Fortgeschrittene Funktionen - Automatisches übernehmen	57
54	Designauswahl und Suche	58
55	Design - Dark, Medium, Bright	58
56	Suchfenster	59

Literatur

- [1] Jonathan Giles. *ControlsFX*. englisch. seit mindestens 2013. URL: <http://fxexperience.com/controlsfx/> (besucht am 13.12.2016).
- [2] J. Goll. *Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java*. SpringerLink : Bücher. Springer Fachmedien Wiesbaden, 2014. ISBN: 9783658055325. URL: <https://books.google.de/books?id=ALohBAAQBAJ>.
- [3] M. Just, M. Dausmann und J. Goll. *Stabile und evolvierbare Software durch Einhaltung der SOLID-Prinzipien*. deutsch. 2015. URL: http://www.itdesignersgruppe.com/fileadmin/Inhalte/Studentenportal/Die_SOLID-Prinzipien__Text_1_.pdf (besucht am 07.12.2016).
- [4] K. Knoernschild. *Java Design: Objects, UML, and Process*. Addison-Wesley, 2002. ISBN: 9780201750447. URL: <https://books.google.de/books?id=4pjbgVHzomsC>.
- [5] CCD School. *CheatSheet Flow Design*. 2016. URL: <http://refactoring-legacy-code.net/wp-content/uploads/2016/10/CheatSheet-Flow-Design.pdf>.
- [6] Wikipedia. *Liskovsches Substitutionsprinzip — Wikipedia, Die freie Enzyklopädie*. 2016. URL: https://de.wikipedia.org/wiki/Liskovsches_Substitutionsprinzip.